ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Παραλληλοποίηση Κώδικα Βρόχων σε Αρχιτεκτονικές μη Ομοιόμορφης Προσπέλασης Μνήμης (NUMA)

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

Μαρία Γ. Αθανασάκη

Αθήνα, Δεκέμβριος 2005

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

ΠΡΑΚΤΙΚΟ ΕΞΕΤΑΣΗΣ ΔΙΔΑΚΤΟΡΙΚΗΣ ΔΙΑΤΡΙΒΗΣ

της

Μαρίας Γ. Αθανασάκη

Διπλωματούχου Ηλεκτρολόγου Μηχανικού και Μηχανικού Υπολογιστών Ε.Μ.Π. (2001)

Παραλληλοποίηση Κώδικα Βρόχων σε Αρχιτεκτονικές μη Ομοιόμορφης Προσπέλασης Μνήμης (NUMA)

Τριμελής Συμβουλευτική επιτροπή: Παναγιώτης Τσανάκας, επιβλέπων
Γεώργιος Παπακωνσταντίνου
Νεκτάριος Κοζύρης

Εγκρίθηκε από την επταμελή εξεταστική επιτροπή την ....

....	....	....
Π. Τσανάκας	Γ. Παπακωνσταντίνου	Ν. Κοζύρης
Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Επίκ. Καθηγητής Ε.Μ.Π.
....		....
Ε. Ζάχος		Τ. Σελλής
Καθηγητής Ε.Μ.Π.		Καθηγητής Ε.Μ.Π.
....		....
Α. Συμβώνης		Θ. Θεοχάρης
Αναπλ. Καθηγητής Ε.Μ.Π.		Καθηγητής Πανεπιστημίου Αθηνών

Αθήνα, Δεκέμβριος 2005

....

Μαρία Γ. Αθανασάκη

Διδάκτωρ Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

NATIONAL TECHNICAL UNIVERSITY OF ATHENS
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING
DEPARTMENT OF COMPUTER SCIENCE
COMPUTING SYSTEMS LABORATORY

Parallelization of Nested Loop Codes for Non-Uniform Memory Access (NUMA) Architectures

PHD THESIS

Maria G. Athanasaki

Athens, Greece, December 2005

....

Maria G. Athanasaki
School of Electrical and Computer Engineering, National Technical University of Athens, Greece

Περίληψη

Η διατριβή αυτή προσθέτει ένα λιθαράκι ακόμη στη λύση του προβλήματος της παραγωγής παράλληλου κώδικα για προγράμματα που περιέχουν τέλεια φωλιασμένους βρόχους. Στη σύγχρονη βιβλιογραφία, η παραλληλοποίηση τέτοιων δομών έχει κατ' αρχήν βασιστεί στο μετασχηματισμό tiling, ή αλλιώς, μετασχηματισμό υπερκόμβων. Έχουν προταθεί μέθοδοι για την αυτόματη μετατροπή του σειριακού κώδικα σε παράλληλο. Επίσης, έχουν προταθεί εναλλακτικές λύσεις για το χρονισμό μεταξύ επικοινωνίας και υπολογισμών. Όλες αυτές οι λύσεις, όμως, αφορούν την εκτέλεση του τελικού προγράμματος σε μία απλή συστοιχία (cluster) υπολογιστών.

Σήμερα, τα πλέον ισχυρά μηχανήματα, δεν αποτελούνται από απλούς υπολογιστές, αλλά από πολυ-επεξεργαστικές μονάδες (δείτε τη λίστα των 500 πιο ισχυρών υπολογιστών του κόσμου του Νοεμβρίου 2004). Το ιδιαίτερο χαρακτηριστικό τους είναι ότι οι επεξεργαστές του ίδιου κόμβου βλέπουν κοινή μνήμη, ενώ όσοι βρίσκονται σε διαφορετικούς κόμβους επικοινωνούν αναγκαστικά με ανταλλαγή μηνυμάτων. Πρόκειται, δηλαδή για διεπίπεδες αρχιτεκτονικές. Μέχρι στιγμής δεν είχε προταθεί κάποια λύση που να λαμβάνει υπόψη την ανομοιομορφία αυτή. Όμως, η ανταλλαγή μηνυμάτων ακόμη και ανάμεσα στους επεξεργαστές που έχουν άμεση πρόσβαση στην ίδια μονάδα μνήμης, αποτελεί σημαντική απώλεια χρόνου για το τελικό πρόγραμμα. Το πρόβλημα αυτό, λοιπόν, αντιμετωπίζεται αποδοτικά στην παρούσα διατριβή. Επιτυγχάνουμε την μέχρι στιγμής βέλτιστη αξιοποίηση του εύρους ζώνης και των δυνατοτήτων των καρτών δικτύου. Ταυτόχρονα, μπορούμε απλά και με σαφήνεια να ορίζουμε μία χρονική δρομολόγηση των υπερκόμβων, παρά την ανομοιόμορφη επικοινωνία μεταξύ τους.

Ένα άλλο θέμα που δεν είχε μέχρι στιγμής αντιμετωπιστεί είναι αυτό της κατανομής των tiles, ή υπερκόμβων σε επεξεργαστές. Στη βιβλιογραφία, όλες σχεδόν οι προσεγγίσεις θεωρούν είτε ότι υπάρχει απεριόριστος αριθμός επεξεργαστών, είτε ότι το μέγεθος των tiles επιλέγεται ώστε οι διαθέσιμοι επεξεργαστές να είναι αρκετοί. Όμως, σκοπός του μετασχηματισμού υπερκόμβων (tiling) δεν είναι μόνο η παραλληλοποίηση του κώδικα, αλλά και η βελτιστοποίηση της τοπικότητας των αναφορών σε δεδομένα της μνήμης. Στην περίπτωση αυτή, οι δύο στόχοι οδηγούν σε αντικρουόμενα αποτελέσματα. Επειδή ο χρόνος που χρειάζεται για την προσπέλαση δεδομένων,

που δε βρίσκονται στην γρήγορη μνήμη του συστήματος, δεν είναι αμελητέος (μπορεί να είναι συγκρίσιμος, ή ακόμη και πολλαπλάσιος του χρόνου που χρειάζεται για την επεξεργασίας τους), δεν θα έπρεπε να παραμεληθεί η παράμετρος αυτή κατά την επιλογή του μετασχηματισμού υπερκόμβων. Στη διατριβή αυτή, λοιπόν, διερευνούμε μεθόδους για την κατανομή των υπερκόμβων στις υπολογιστικές μονάδες, σε περίπτωση που ο μετασχηματισμός υπερκόμβων και το πλήθος τους είναι ήδη δεδομένο. Προκειμένου οι μέθοδοι αυτοί να μπορούν να ενσωματωθούν αποδοτικά σε ένα εργαλείο αυτόματης παραγωγής κώδικα, εστιάζουμε την προσοχή μας σε μεθόδους στατικής κατανομής των υπολογισμών, οι οποίοι παρουσιάζουν κάποια κανονικότητα.

Λέξεις-κλειδιά: Μετασχηματισμός tiling, Μετασχηματισμός υπερκόμβων, Ομαδοποίηση υπερκόμβων, Αλληλοεπικάλυψη επικοινωνίας και υπολογισμών, Υπερεπίπεδα, Συστοιχίες πολυ-επεξεργαστικών μονάδων, Περιορισμένος αριθμός κόμβων.

Abstract

This thesis adds some intuition and some practical solutions to the well-studied problem of parallelizing nested `for`-loops. In literature, parallelization of such code segments has been based on supernode, or tiling transformation. There have been proposed some methods for the automatic transformation of sequential code into parallel one. In addition, the timing between communication and computation has been studied. However, these solutions concern the execution of the final parallel code onto a cluster of single CPU nodes.

Nowadays the most powerful computing systems are consisted of multiprocessor units (see the Top 500 supercomputer list for November 2004). In such supercomputers, processors within the same node can directly access the same memory data, while processors in different nodes should communicate via message passing. No solution had been proposed so far to overcome this heterogeneity. Message passing among processors inside the same SMP node implies a significant communication overhead. The above mentioned problem is efficiently alleviated in this thesis. We pursue and achieve a proper utilization of the bandwidth and the possibilities of the network cards. At the same time, we can simply and explicitly define a time scheduling of tiles, in spite of the heterogeneous communication patterns.

Another issue, that had not been thoroughly examined so far, is the allocation of tiles, or supernodes to processors. Almost all approaches in literature consider either an unlimited number of processors, or that tile size is properly selected to fit the existing architecture. However, tiling has not been used only for parallelization, but also for achieving cache locality of data memory references. These two goals conflict with each other, concerning the tile size selection. Since the time needed for accessing data in main memory is not at all negligible (it may be comparable, or even a multiple of time needed for processing data), this parameter should not be left out when selecting a tiling transformation. In this thesis, we investigate certain techniques for allocating tiles to computing nodes, in case the tiling transformation, the size of the tile space and of the architecture are given. We consider static, regular techniques of allocation, in order to be able to incorporate them efficiently into an automatic parallel code generation tool.

Keywords: Supernodes, Loop tiling, Tile grouping, Overlapping communication, Pipelined Schedules, Hyperplanes, Clusters of SMPs, Fixed number of nodes.

Contents

List of Figures

List of Tables

Αντί Προλόγου

Η παρούσα διδακτορική διατριβή εκπονήθηκε στον Τομέα Τεχνολογίας Πληροφορικής και Υπολογιστών, της Σχολής Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, του Εθνικού Μετσόβιου Πολυτεχνείου. Περιλαμβάνει την έρευνα και τα συμπεράσματα που προέκυψαν κατά τη διάρκεια των μεταπτυχιακών σπουδών μου στο Εργαστήριο Υπολογιστικών Συστημάτων της σχολής αυτής.

Στο σημείο αυτό θα ήθελα να εκφράσω τις ειλικρινείς ευχαριστίες μου σε ένα πλήθος ανθρώπων, που με βοήθησαν ουσιαστικά στην πραγματοποίηση της εργασίας αυτής. Πρώτα από όλους θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή μου, Παναγιώτη Τσανάκα, επειδή, όντας προπτυχιακή φοιτήτρια, εκείνος πρώτος με έφερε σε επαφή με το χώρο στον οποίο αργότερα αποφάσισα να συνεχίσω τις σπουδές μου ως μεταπτυχιακή φοιτήτρια. Τον ευχαριστώ ιδιαίτερα επειδή με επέλεξε για συνεργάτη του, για τις χρήσιμες γνώσεις που μου μετέδωσε, για την οικειότητα που μου εμπνέει, αλλά περισσότερο για την εμπιστοσύνη που μου έδειξε σε όλα τα θέματα.

Επίσης, θα ήθελα να ευχαριστήσω θερμά τον καθηγητή Γεώργιο Παπακωνσταντίνου, μέλος της τριμελούς συμβουλευτικής επιτροπής μου και επικεφαλή του εργαστηρίου, για την αγάπη του, τις συμβουλές του, για τη διάθεσή του να ασχοληθεί με κατανόηση με οποιοδήποτε πρόβλημά μας.

Ιδιαίτερη αναφορά θα ήθελα να κάνω στο τρίτο μέλος της συμβουλευτικής επιτροπής μου, τον επίκουρο καθηγητή Νεκτάριο Κοζύρη. Ήταν ο άνθρωπος που καθόρισε την κατεύθυνση της έρευνάς μου, που ασχολήθηκε ουσιαστικά με την πορεία και τα προβλήματα που αντιμετώπισα κατά τη διάρκεια των σπουδών μου, που ανέκαμπτε το ηθικό μου σε δύσκολες ερευνητικά περιόδους, που φρόντισε να έχω διαθέσιμο τον εξοπλισμό που χρειαζόμουν. Πέρα, όμως, από τα καθαρά επιστημονικά θέματα, δίπλα του πήρα αξέχαστα μαθήματα ζωής με τις πολύωρες συζητήσεις που είχαμε μαζί του. Μου έμαθε να πιστεύω στον εαυτό μου, να θέτω και να πετυχαίνω στόχους, να είμαι πιο ανοιχτή απέναντι σε ανθρώπους με εντελώς διαφορετική νοοτροπία και επιδιώξεις. Όλα αυτά είναι ιδιαίτερα σημαντικά, όχι μόνο για την επαγγελματική αποκατάστασή ενός ανθρώπου, αλλά και για την οικογενειακή και κοινωνική ζωή του.

Βέβαια, πέρα από τους καθηγητές μου, θα ήταν πολύ μεγάλη παράλειψη να μην αναφερθώ και στα υπόλοιπα μέλη του εργαστηρίου. Κατ' αρχήν, ο Γιώργος Γκούμας ήταν ένας μικρότερος καθηγητής μου. Τον ευχαριστώ ιδιαίτερα, όχι μόνο για τη συμβολή του στην ερευνητική δουλειά μου, αλλά και γιατί με το παράδειγμά του σκιαγράφησε το πρότυπο συμπεριφοράς ενός ανθρώπου που έχει βρει τη σωστή ισορροπία μεταξύ επαγγελματικής και κοινωνικής ζωής, που ξέρει να δίνει στα πρόσωπα και τις καταστάσεις την προσοχή που τους αρμόζει. Ιδιαίτερη για μένα ήταν και η σχέση που ανέπτυξα με το Νίκο Δροσινό. Τον ευχαριστώ, όχι μόνο για τη συνεργασία μας σε ερευνητικά θέματα, από την οποία αποκόμισα πολύτιμες γνώσεις, αλλά και για τις συζητήσεις μας σε θέματα που κουβεντιάζονται μόνο μεταξύ πραγματικών φίλων.

Στη συνέχεια, οφείλω να ευχαριστήσω θερμά όλα τα παιδιά με τα οποία συνεργάστηκα, τον Άρη Σωτηρόπουλο, το Γιώργο Τσουκαλά, το Βαγγέλη Κούκη. Χωρίς τη συμβολή τους, η διδακτορική διατριβή μου θα ήταν σίγουρα πολύ φτωχότερη. Δυστυχώς, θα μακρηγορούσα πολύ αν στεκόμουν σε κάθε ένα από τα μέλη του εργαστηρίου ξεχωριστά. Παρόλα αυτά, πρέπει να αναφέρω ότι ο Αντώνης Ζήσιμος, ο Αντώνης Χαζάπης, ο Κορνήλιος Κούρτης, ο Γιώργος Βερυγάκης, ο Νίκος Αναστόπουλος, προσθέτουν ο καθένας με τον τρόπο του, με τις γνώσεις και το χαρακτήρα του, μία ιδιαίτερη νότα στην κουλτούρα του εργαστηρίου. Ασφαλώς, δεν πρέπει να εννοηθεί ότι τα παιδιά που δεν αναφέρθηκαν ονομαστικά έχουν μικρότερη συμβολή στο φιλικό κλίμα συνεργασίας και στην περιρρέουσα γνώση του εργαστηρίου.

Επίσης, ένα τεράστιο ευχαριστώ για αναρίθμητους λόγους οφείλω στην αδελφή μου και υποψήφια διδάκτορα του εργαστηρίου, Ευαγγελία Αθανασάκη. Όχι μόνο γιατί από τα παιδικά μου χρόνια ήταν η καλύτερη φίλη μου, όχι μόνο επειδή με ακολούθησε και με συντρόφευσε σε όλα τα σημαντικά βήματα της ζωής μου, όχι μόνο επειδή ήταν πάντα ο πρώτος άνθρωπος που θα ασχολούνταν με οποιαδήποτε ανησυχία μου. Αλλά και επειδή με την ενεργή παρουσία της καθόρισε, σε βαθμό παραπλήσιο με τους γονείς μου, την προσωπικότητά μου.

Τέλος, ευχαριστώ θερμά το Κοινωφελές Ίδρυμα Αλέξανδρος Ωνάσης για την οικονομική στήριξη που μου παρείχε μέσω μίας υποτροφίας μεταπτυχιακών σπουδών.

Η εργασία αυτή αφιερώνεται στην οικογένειά μου και σε όσους αποτελούν ένα ευτυχές αναπόσπαστο κομμάτι της ζωής μου.

1

Introduction

1.1 Motivation

Tiling, or supernode transformation has been widely used in parallel processing for restructuring nested `for`-loop code segments. When applying tiling, neighboring iterations are grouped together into a tile, or supernode. Thereupon, each tile is treated as one computation unit. That is, we schedule tiles instead of iterations, we decide which tiles will be assigned to a processor and so on. Therefore, we achieve to decrease the total communication load of the code segment as follows:

- Assuming that iterations of the initial code segment may be assigned to any processor of the parallel architecture, the communication load implied may be vast in comparison to the computation load. When applying tiling, we force neighboring iterations to be executed onto the same processor. Therefore, the communication requirements among them are eliminated.

- In message passing interfaces, designed for distributed memory computing systems, the cost of initializing a data transfer is not negligible. When applying tiling, apart from grouping iterations, we also group the resulting data transfers. Thus, we may initialize only one message per tile per communication direction, reducing in this way the number of messages and the communication startup cost.

A lot of work has been conducted in this area, concerning the selection of the optimal tiling transformation. Researchers have concluded that, on the one hand, rectangular tiling is simple. Thus, both the application of the tiling transformation and the execution of the final tiled code is efficient [TX00]. On the other hand, non-rectangular tiling may be more appropriate for a specific code segment [HS02], [HCF03]. Thus, if it is properly applied, it may give the peak performance [GDAK02a].

As far as parallel processing is concerned, the size and shape of tiles is mainly selected so as to minimize the communication overhead. The resulting tiling transformation seems to be the same when either a distributed [Xue97a] or a shared memory [RR02] system is aimed. Consequently, when a multilevel parallel architecture is involved, the optimal tiling transformation is just the same.

However, when applying a tiling transforation, tile shape and size are not the only concerns. One should also determine a time schedule, for both computations and communication. This problem has also been addressed when either a distributed or a shared memory architecture is involved. It has not been addressed for a multilevel parallel architecture, such as a cluster of shared memory multiprocessors (SMPs). In this thesis, a time schedule is produced, which takes into account the communication requirements among processors, which may reside either in the same or in different SMP nodes.

Once a tiling transformation has been applied onto a nested `for`-loop code segment, and a time scheduling has been produced, one may assume that it can be really implemented onto a parallel architecture. In fact, this is not always true. The number of processors of an existing platform may be less than the number of processors required for the application of a time schedule. Although in literature a lot of papers deal with the problem of scheduling onto a fixed number of processors, very few of them are applicable on nested `for`-loops, that cannot be partitioned into independent sub-spaces. In this thesis five alternative static schemes, for scheduling a tile space and assigning tiles to the processors of an existing parallel architecture, are proposed.

1.2 Related Work

A few years ago the constant increase of the execution speed of programs was mainly based on the clock frequency increase. In 1980's, both academia and industry realized that it was meaningless to further promote the clock speed if they could not feed the processor with data from memory [PH94], [HP03]. Their efforts concentrated onto minimizing the distance between the processor and memory data, using cache memories. They went on increasing the clock speed, but at the same time they increased the size and bandwidth of caches, they improved the algorithms used for storing and searching data in them.

Nowadays, technology seems to have approached the core. A further increase of either the clock speed or the cache bandwidth is sustained by physical restrictions, such as the speed of light and the minimum distances that should exist inside a chip, so as electrical signals do not interfere with each other. Therefore, the only notion that can supply computer performance with a thrust seems to be parallel processing.

However, without an intervention from the programmer, parallel processing may have an impact only when several independent programs are to be executed simultaneously. A minor intervention is required when a single program can be partitioned into independent or loosely dependent tasks. What happens when we are interested in speeding up a single program, which cannot be partitioned into independent regions? Then, a thorough analysis of data dependences [Ban88], [Pug92] is required, so as to decide which tasks could be efficiently parallelized.

Nested `for`-loops can be placed among the most critical code segments, which deserve parallelization. They usually impose a significant overhead to the total program execution, since they iterate many times over the same statements. In order to achieve the maximum acceleration, one of the key issues to be considered is minimization of the communication overhead. Papers elaborating on this issue can be divided into two main categories corresponding to fine grain parallelization and coarse grain parallelization.

As far as fine grain parallelism is concerned, the communication overhead is reduced by applying methods that group together neighboring chains of iterations [KCN91], [SC95], while preserving the optimal hyperplane schedule [DGK+00], [ST91], [TKP00]. The objective of partitioning the initial iteration space into chains of iterations has always been the minimization of inter-chain dependences. Thereupon, some chains may be grouped together and executed in the same processor, aiming again to reduce the inter-processor dependences.

As far as coarse grain parallelism is concerned, researchers have dealt with the problem of alleviating the communication overhead by applying the supernode or tiling transformation. Supernode partitioning of the iteration space was initially proposed by Irigoin and Triolet in [IT88]. They introduced the initial model of loop tiling and gave conditions for a tiling transformation to be valid. Later, Ramanujam and Sadayappan in [RS92] showed the equivalence between the problem of finding a set of extreme vectors for a given set of dependence vectors and the problem of finding a tiling transformation that produces valid, deadlock-free tiles. The problem of determining the optimal shape was surveyed, and more accurate conditions were also given by others, as in [BDRR94], [HS02], [HCF03]. Some of these approaches aim at minimizing the amount of data transferred through a message passing interface [Xue97a]. Some more of them are applicable on a shared memory architecture and pursue the minimum amount of data to be accessed by more than one processors [AKN95], [RR02]. The rest of them attempt to minimize the time each processor remains idle waiting for the necessary data to be available, before going on with the computations assigned to it [DDRR97], [HCF97], [HCF99]. All three approaches result to the same mathematical formulas for the calculation of the optimal tiling transformation.

Scheduling tiled iteration spaces onto parallel architectures is another important issue, which has been partially addressed in literature. Dion et al. [DRR96] and Rastello et al. [RRP03] have reduced the total run-time by properly scheduling the iterations inside a tile. They assume that a tile execution is non-atomic and each data element is sent to processors that will need it, as soon as it is computed. Although such an approach may be practical on a VLSI processor array, it will not be efficient on a modern cluster, where the startup latency of a message cannot be ignored, imposing coarse-grain communication.

Although scheduling of tasks on a cluster of workstations seems to be a well elaborated idea [CKE+04], in fact very few approaches have taken into account the regularity of nested `for`-loops. Several of them [SG97], [Sak97], [HP96] deal with the distribution of loop iterations to processors, in special cases, when the iteration space can be decomposed to regions, that can be parallelized with no communication or synchronization among processors. However, this is not always the case. As concluded by [LL98], the dependences among iterations may not allow the application of such a scheduling. In [ML94] a run-time scheduling is presented, which minimizes communication and synchronization overhead. In [ID98], [ZLP97], a dynamic load-balancing scheduling algorithm is presented, with a combination of compile-time and run-time support (hybrid compile and run-time process). However, as argued in [TN93], dynamic, or run-time scheduling achieves a better load balance when the computation load of iterations is unevenly distributed. In addition, it is applicable if the loop bounds are unknown at compile time. Static, or compile-time scheduling is more appropriate for uniformly distributed loops, following the algorithmic model of this thesis.

As far as the execution of tiles on a cluster of PCs is concerned, all conventional approaches [ABRY03], [ABR96], [HS98], [OSKO95], [RS92] consider that each processor executes all tiles along a specific dimension, by interleaving computation and communication phases. All processors first receive data, then compute, and finally send result data to neighbors in explicitly distinct phases, according to the hyperplane scheduling vector. Taking into account that modern network interfaces allow for concurrent communication and computation, in [GSK01] an alternative method for the problem of scheduling the tiles to single CPU nodes was proposed. The proposed method acts like enhancing the performance of a processor's datapath with pipelining [PH94], because a processor computes its tile at k time step and concurrently receives data from all neighbors to use them at $k+1$ time step and sends data produced at $k-1$ time step. Such a pipelined execution scheme was proven [STK02] to nearly double the performance of the algorithms, provided that we use modern NICs (Network Interface Cards), capable of performing communication without annoying the CPU, and advanced communication protocols (i.e. VIA) with Zero-Copy [CTHI98], DMA support and User-Level [Blu96] characteristics.

Although the tiling transformation had been so widely studied, in practice it was almost unattainable to implement the proposed methods in real applications. The overhead for producing the parallel code was almost prohibitive. In [AL93], Amarasinghe and Lam presented a method for automatically producing parallel SPMD code, based on the mathematical representation of the iteration space, the data space and the communication data, using a set of inequalities. In [TX00], Tang and Xue presented a complete framework for producing SPMD code for distributed memory parallel architectures. However, their approach concerns only rectangular tiling transformations. Finally, in [GAK03], [GDAK02a] a complete framework has been presented for automatically producing parallel code for arbitrarily tiled nested `for`-loops.

This method, apart from enhancing the efficiency of the final parallel code, aims at reducing the overhead of the automatic parallelization.

1.3 One step ahead: What do we need?

Nowadays the most powerful computing systems are consisted of multi-level parallel architectures, such as a cluster of Shared-Memory Multiprocessors. The top 5 computing systems announced in the 2004 Supercomputer Conference (SC2004) [TOP] in Pittsburgh (BlueGene/L, Columbia, Earth Simulator, MareNostrum, Thunder), are all based on a multi-level parallel architecture (see, for example, Figures 1.1 and 1.2).

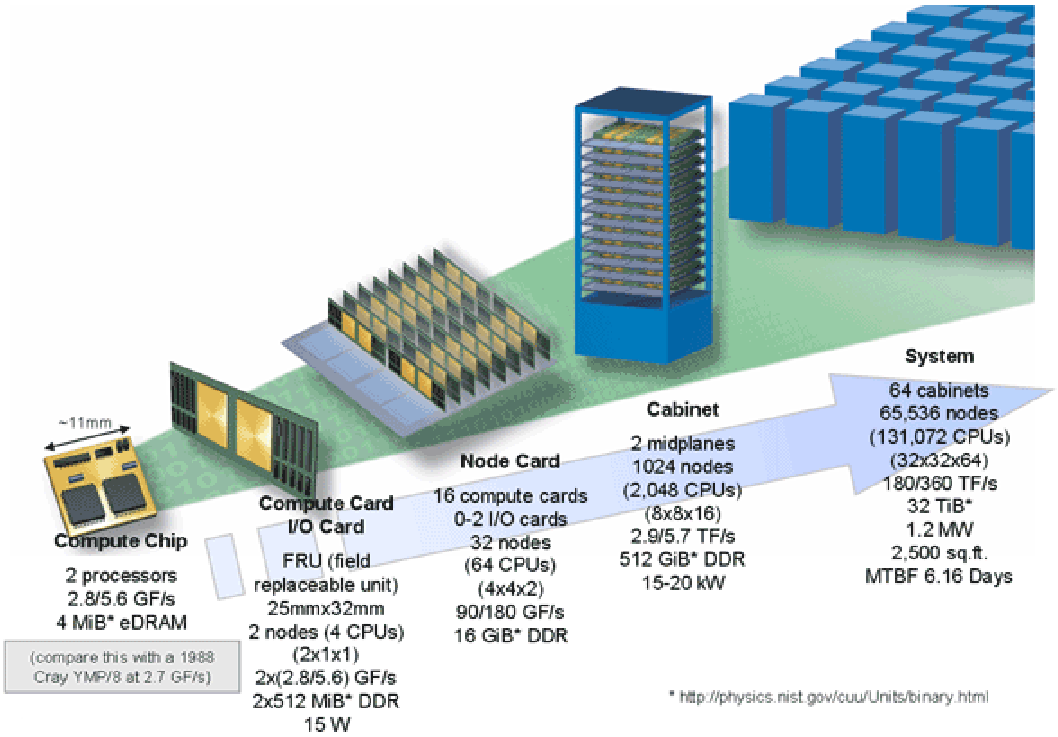


Figure 1.1: The BlueGene/L Architecture - No 1 in the 24th Top500 Supercomputer list

The method presented in [GSK01], [STK02] had been applied only on clusters of single CPU nodes. If applied on a cluster of SMP nodes (Symmetric Multi-Processors), it could not take into consideration the fact that, among processors of the same node, which can directly communicate with each other through the node's shared memory, there is no need for message interchange, in order to exchange data. This fact has not been taken into account in [MA01] either, which aims at scheduling tiles on a cluster of SMP nodes. The result of such a consideration may be unnecessary transfers from the processing unit to the network card and vice versa, which will consume a portion of the intra-node communication bandwidth. In the best case, when the compiler can detect and prevent such unnecessary communication between the processor and the network card, it will not evict unnecessary transfers among the shared and private space of

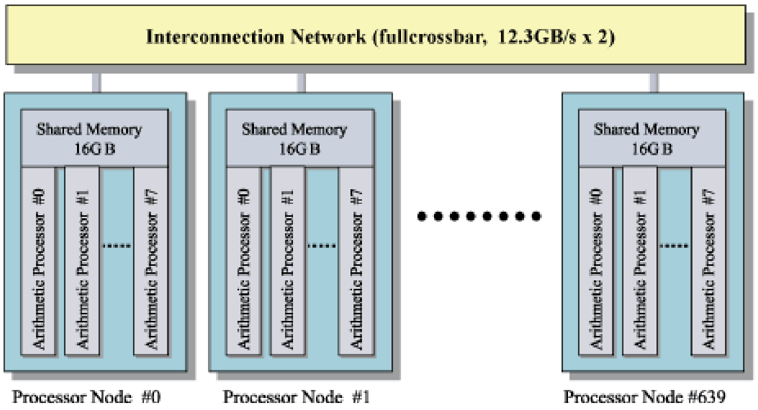


Figure 1.2: The Earth Simulator Architecture - No 3 in the 24th Top500 Supercomputer list

threads inside the same SMP node [DK04].

In this thesis, as in [AST+05], [ASTK02b], [ASTK02a], the method proposed in [GSK01], [STK02] is applied on clusters of SMP nodes. For this purpose, we group together tiles, which should be simultaneously executed by processors of the same node. Thus, we annihilate the need for communication among processors of the same node. In the sequel, in order to schedule the groups of tiles, which have arisen, we can make use of the overlapping communication-computation model, proposed in [GSK01], [STK02].

Unfortunately, the subsequent execution scheme (similar to its parent schemes proposed in [HS98], [GSK01] and the automatic schedules produced when using a code generation tool [GDAK02a]) preassumes an unlimited number of processing nodes, or that the tile size has been selected so that the number of nodes needed is less than or equal to the nodes available. Of course, it is not always true. The tile size may often be selected so as to minimize the communication overhead [Xue97a], [AKN95], [RR02] or maximize memory data references locality [KRC99], [LRW91], [WL91a], [PHP03], [MHCF98]. Thus, we need an efficient method to allocate the tasks to a predefined number of processors. In this thesis, as in [AKK04], [AKK03], some different assignment schemes for scheduling tiles onto a cluster with a fixed number of SMP nodes, will be proposed.

1.4 Thesis Contribution

The contribution of this thesis, can be mainly focused on the following two issues:

1. A theoretic model is supplied for scheduling tiles onto a cluster of SMP nodes, using either the overlapping or the non-overlapping execution policy, as described in [GSK01], [STK02], [HS98]. This is attained by grouping together tiles, which should be simultaneously executed by processors of the same node. Thus, the need for communication among processors of the same node is annihilated. They should only synchronize with each other

using a barrier or a semaphore. In addition, the subsequent communication among processors in different SMP nodes can be similarly grouped, which further reduces the overall communication overhead of a code segment.

2. In order to apply all above mentioned techniques and automatic code generation tools [Gou03] onto a cluster with a fixed number of nodes, five alternative assignment schemes for scheduling tiles are proposed. The advantages and disadvantages of each one are theoretically and experimentally investigated. Thus, the guidelines for selecting the appropriate assignment scheme for each tile space, are provided.

1.5 Thesis Overview

In Chapter 2 of this thesis, some basic preliminary concepts and the mathematical background required for the comprehension of our methodology are presented. First of all, some mathematical symbols used throughout the thesis are defined. Then, we briefly describe the model of algorithms, which can be parallelized using the proposed techniques. In the sequel, some basic concepts from parallel processing, such as dependences and time scheduling, are described. In addition, some loop transformations, which have been widely used in compiler optimizations, are briefly discussed. They are divided into linear and non-linear transformations. Among non-linear loop transformations, we emphasize the tiling transformation, which will be used throughout the rest of this thesis. Finally, we outline the non-overlapping [HS98] and the overlapping [GSK01] execution policies, which constitute the base for the application of our theory.

In Chapter 3, a methodology for the construction of a tool, which can automatically produce parallel tiled code, is discussed. Special care is taken, so as the final tool to be efficient in consideration of both the time needed for the generation of the parallel code and the quality of the code produced. The efficiency at compile-time is enhanced by a reduction of the inequalities describing the tile space, through a proper expansion of the initial space boundaries. The efficiency at run-time is achieved by a transformation of the tile iteration space into a rectangular one. Finally, as far as the communication among processors is concerned, an enhancement of the ideas presented in [GDAK02a], [Gou03] for a cluster of single-processing nodes, is described.

In Chapter 4, the non-overlapping and the overlapping execution policies are generalized, so as to be applied on a cluster of shared memory multiprocessors. In order to achieve this generalization, we introduce the technique of grouping, which is a kind of tiling applied onto tiles. We determine the guidelines for the selection of the grouping transformation. Then, a valid and optimal time schedule for the subsequent group space is produced. We also indicate how computation tasks should be allocated to the processors. Finally, we theoretically and experimentally validate the techniques proposed.

In Chapter 5, we assume that a cluster with a fixed number of SMP nodes is available for the execution of the tiled iteration space. Thus, our scheduling needs to be adapted, so as to take into consideration that a fixed number of tiles can be computed at the same time. Five alternative schedules are proposed: cyclic assignment schedule (§5.2), mirror assignment schedule (§5.3), cluster assignment (§5.4), retiling (§5.5) and block-cyclic assignment schedule (§5.7). Then, we theoretically and experimentally argue about which one should be selected for the parallelization of a tile space.

In Chapter 6, we conclude with a summary of the arguments presented in this thesis and we report some future extensions of our work. In Appendix A a summary table of the symbols used throughout the thesis is provided. Appendix B constitutes a quick reference of our algorithmic assumptions. Finally, in Appendix C, some simple mathematical formulas, which are often used in this thesis, are proven.

1.6 Publications

INTERNATIONAL JOURNALS

- M. Athanasaki, A. Sotiropoulos, G. Tsoukalas, N. Koziris, and P. Tsanakas. Hyperplane Grouping and Pipelined Schedules: How to Execute Tiled Loops Fast on Clusters of SMPs. *The Journal of Supercomputing*, 33(3):197–226, Sep. 2005.

- G. Goumas, M. Athanasaki, and N. Koziris. An Efficient Code Generation Technique for Tiled Iteration Spaces. *IEEE Trans. on Parallel and Distributed Systems*, 14(10):1021–1034, Oct. 2003.

- G. Goumas, M. Athanasaki, and N. Koziris. Code Generation Methods for Tiling Transformations. *Journal of Information Science and Engineering*, 18(5):667–691, Sep. 2002.

INTERNATIONAL CONFERENCES

- G. Goumas, N. Drosinos, M. Athanasaki, and N. Koziris. Automatic Parallel Code Generation for Tiled Nested Loops. In *Proceedings of the 2004 ACM Symposium on Applied Computing (SAC 2004)*, pages 1412–1419, Nicosia, Cyprus, March 2004.

- M. Athanasaki, E. Koukis, and N. Koziris. Scheduling of Tiled Nested Loops onto a Cluster with a Fixed Number of SMP Nodes. In *Proceedings of the 12-th Euromicro Conference on Parallel, Distributed and Network based Processing (PDP04)*, pages 424–433, A Coruna, Spain, Feb. 2004. IEEE Computer Society Press.

- M. Athanasaki, E. Koukis, and N. Koziris. Efficient Scheduling of Tiled Iteration Spaces onto a Fixed Size Parallel Architecture. In *Proceedings of the 9th Panhellenic Conference in Informatics*, pages 178–192, Thessaloniki, Greece, Nov. 2003.

- N. Drosinos, G. Goumas, M. Athanasaki, and N. Koziris. Delivering High Performance to Parallel Applications Using Advanced Scheduling. In *Proceedings of the Parallel Computing 2003 (ParCo 2003)*, Dresden, Germany, Sep. 2003.

- M. Athanasaki, A. Sotiropoulos, G. Tsoukalas, and N. Koziris. Pipelined Scheduling of Tiled Nested Loops onto Clusters of SMPs using Memory Mapped Network Interfaces. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing (SC2002)*, Baltimore, Maryland, Nov. 2002. IEEE Computer Society Press.

- G. Goumas, N. Drosinos, M. Athanasaki, and N. Koziris. Compiling Tiled Iteration Spaces for Clusters. In *Proceedings of the 2002 IEEE Int'l Conference on Cluster Computing*, pages 360–369, Chicago, Illinois, Sep. 2002.

- M. Athanasaki, A. Sotiropoulos, G. Tsoukalas, and N. Koziris. A Pipelined Execution of Tiled Nested Loops on SMPs with Computation and Communication Overlapping. In *Proceedings of the Workshop on Compile/Runtime Techniques for Parallel Computing, in conjunction with 2002 Int'l Conference on Parallel Processing (ICPP-2002)*, pages 559–567, Vancouver, Canada, Aug. 2002.

- G. Goumas, N. Drosinos, M. Athanasaki, and N. Koziris. Data Parallel Code Generation for Arbitrarily Tiled Nested Loops. In *Proceedings of the 2002 Int'l Conference on Parallel and Distributed Processing Techniques and Applications*, pages 610–616, Las Vegas, Nevada, USA, June 2002.

- G. Goumas, M. Athanasaki, and N. Koziris. Automatic Code Generation for Executing Tiled Nested Loops Onto Parallel Architectures. In *Proceedings of the 2002 ACM Symposium on Applied Computing (SAC 2002)*, pages 876–881, Madrid, Spain, March 2002.

2

Preliminary Concepts - Mathematical Background

In this chapter, we present some basic preliminary concepts and the mathematical background, which are necessary for the comprehension of the rest of this thesis. First of all, we supply an outline of the algorithmic model aimed by the techniques presented in this thesis. This model is further specified and restricted later on in this chapter. A summary of the restrictions imposed is also given in Appendix B. While going through this thesis, readers may use Appendix B as a quick reference of our algorithmic model. In addition, some terms originating from the scientific area of algebra (e.g. lexicographic order) are briefly defined in this chapter. Moreover, we discuss some concepts widely used in the area of parallel processing (e.g. dependence analysis, time scheduling, linear loop transformations, tiling). Finally, we outline the architectural characteristics, which are necessary for the implementation of the techniques described in this thesis.

2.1 Notation

Throughout this thesis, we indicate the set of natural numbers by N, and the set of natural numbers, excluding zero by N^* ($N^* = N - \{0\}$). In addition, we indicate the set of integer numbers by Z, and the set of integer numbers, excluding zero by Z^* ($Z^* = Z - \{0\}$).

In addition, when writing $\vec{a} > 0$ (or $\vec{a} \geq 0$), we mean that all coordinates of vector $\vec{a}$ should be positive (or non negative). Similarly, when writing $A > 0$ (or $A \geq 0$), where A is a matrix, we mean that all elements of A should be positive (or non negative).

By $\lfloor \vec{a} \rfloor$, we imply the application of the floor integer function to all coordinates of $\vec{a}$. Similarly, by $\lfloor A \rfloor$, we imply the application of the floor integer function to all elements of matrix A.

2.2 Algorithmic Model - Nested `for`-loops

The methods proposed in this thesis may be applied to any code segment of perfectly nested `for`-loops with uniform data dependences (see §2.3) [SF91]. That is, our algorithms are of the form:

```
for (j1=l1; j1 ≤ u1; j1++){
   ...
   for (jn=ln; jn ≤ un; jn++){
      Loop Body
   }
   ...
}
```

where l_1 and u_1 are integer parameters, l_k and u_k ($k = 2, \ldots, n$) are functions of the outer loop indices. Specifically, they may have the form:

$$l_k = max(\lceil f_{k1}(j_1, \ldots, j_{k-1}) \rceil, \ldots, \lceil f_{kr}(j_1, \ldots, j_{k-1}) \rceil)$$

and

$$u_k = min(\lfloor g_{k1}(j_1, \ldots, j_{k-1}) \rfloor, \ldots, \lfloor g_{kr}(j_1, \ldots, j_{k-1}) \rfloor)$$

where f_{ki} and g_{ki} are affine functions. Therefore, we are not only dealing with rectangular iteration spaces, but also with more general convex spaces, with the only assumption that the iteration space is defined as the bisection of a finite number of semi-spaces of the n-dimensional space Z^n.

Each iteration of this code segment is represented by an n-dimensional vector

$$\vec{j} = (j_1, j_2, \ldots, j_n) \in Z^n,$$

called as **iteration vector**. Each coordinate of the iteration vector represents one of the loop indices. Coordinate j_1 represents the outermost loop index, while j_n represents the innermost one.

Definition 2.1 *We define as* ***iteration space*** *the set of iteration vectors (representing iterations), which are to be traversed during the execution of a nested for-loop code segment, as described in page 12.*

$$J^n = \{\vec{j} = (j_1, j_2, \ldots, j_n) | j_i \in Z \land l_i \leq j_i \leq u_i, 1 \leq i \leq n\}$$

The iteration space J^n can also be described with a system of linear inequalities. An inequality of this system expresses a boundary surface of the iteration space. Thus, J^n can be equivalently defined as:

$$J^n = \{\vec{j} \in Z^n | B\vec{j} \leq \vec{b}\} \tag{2.1}$$

Matrix B and vector $\vec{b}$ can be easily derived from the affine functions l_k and u_k and vice versa.

Each iteration $\vec{j} = (j_1, j_2, \ldots, j_n) \in Z^n$ may be represented in the n-dimensional space by point $(j_1, j_2, \ldots, j_n)$. In consequence, the iteration space may be represented as a subset of Z^n, as indicated in the following example.

Example 2.1: The following nested `for`-loops are consistent to the algorithmic model described in this section.

1. **Rectangular iteration space:**

```
for (j1=0; j1 ≤ 7; j1++)
   for (j2=0; j2 ≤ 5; j2++){
      Loop Body
   }
```

   Matrices B and $\vec{b}$, corresponding to this loop segment, can be derived as follows:

$$\left.\begin{array}{l} j_1 \leq 7 \\ j_1 \geq 0 \\ j_2 \leq 5 \\ j_2 \geq 0 \end{array}\right\} \Leftrightarrow \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{bmatrix} \vec{j} \leq \begin{pmatrix} 7 \\ 0 \\ 5 \\ 0 \end{pmatrix}$$

2. **Trapezoidal iteration space:**

```
for (j1=0; j1 ≤ 7; j1++)
   for (j2=0; j2 ≤ 9 − j1; j2++){
      Loop Body
   }
```

Matrices B and $\vec{b}$, corresponding to this loop segment, can be derived as follows:

$$\left.\begin{array}{c} j_1 \leq 7 \\ j_1 \geq 0 \\ j_2 \leq 9 - j_1 \\ j_2 \geq 0 \end{array}\right\} \Leftrightarrow \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 1 & 1 \\ 0 & -1 \end{bmatrix} \vec{j} \leq \begin{pmatrix} 7 \\ 0 \\ 9 \\ 0 \end{pmatrix}$$

3. **Convex space:**

```
for (j1=0; j1 ≤ 7; j1++)
   for (j2=max(0, 1 − j1); j2 ≤ min(6, 9 − j1); j2++){
      Loop Body
   }
```

Matrices B and $\vec{b}$, corresponding to this loop segment, can be derived as follows:

$$\left.\begin{array}{c} j_1 \leq 7 \\ j_1 \geq 0 \\ j_2 \leq 6 \\ j_2 \leq 9 - j_1 \\ j_2 \geq 0 \\ j_2 \geq 1 - j_1 \end{array}\right\} \Leftrightarrow \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 1 & 1 \\ 0 & -1 \\ -1 & -1 \end{bmatrix} \vec{j} \leq \begin{pmatrix} 7 \\ 0 \\ 6 \\ 9 \\ 0 \\ -1 \end{pmatrix}$$

The respective iteration spaces can be represented in a 2-dimensional space, as depicted in Figure 2.1.

According to the constraints concerning the form of loop bounds l_i, u_i, iteration space J^n may be a convex subset of Z^n. This model is compatible with several real applications, mainly from the scientific areas of maths, physics, molecular biology, e.t.c. For example, we may refer to some of them: Jacobi, Gauss Successive Over-Relaxation - SOR, Alternative Direction Implicit Integration - ADI [GDAK02a], Texture Smoothing - TS [PB99], 9-point Star Differential Equation Stencil - PDE [AI91], Global Sequence Alignment - Fickett's Algorithm [ABRY03].

Unless a loop transformation is applied, the iterations of a nested-loop code segment are executed sequentially, in lexicographic order.

Definition 2.2 *Iteration $\vec{j}$ is lexicographically previous than iteration $\vec{j'}$ ($\vec{j} \prec \vec{j'}$), iff $j_i = j'_i, \forall i = 1, \ldots, k-1 \wedge j_k < j'_k$, $k \leq n$.*

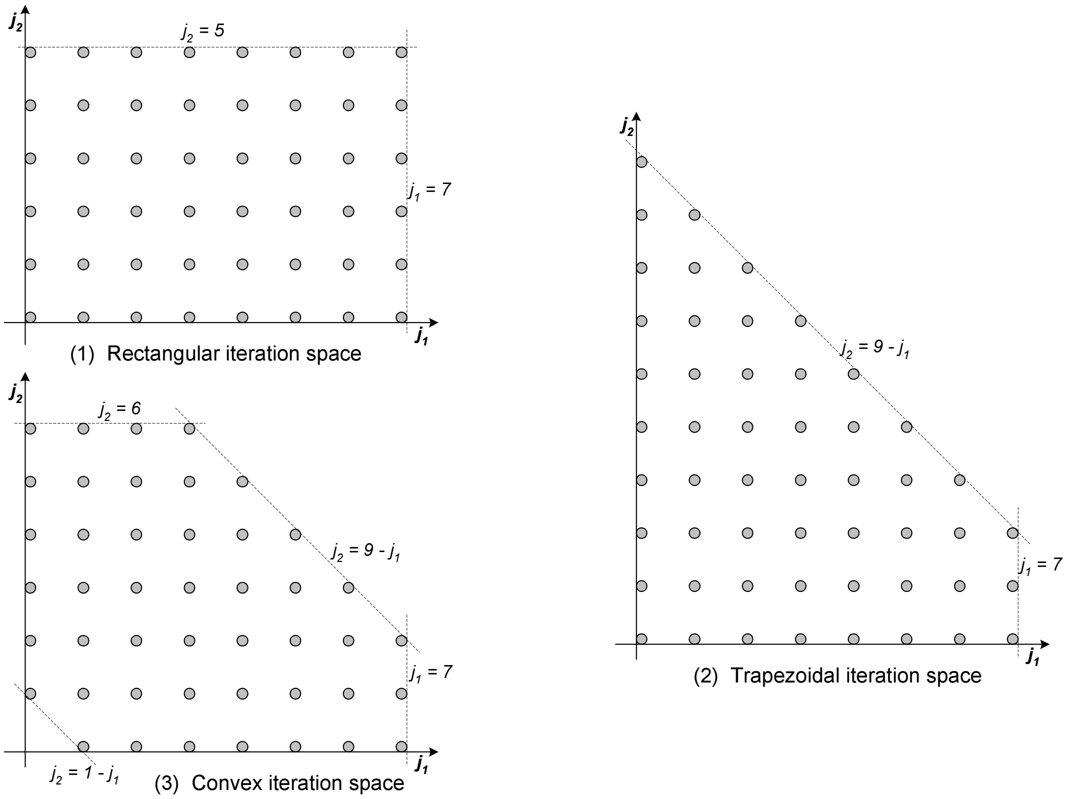


Figure 2.1: Example 2.1 - Graphical representation of 2-dimensional iteration spaces onto Z^n

For example, it holds that $(1, 2, 5) \prec (4, 1, 0) \prec (4, 1, 1) \prec (4, 3, -8)$. In Figure 2.2, we have depicted the lexicographic order, which is coincident to the program order, for the iterations of the code segment in Example 2.1(3).

2.3 Dependence Vectors

Definition 2.3 *Iteration $\vec{j_2}$ is* ***dependent on*** *iteration $\vec{j_1}$ iff*

1. *All three conditions are valid:*

   (a) $\vec{j_1} \prec \vec{j_2}$ and

   (b) Both iterations $\vec{j_1}, \vec{j_2}$ access the same memory data item M and

   (c) At least one of these memory data accesses is a write access,

   or,

2. *Iteration $\vec{j_2}$ is dependent on iteration $\vec{j_3}$ and iteration $\vec{j_3}$ is dependent on iteration $\vec{j_1}$.*

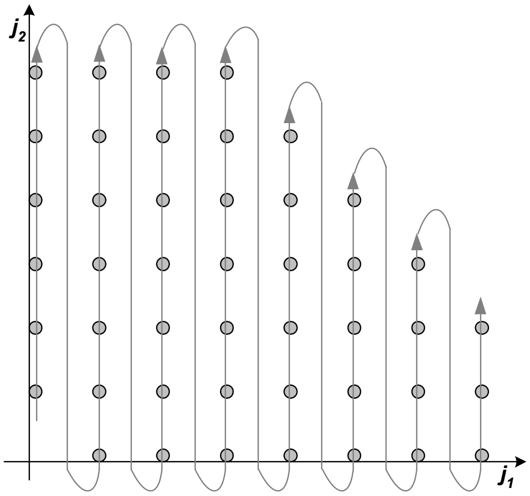


Figure 2.2: Lexicographic order of iterations for the iteration space of Example 2.1(3). It is coincident to the order of execution of the iterations if no transformation is applied to the iteration space.

In the first case, $\vec{j_2}$ is ***directly dependent on*** *$\vec{j_1}$, while in the second one, $\vec{j_2}$ is* ***indirectly dependent on*** *$\vec{j_1}$.*

When $\vec{j_2}$ is dependent on $\vec{j_1}$, we equivalently say that there is a **dependence** between iterations $\vec{j_1}$ and $\vec{j_2}$. Formally, dependences are modelled by **dependence vectors**: $\vec{d} = \vec{j_2} - \vec{j_1}$.

Dependence analysis is especially critical for the parallelization of programs, since any two iterations can be executed in parallel, if there is no direct or indirect dependence between them [Ber66], [Ban94]. However, when modelling dependences using dependence vectors, we only deal with direct dependences. Indirect dependences are implied.

Direct dependences are distinguished into three categories [Ban88]:

- **flow** or **true dependences**, if iteration $\vec{j_1}$ writes on M and dependent iteration $\vec{j_2}$ reads the value of M.
- **anti-dependences**, if iteration $\vec{j_1}$ reads the value of M and then dependent iteration $\vec{j_2}$ writes on M.
- **output dependences**, if both iterations $\vec{j_1}$ and $\vec{j_2}$ write on M.

In our algorithmic model, we only deal with flow or true dependences. Anti-dependences and output dependences can be eliminated using more variables [CDRV98]. In addition, notice that, in our algorithmic model (§2.2), all dependence vectors are considered as uniform, i.e. independent of the indices of computations. Thus, we may construct the **dependence matrix** D of a code segment, which consists of all dependence vectors starting from any iteration of J^n. Each dependence vector forms a column of matrix D: $D = [d_1|d_2|...|d_q]$.

Example 2.2: Let us consider the nested `for`-loop code segment:

```
for (j1=0; j1 ≤ 7; j1++)
   for (j2=max(0,1-j1); j2 ≤ min(6,9-j1); j2++){
      A[j1,j2] = B[j1+4,j2]+A[j1-2,j2]
      B[j1,j2] = A[j1-3,j2+1] -A[j1,j2-1]
   }
```

Iteration (j_1, j_2) reads matrix elements $A[j_1 - 2, j_2]$, $A[j_1 - 3, j_2 + 1]$, $A[j_1, j_2 - 1]$, which are written by iterations $(j_1 - 2, j_2)$, $(j_1 - 3, j_2 + 1)$, $(j_1, j_2 - 1)$, respectively. Thus, there are true or flow dependences: $\vec{d_1} = (2, 0)$, $\vec{d_2} = (3, -1)$, $\vec{d_3} = (0, 1)$. In addition, iteration (j_1, j_2) reads matrix element $B[j_1 + 4, j_2]$, which is later written by iteration $(j_1 + 4, j_2)$, imposing anti-dependence $\vec{d_4} = (4, 0)$. Therefore, the dependence matrix of this code segment is: $D = \left[\begin{array}{c|c|c|c} 2 & 3 & 0 & 4 \\ 0 & -1 & 1 & 0 \end{array}\right]$. Notice that all four dependence vectors are lexicographically positive.

In order to eliminate anti-dependence $\vec{d_4} = (4, 0)$, we may equivalently rewrite the previous code segment as follows:

```
for (j1=0; j1 ≤ 7; j1++)
   for (j2=max(0,1-j1); j2 ≤ min(6,9-j1); j2++)
      B_temp[j1+4,j2] = B[j1+4,j2]
for (j1=0; j1 ≤ 7; j1++)
   for (j2=max(0,1-j1); j2 ≤ min(6,9-j1); j2++){
      A[j1,j2] = B_temp[j1+4,j2]+A[j1-2,j2]
      B[j1,j2] = A[j1-3,j2+1] -A[j1,j2-1]
   }
```

The dependence matrix for the second nested `for`-loop of this code segment is: $D = \left[\begin{array}{c|c|c} 2 & 3 & 0 \\ 0 & -1 & 1 \end{array}\right]$. These dependences can be graphically represented, as depicted in Figure 2.3.

2.4 Fourier-Motzkin Elimination Method

The Fourier-Motzkin elimination method (FME) can be used to convert a system of linear inequalities $A\vec{x} \leq \vec{a}$ into a form, in which the lower and upper bounds of each element x_i of the vector $\vec{x}$ is expressed in terms of the elements $x_1, \ldots, x_{i-1}$ only. This fact is very important when using a nested loop, in order to traverse an iteration space J^n defined by a system of inequalities. In this case, the bounds of index j_k of the nested loop must be expressed in terms of the $k - 1$ outer indices only. This means that the Fourier-Motzkin elimination method can convert a system describing a general iteration space into a form suitable for use in nested loops.

After applying the Fourier-Motzkin elimination method, the eliminated system consists of a

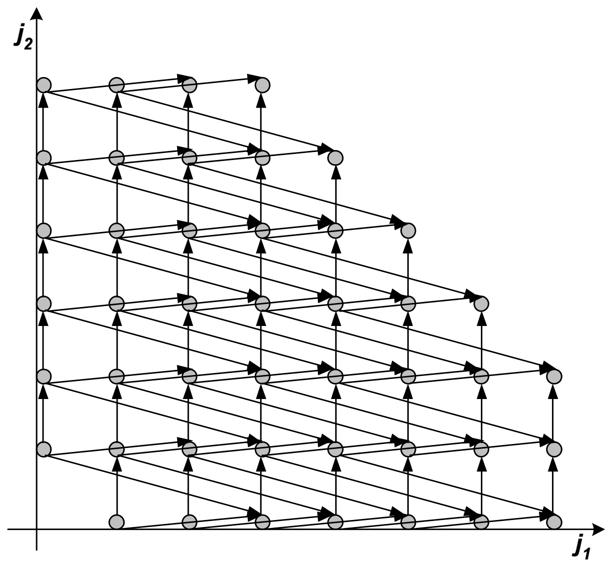


Figure 2.3: Example 2.2 - Graphical representation of flow dependences

very large number of inequalities describing the bounds of each variable x_i, but some of them are not necessary for the calculation of x_i's bounds. The unnecessary inequalities must be eliminated to simplify the resulting system. In order to remove the redundant inequalities, two methods have been proposed: the **ad-Hoc simplification method** and the **exact simplification method**. A full description of the Fourier-Motzkin elimination method, the ad-Hoc simplification and the exact simplification is presented in [BW95].

If the initial system of inequalities consists of k inequalities with n variables, then the complexity of the Fourier-Motzkin elimination algorithm can be expressed by the formula ([Jim99]):

$$Complexity = O(\frac{k^{2^n}}{2^{2^{(n+1)}-2}}) \approx O((\frac{k}{2})^{2^n})$$

The Fourier-Motzkin elimination method is extremely complex, since it depends *doubly exponentially* on the number of loops involved.

In addition, a single application of the method is almost always useless, since it results to a lot of inequalities, which are not necessary for the calculation of the loop bounds. They should be calculated a lot of times during the execution of the final code and impose an unacceptable overhead to the final code execution. Thus, the above simplification methods should be applied, in order to eliminate the redundant inequalities. The ad-Hoc simplification method, which is quite fast, achieves to eliminate only some of the redundant inequalities. The rest of them should be eliminated with the use of the exact simplification method. It applies once the Fourier-Motzkin elimination method for each inequality of the final system, in order to check whether it is redundant. Thus, it increases considerably the complexity of the final program.

2.5 Time Scheduling

When parallelizing a nested `for`-loop, one should primarily reorganize the sequential execution of iterations, in order to create parallel regions, which may be executed at the same time by different CPUs. The final goal is the minimization of the total execution time. This is the case when no other applications are running simultaneously on the same computing system and thus we are not interested in the interaction among different applications.

The functions which map the iterations of a nested `for`-loop onto different time instances, are called **time scheduling functions**. When devising a time scheduling function, our goal is to enable the execution of as many parallel iterations as possible, so as to achieve the minimum total execution time, without modifying the results produced by the initial sequential execution of the program.

In order to certify that the results produced by the initial sequential execution are not modified, a time schedule must respect the initial program dependences. In other words, it should map iterations connected by a dependence vector to distinct execution steps. In this way, it is ensured that only those iterations of the initial nested `for`-loop that have no direct or indirect dependence among them will be executed in parallel. Thus, a time schedule is valid when for each dependence vector, the source iteration is mapped to a time instance previous than the destination iteration.

Definition 2.4 *Time scheduling function $s : J^n \rightarrow Z$ is valid for a nested for-loop, with a dependence matrix D, iff for each pair of iterations $\vec{j_1}, \vec{j_2} \in J^n : \vec{j_2} = \vec{j_1} + \vec{d}, \vec{d} \in D$, it holds that $s(\vec{j_1}) < s(\vec{j_2})$.*

2.5.1 Linear Time Scheduling

Linear time scheduling is a special case of time scheduling. It arises when the scheduling function $s(\vec{j})$ is linear. Linearity is convenient, as we shall see in Chapters 4 and 5, since it results in a regular assignment of iterations or tiles (see §2.6.2 for a definition of tile) to CPUs.

Definition 2.5 *We define as linear time scheduling of a nested for-loop, any time scheduling s_Π, such that: $\forall \vec{j} \in J^n$*

$$s_\Pi(\vec{j}) = \lfloor \frac{\Pi \vec{j}^T + t_0}{disp\Pi} \rfloor$$

where $\Pi \in Z^{1 \times n}$, $disp\Pi = min\{\Pi \vec{d_i}^T : \vec{d_i} \in D\}$ and t_0 is an integer constant.

We notice that in Definition 2.5:

- Row-vector Π is called as **linear scheduling vector**.

- Integer constant t_0 is called as **alignment constant**.

- Constant $disp\Pi$ is called as **displacement constant**.

Linear scheduling vector Π defines a class of hyperplanes such that: All iterations of J^n belonging to the same hyperplane are mapped to the same time instance. When using the term *hyperplane*, we mean a beeline for a 2-dimensional iteration space, a ruled surface for a 3-dimensional iteration space and so on.

It can be proven [PTK98] that a linear time scheduling preserves depedences iff

$$\forall \vec{d_i} \in D : \Pi \vec{d_i}^T > 0 \tag{2.2}$$

According to a linear time scheduling s_Π, the time required for the execution of a nested `for`-loop (makespan) is calculated with the use of formula:

$$\wp = max\{s_\Pi(\vec{j}) : \vec{j} \in J^n\} - min\{s_\Pi(\vec{j}) : \vec{j} \in J^n\} + 1 \tag{2.3}$$

Example 2.3: In this example, we will produce a parallel time schedule for the iterations of the nested `for`-loop code segment:

```
for (j1=0; j1 ≤ 7; j1++)
   for (j2=max(0,1-j1); j2 ≤ min(6,9-j1); j2++){
      A[j1,j2] = B_temp[j1+4,j2]+A[j1-2,j2]
      B[j1,j2] = A[j1-3,j2+1] -A[j1,j2-1]
   }
```

The dependences of this nested `for`-loop have been designed in Figure 2.3. Let us select vector $\Pi = \begin{bmatrix} 1 & 1 \end{bmatrix}$, as a linear scheduling vector for this iteration space.

$$\Pi \vec{d_1}^T = \begin{bmatrix} 1 & 1 \end{bmatrix} \begin{pmatrix} 2 \\ 0 \end{pmatrix} = 2 > 0,$$

$$\Pi \vec{d_2}^T = \begin{bmatrix} 1 & 1 \end{bmatrix} \begin{pmatrix} 3 \\ -1 \end{pmatrix} = 2 > 0,$$

$$\Pi \vec{d_3}^T = \begin{bmatrix} 1 & 1 \end{bmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = 1 > 0,$$

According to formula (2.2), Π is a valid scheduling vector for this example. In addition, according to Definition 2.5, $disp\Pi = min\{\Pi\vec{d_i}^T : \vec{d_i} \in D\} = 1$. If we set $t_0 = -1$, then we get:

$$s_\Pi(j_1, j_2) = j_1 + j_2 - 1$$

In Figure 2.4 we have depicted the resulting time schedule. Notice that, according to formula (2.3), the makespan is $\wp = 9$.

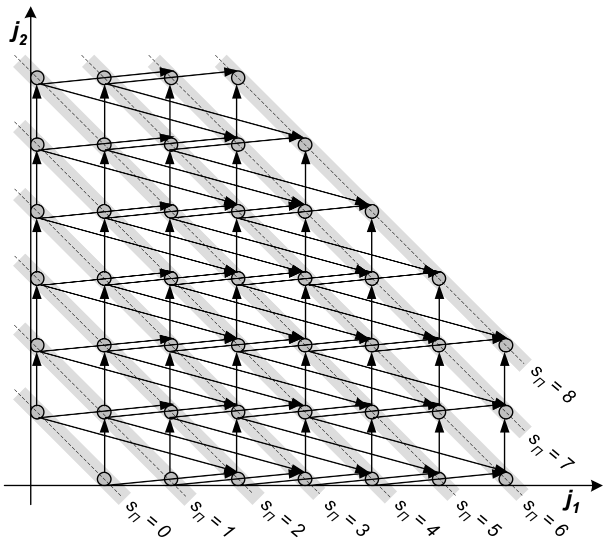


Figure 2.4: Example 2.3 - Time Schedule produced by linear scheduling vector $\Pi = [1\ 1]$. The dashed lines indicate the class of hyperplanes-beelines defined by the linear scheduling vector Π ($\Pi\vec{j} = constant$). The grey areas include iterations that are mapped to the same time instance, according to the scheduling function $s_\Pi(j_1, j_2) = j_1 + j_2 - 1$. Since $disp\Pi = 1$, each grey area includes only one hyperplane.

If we select vector $\Pi = \begin{bmatrix} 2 & 3 \end{bmatrix}$, as a linear scheduling vector:

$$\Pi\vec{d_1}^T = \begin{bmatrix} 2 & 3 \end{bmatrix} \begin{pmatrix} 2 \\ 0 \end{pmatrix} = 4 > 0,$$

$$\Pi\vec{d_2}^T = \begin{bmatrix} 2 & 3 \end{bmatrix} \begin{pmatrix} 3 \\ -1 \end{pmatrix} = 3 > 0,$$

$$\Pi\vec{d_3}^T = \begin{bmatrix} 2 & 3 \end{bmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = 3 > 0,$$

According to formula (2.2), Π is a valid scheduling vector for this example. In addition, according to Definition 2.5, $disp\Pi = min\{\Pi\vec{d_i}^T : \vec{d_i} \in D\} = 3$. If we set $t_0 = -2$, then we get:

$$s_\Pi(j_1, j_2) = \lfloor \frac{2j_1 + 3j_2 - 2}{3} \rfloor$$

In Figure 2.5 we have depicted the resulting time schedule. Notice that, according to formula (2.3), the makespan is $\wp = 8$.

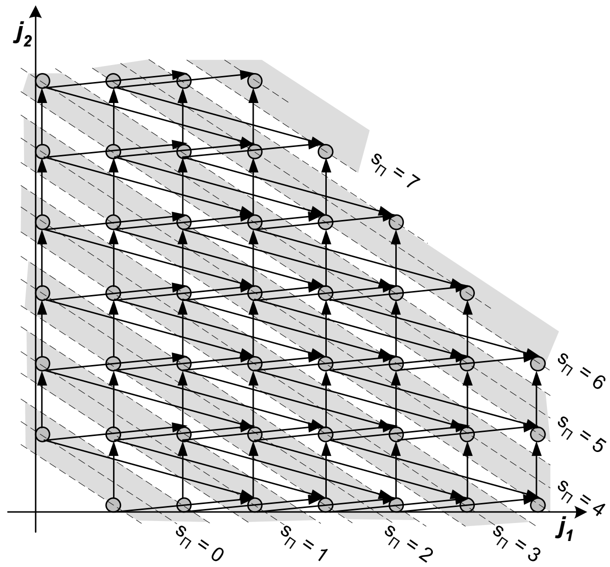


Figure 2.5: Example 2.3 - Time Schedule produced by linear scheduling vector $\Pi = [2\ 3]$. The dashed lines indicate the class of hyperplanes-beelines defined by the linear scheduling vector Π ($\Pi\vec{j} = constant$). The grey areas include iterations that are mapped to the same time instance, according to the scheduling function $s_\Pi(j_1, j_2) = \lfloor\frac{2j_1+3j_2-2}{3}\rfloor$. Since $disp\Pi = 3$, each grey area includes 3 hyperplanes.

2.6 Loop Transformations

2.6.1 Linear Loop Transformations

Linear transformations, which are often used in loop transformation literature can be distinguished into three main categories:

1. loop interchange

2. loop reversal

3. loop skewing

Each linear loop transformation can be represented by a $n \times n$ transformation matrix T. Thus, iteration $\vec{j}$ of the initial iteration space is mapped to iteration $T\vec{j}$ of the final iteration space and dependence vector $\vec{d_i}$ is transformed to dependence vector $T\vec{d_i}$. A loop transformation results in a code segment equivalent to the original one iff it preserves dependences, that is iff all transformed dependence vectors are lexicographically positive ($\forall \vec{d_i} \in D$ it holds $T\vec{d_i} \succ \vec{0}$) [WL91b].

If more than one linear transformations T_1, T_2 are successively performed, the final loop transformation can be represented by the product of the respective transformation matrices $T = T_2 T_1$.

Loop interchange transforms iteration vector (j_1, j_2) into iteration vector (j_2, j_1) (see Figure 2.6). This transformation can be represented by matrix $T = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$. Thus

$$\vec{j}' = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{pmatrix} j_1 \\ j_2 \end{pmatrix} = \begin{pmatrix} j_2 \\ j_1 \end{pmatrix}$$

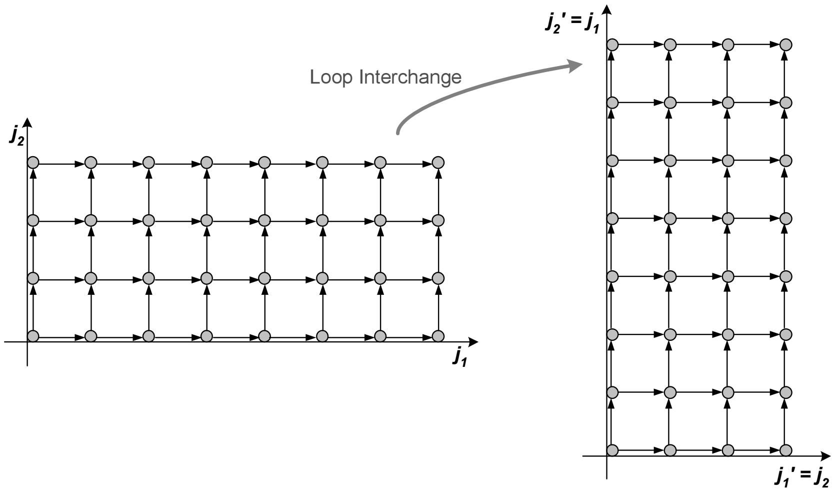


Figure 2.6: Graphical representation of an interchange transformation

Two successive loop interchanges can model a cyclic exchange of three loop indices, so as the innermost loop index j_3 to become the outermost one. First, interchange of loop indices j_2, j_3 is represented by matrix $T_1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$. Second, interchange of loop indices j_1, j_2 is represented by matrix $T_2 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$. The total transformation is represented by matrix $T = T_2T_1 = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$.

Loop reversal is modelled by multiplying a loop index by -1. For example, the reversal transformation depicted in Figure 2.7 is modelled by transformation matrix $T = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$.

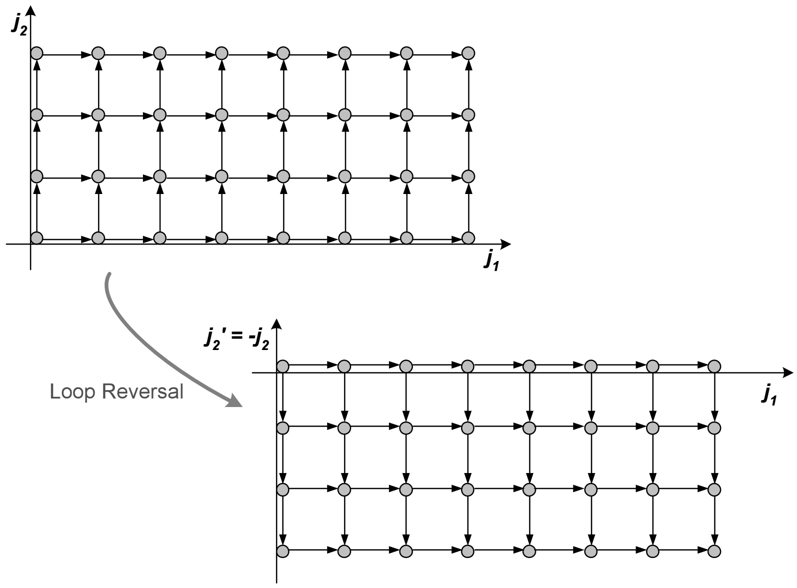


Figure 2.7: Graphical representation of a reversal transformation

Loop skewing adds a loop index multiple to another loop index. For a 2-dimensional iteration space, it can be modelled by a transformation matrix $T = \begin{bmatrix} 1 & 0 \\ f & 1 \end{bmatrix}$ or $T = \begin{bmatrix} 1 & f \\ 0 & 1 \end{bmatrix}$, where $f \in Z$. For example, the transformation shown in Figure 2.8 is represented by matrix $T = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$.

All above loop transformations are **unimodular transformations** and are represented by **unimodular matrices**.

Definition 2.6 *A square matrix A is unimodular, if it consists of only integer elements and its determinant equals to ± 1.*

Unimodular transformations have a very useful property: their inverse transformation is integral as well. On the other hand the inverse of a non-unimodular matrix is not integral, which causes the transformed space to have *holes*. We call *holes* the integer points of the transformed space that have no integer anti-image in the original space.

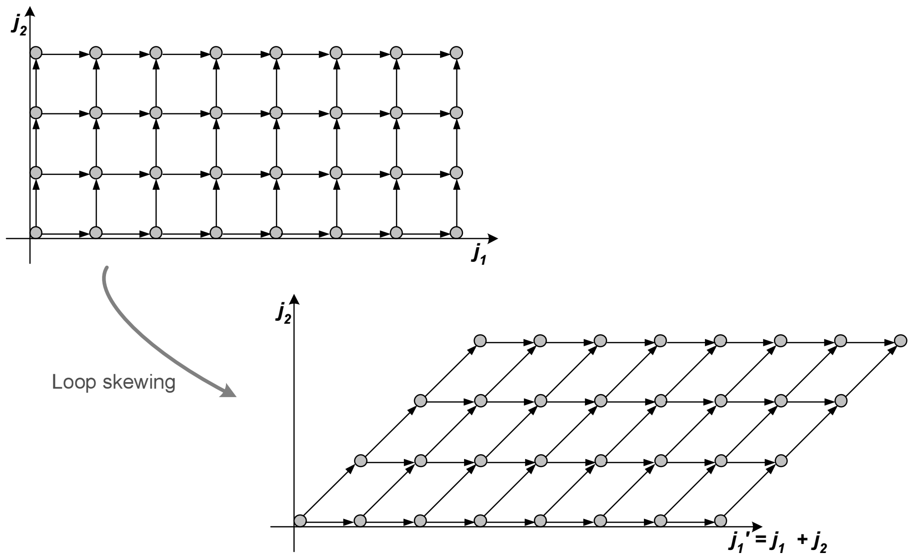


Figure 2.8: Graphical representation of a skewing transformation

Definition 2.7 *Let A be an $m \times n$ integer matrix. We call the set $\mathcal{L}(A) = \{\vec{y} | \vec{y} = A\vec{x} \wedge \vec{x} \in Z^n\}$ the lattice that is generated by the columns of A.*

Consequently, we can define the holes of a non-unimodular transformation as follows: if T is a non-unimodular transformation, we call *holes* the points $\vec{j'} \in Z^n$, such that $T^{-1}\vec{j'} \notin Z^n$. On the contrary, we call *actual* points of a non-unimodular transformation T the points $\vec{j'} \in Z^n$, for which it holds $T^{-1}\vec{j'} \in Z^n \Leftrightarrow \vec{j'} \in \mathcal{L}(T)$. Figure 2.9 shows the image of an iteration space after the application of a unimodular and a non-unimodular transformation. Holes are depicted with white dots and actual points with grey ones. It has been proven in [Ram92] that if T is a $m \times n$ integer matrix, and C is an $n \times n$ unimodular matrix, then $\mathcal{L}(T) = \mathcal{L}(TC)$.

Definition 2.8 *We say that a square, non-singular matrix $H = [\vec{h_1}, \dots, \vec{h_n}] \in R^{n \times n}$ is in column hermite normal form (HNF) iff H is lower triangular ($h_{ij} \neq 0$ implies $i \geq j$) and for all $i > j$, $0 \leq h_{ij} < h_{ii}$ (the diagonal is the greatest element in the row and all entries are positive.)*

As proven in [Ram92], if T is a $m \times n$ integer matrix of full row rank, then there exists an $n \times n$ unimodular matrix C such that $TC = [\widetilde{T}0]$ and $\widetilde{T}$ is in hermite normal form. Every integer matrix with full row rank has a unique hermite normal form. It holds that $\mathcal{L}(T) = \mathcal{L}(\widetilde{T})$, which means that an integer matrix of full row rank and its hermite normal form produce the same lattice. This property is very useful for code generation of tiled spaces, as we shall see in Chapter 3.

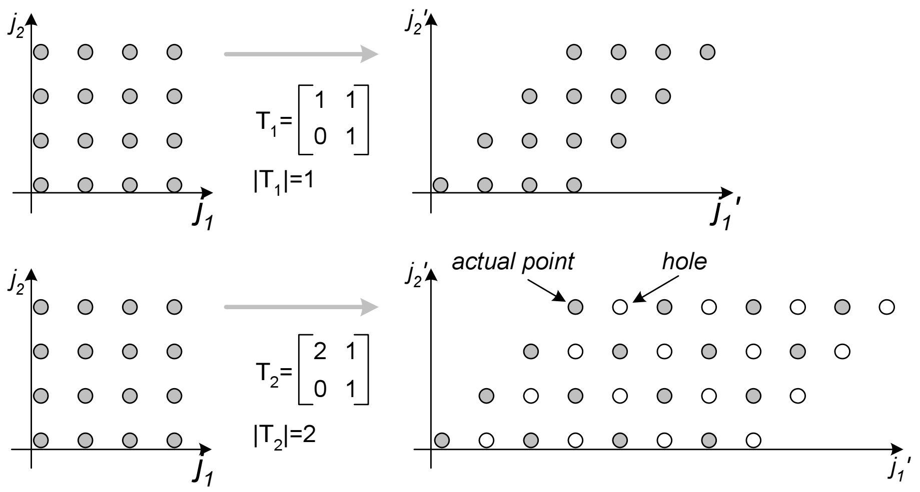


Figure 2.9: Unimodular and non-unimodular transformations.
The main difference between unimodular and non-unimodular transformations is that: The former constitute a 1-1 function from Z^n to Z^n. The latter results to "holes" in the transformed space, which do not have an integer anti-image in the initial space, as depicted by white dots in this figure.

2.6.2 Tiling or Supernode Transformation

Fine vs. Coarse grained parallelism

When parallelizing a code segment, apart from performing a dependence analysis and determining which iterations may be executed simultaneously (as seen in §2.5), we should also determine which iterations will be executed by which processors. For example, the schedule depicted in Figure 2.4, can be implemented by assigning a row of iterations to each processor, as seen in Figure 2.10. This partitioning of the iteration space can supply an intuition of **fine grain parallelism** [PTK98]. The goal of this mapping is the parallel execution of as many iterations as possible.

In Figure 2.10, we have erased dependences among iterations assigned to the same processor. Only dependences among iterations assigned to different processors are represented by black arrows. These dependences correspond to data computed in a processor, which should be used in computations executed by another processor. Thus, they correspond to data that should be somehow transferred from a processor to another. This transfer implies a communication overhead, which may be minimal, when a systolic parallel architecture is embedded on chip [PTK98], or vast when implemented upon a message passing interface, such as MPI [MPI94], [MPI97].

The volume of data that must be transferred may be large enough to annihilate the advantages of parallelization. It is strongly possible that the parallel program will take longer to execute than the sequential one. The problem in this implementation is not only the amount of data to be transferred, but also the number of distinct messages encapsulating the data. Thus,

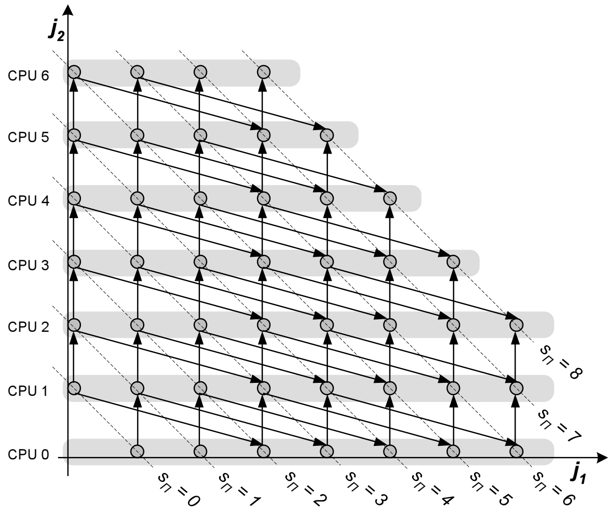


Figure 2.10: Fine-grained parallelism.
In this figure, iterations along the same dashed line are executed at the same time. Iterations inside the same grey area are executed by the same processor. Dependences among iterations assigned to the same processor have been eliminated. With black arrows, we have depicted only dependences among iterations assigned to different processors.

in order to achieve an efficient parallelization one should devise a way to

1. reduce the amount of data transferred and

2. group them into fewer messages.

Both of these objectives can be achieved by a **supernode** or **tiling transformation**, that is by grouping together a number of neighboring iterations and considering them as an atomic unit. Then, instead of scheduling iterations, we schedule tiles. Communication occurs before and after the execution of a whole tile. In other words, a processor should receive the data required for the computation of a tile, before the execution of this tile's iterations start, and send data computed inside this tile, after the execution of the entire tile has been completed. Thus, apart from reducing the amount of data to be transferred, we may also group in a single message the transmission of data computed in the same tile, as seen in Figure 2.11.

An Intuitive Definition of Tiling Transformation

In general, when applying tiling, an n-dimensional iteration space J^n is partitioned by n independent families of parallel hyperplanes into n-dimensional hyperparallelepipeds, named as **tiles**. Each tile is represented by an n-dimensional vector $\vec{j^S} = (j_1^S, j_2^S, \ldots, j_n^S) \in Z^n$, called as **tile vector** (in correspondence to iterations being represented by iteration vectors). In Figure 2.12 we have indicated the tile vector, which identifies each tile.

In addition, each tile has a unique starting iteration, called as **tile origin iteration**. Iteration $(0, \ldots, 0)$ is the origin iteration of tile $(0, \ldots, 0)$. In order to identify the origin iteration

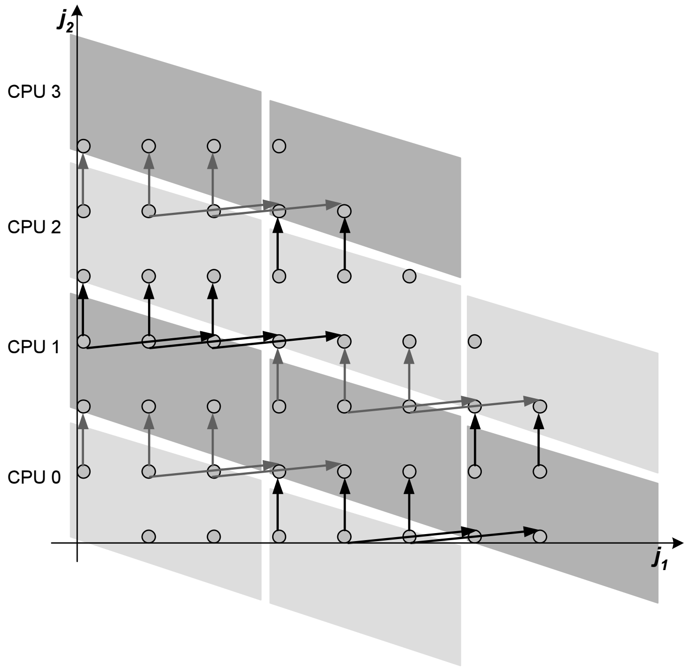


Figure 2.11: Coarse-grained parallelism.
Iterations within the same parallelogram are grouped together in the same tile. Neighboring tiles of the same shade are assigned to the same processor and executed successively. Dependences among iterations assigned to the same processor have been eliminated. In addition, dependences with origin inside the same tile have been depicted with arrows of the same shade. The respective data transfers can be grouped in a single message.

of another tile $\vec{j_x^S}$, we should parallely shift tile $(0,\ldots,0)$, so as to be congruent with tile $\vec{j_x^S}$. Then, the iteration of tile $\vec{j_x^S}$, which is congruent with iteration $(0,\ldots,0)$ is the origin iteration of tile $\vec{j_x^S}$. In Figure 2.12 we have pointed out the origin iteration of each tile. Notice that tile origin iterations may not be included in the iteration space. For example, in Figure 2.12, iteration $(0,\ldots,0)$, which is the origin iteration of tile $(0,\ldots,0)$, is not included in J^n. In order to distinguish this iteration from other tile origin iterations, we have depicted it as a white dot.

A tiling transformation can be uniquely defined by n vectors-edges of the tiles-hyperparallelepipeds. Thus, a tiling transformation can be defined by an $n \times n$ matrix P, called **inverse tiling matrix**, whose columns consist of the above mentioned vectors-edges. For example, in Figure 2.13, we have indicated how the inverse tiling matrix is derived from Figure 2.12.

Dually, a tiling transformation can be defined by an $n \times n$ matrix $H = P^{-1}$, called **tiling matrix**. Each row-vector of H is perpendicular to a class of hyperplanes partitioning the iteration space into tiles.

The tiling matrix H has some important properties concerning tiling transformation:

1. Iteration $\vec{j}$ is mapped to tile $\vec{j^S} = \lfloor H\vec{j} \rfloor$.

2. Iteration $\vec{j_0} = H^{-1}\vec{j^S}$ is the origin iteration of tile $\vec{j^S}$.

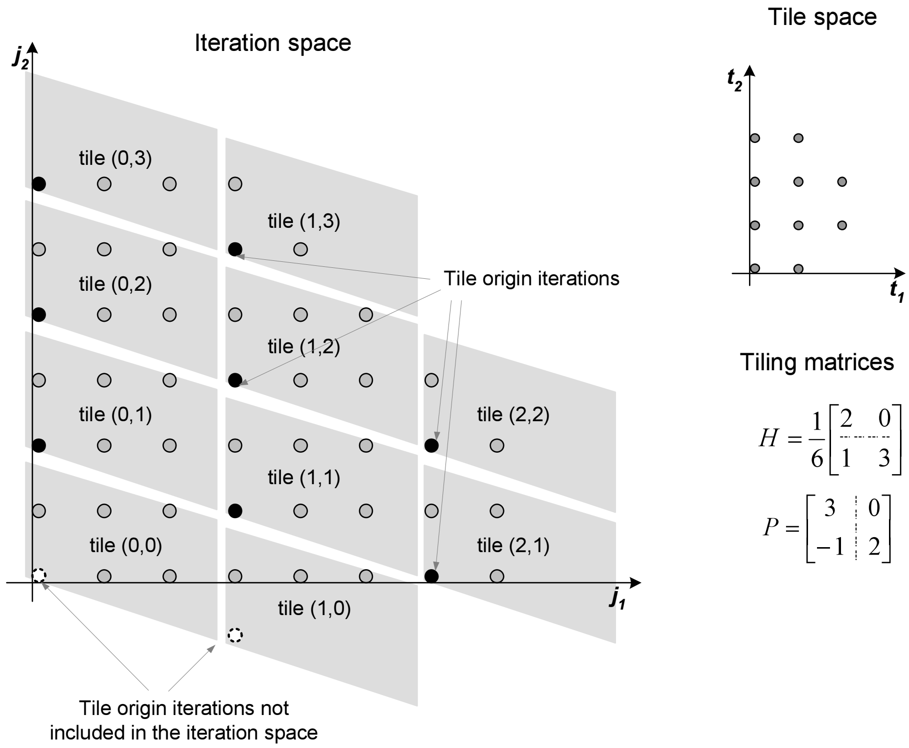


Figure 2.12: Tiling Transformation.
The iterations inside the same grey area are mapped to the same tile. Each tile is identified by a unique tile vector, which has been indicated inside the respective grey area. Black dots represent the origin iterations of each tile. Notice that tile origin iterations may not be included in the iteration space. See, for example, the tile origin iterations of tiles $(0,0)$ and $(1,0)$, which have been designed as white dots.

Notice that, as far as parallel processing is concerned, tiling transformation is useful only in case the iteration space cannot be partitioned into independent subsets. This happens when the class of dependence matrix D equals to n. Otherwise, the independent subsets may be assigned one to each processor [WL91b], [Hol92], [SF92], [PC89]. Then, there is no need for communication among processors during the execution of the iteration space (see, for example, Figure 2.14).

A Formal Definition of Tiling Transformation

Formally, tiling transformation is defined as follows:

$$r : Z^n \longrightarrow Z^{2n}, r(\vec{j}) = \begin{bmatrix} \lfloor H\vec{j} \rfloor \\ \vec{j} - H^{-1}\lfloor H\vec{j} \rfloor \end{bmatrix}$$

where vector $\lfloor H\vec{j} \rfloor$ identifies the coordinates of the tile that index point $\vec{j} = (j_1, j_2, \ldots, j_n)$ is mapped to, and $\vec{j} - H^{-1}\lfloor H\vec{j} \rfloor$ gives the coordinates of $\vec{j}$ within that tile relative to the tile

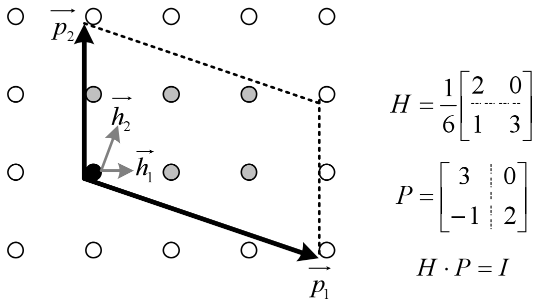


Figure 2.13: Construction of Tiling Matrices.
Matrix P consists of the edge-vectors of the tile-hyperparallelepiped. Matrix H is the inverse of matrix P.

origin. Thus, the initial n-dimensional iteration space J^n is transformed to a $2n$-dimensional one, consisting of the n-dimensional space of tiles (tile space) and the n-dimensional space of indices within tiles (tile iteration space).

- The **tile space** J^S is defined as follows:

$$J^S = \{\vec{j^S} | \vec{j^S} = \lfloor H\vec{j} \rfloor, \vec{j} \in J^n\} \tag{2.4}$$

  It can be also written as

$$J^S = \{\vec{j^S} = (j_1^S, \ldots, j_n^S) | j_i^S \in Z \wedge l_i^S \leq j_i^S \leq u_i^S, 1 \leq i \leq n\}$$

  where l_i^S, u_i^S can be directly computed from the functions $l_1, \ldots, l_n$, $u_1, \ldots, u_n$ and the tiling matrix H, as described in [AI91], [GAK03] and in Chapter 3 of this thesis. Each point $\vec{j^S}$ in this n-dimensional integer space J^S is a distinct tile with coordinates $(j_1^S, j_2^S, \ldots, j_n^S)$.

- The **tile iteration space**

$$TIS = \{\vec{j} \in Z^n | 0 \leq \lfloor H\vec{j} \rfloor < 1\} \tag{2.5}$$

  contains all points that belong to the tile starting at the axes origins.

- The **tile origin space**

$$TOS = \{\vec{j_0} \in Z^n | \vec{j_0} = H^{-1}\vec{j^S}, \vec{j^S} \in J^S\} \tag{2.6}$$

  contains the origins of tiles in the original iteration space.

Thus, it holds: $J^n \xrightarrow{H} J^S$ and $J^S \xrightarrow{P} TOS$. Note that all points of J^n that belong to the same tile, are mapped to the same point of J^S. Note also that TOS is not necessarily a subset of

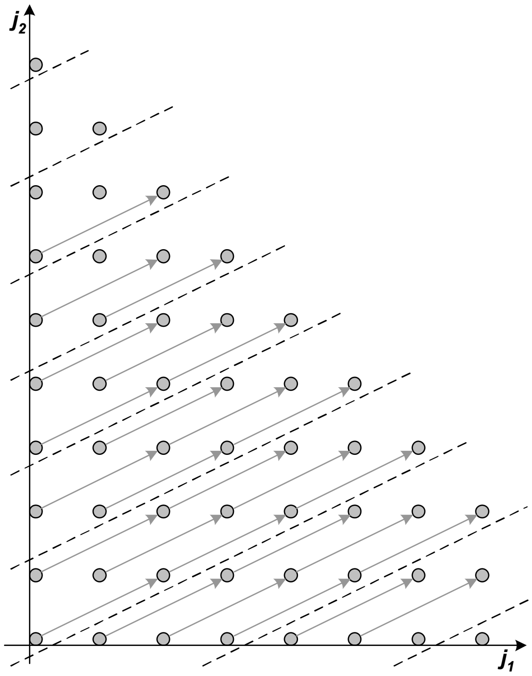


Figure 2.14: When the class of dependence matrix D is less than n we can partition the n-dimensional iteration space into independent subsets. Thus, we achieve parallelization of this iteration space with no communication at all.

J^n, since there may exist tile origins which do not belong to the original iteration space J^n, but some iterations within these tiles do belong to J^n. These tile origins are depicted in Figure 2.12 by white dots.

Points belonging to the same tile with tile origin $\vec{j_0} \in TOS$, satisfy the system of inequalities

$$0 \leq H(\vec{j} - \vec{j_0}) < 1 \tag{2.7}$$

In order to deal with integer inequalities, we define g to be the smallest natural number such that gH is an integer matrix. Thus, we can rewrite the above system of inequalities as follows: $0 \leq gH(\vec{j} - \vec{j_0}) < g \Leftrightarrow$

$$0 \leq gH(\vec{j} - \vec{j_0}) \leq (g-1) \tag{2.8}$$

We denote

$$S = \begin{pmatrix} gH \\ -gH \end{pmatrix} \text{ and } \vec{s} = \begin{pmatrix} (g-1)\vec{1} \\ \vec{0} \end{pmatrix}$$

Equivalently, system (2.8) becomes:

$$S(\vec{j} - \vec{j_0}) \leq \vec{s} \tag{2.9}$$

Note that if $\vec{j_0} = 0$, $S(\vec{j} - \vec{j_0}) \leq \vec{s}$ is satisfied iff a point belongs to TIS.

Example 2.4: If we apply the tiling transformation of Figure 2.12 to the iteration space of Example 2.3, then, as shown in Figure 2.12,

1. J^n is transformed by matrix H to the tile space

$$J^S = \{(0,0),(0,1),(0,2),(0,3),(1,0),(1,1),(1,2),(1,3),(2,1),(2,2)\}$$

2. The tile iteration space contains the points $TIS = \{(0,0),(0,1),(1,0),(1,1),(2,0),(2,1)\}$.

3. The tile space is transformed by matrix P to the tile origin space

$$TOS = \{(0,0),(0,2),(0,4),(0,6),(3,-1),(3,1),(3,3),(3,5),(6,0),(6,2)\}$$

Note that points $(0,0),(3,-1) \in TOS$ do not belong to J^n.

Since $g = 6$, the system of inequalities $S(\vec{j} - \vec{j_0}) \leq \vec{s}$ describing the boundaries of a tile is

$$\begin{pmatrix} 2 & 0 \\ 1 & 3 \\ -2 & 0 \\ -1 & -3 \end{pmatrix} \begin{pmatrix} j_1 - j_{01} \\ j_2 - j_{02} \end{pmatrix} \leq \begin{pmatrix} 5 \\ 5 \\ 0 \\ 0 \end{pmatrix}$$

2.6.3 Tile Dependences

As seen in page 27, one of the final goals of tiling is to construct a more efficient parallel execution schedule for a specific application. Instead of scheduling iterations, as in §2.5, we should now schedule tiles. Thus, instead of dependences among iterations (see Definition 2.4), we should take into consideration the dependences among tiles.

Dependences among tiles are given by the column-vectors of the **tile dependence matrix** D^S, which is defined as follows:

$$D^S = \{\vec{d^S} | \vec{d^S} = \lfloor H(\vec{j_{t_0}} + \vec{d}) \rfloor, \vec{d} \in D, \vec{j_{t_0}} \in Z^n \wedge \lfloor H\vec{j_{t_0}} \rfloor = 0\},$$

where $\vec{j_{t_0}}$ denotes the index points belonging to the first complete tile starting from iteration

$(0, \ldots, 0)$ (tile $(0, \ldots, 0)$).

Given an algorithm with dependence matrix D, for a tiling to be legal, it must hold $HD \geq 0$ (see [IT88], [RS92]). This ensures that tiles are atomic and that the initial execution order is preserved. In the opposite case, any execution order of tiles would result in a deadlock (see Figure 2.15).

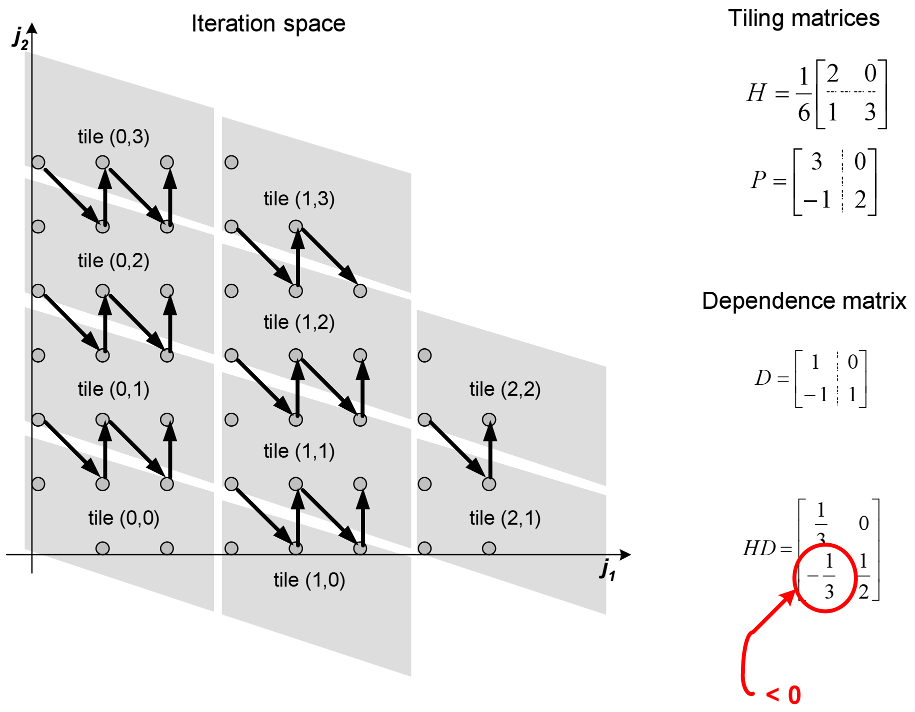


Figure 2.15: Validity of a tiling transformation.

All elements of matrix HD should be non-negative. In this figure $\vec{h_2}\vec{d_1} < 0$. Thus, we can find no time scheduling of tiles which preserves dependences. For example, tile $(1, 2)$ is dependent on tile $(1, 1)$ and tile $(1, 1)$ is dependent on tile $(1, 2)$. Assuming an atomic execution of tiles, this tiling results to a deadlock.

In this thesis, as in [GSK01], we assume that all dependence vectors are smaller than the tile size, thus they are entirely contained in each tile's area. This means that all elements of matrix HD are smaller than 1 ($\vec{h_i}\vec{d_j} \leq 1$, $\forall i, j = 1, \ldots, n$) [Xue97b], or, alternatively, that the tile dependence matrix D^S contains only 0's and 1's. This assumption is quite reasonable, since dependence vectors for common problems are relatively small, while tile sizes may result to be orders of magnitude greater in systems with very fast processors. In this case every tile needs to exchange data only with its nearest neighbors, one in each dimension of J^S.

2.7 Overlapping vs. Non-Overlapping Execution

2.7.1 Non-Overlapping Execution Policy

In [HS98], Hodzic and Shang have presented a scheme for scheduling loops that have been transformed by a tiling transformation. Their approach is to minimize the total execution time, as follows: First, the optimal tiling matrix H is determined and then the tiling transformation H is applied to the original iteration space. The resulting tile space J^S is scheduled using a linear time hyperplane Π. All tiles along a certain dimension are mapped to the same processor. Total execution of tiles consists of successive computation phases interleaved with communication ones. A processor receives the data needed to execute a tile at time step i, performs the computations and sends to its neighboring processors the boundary data, which will be used for tile calculations in time step $i+1$.

Thus, the total execution time is given by formula:

$$T_{nonoverlap} = \wp(t_{comp} + t_{comm}) \tag{2.10}$$

where $\wp$ is the number of time steps needed to complete the parallel execution (makespan), t_{comp} is the execution time of a tile and t_{comm} is the communication time.

Therefore, the overall parallel loop execution consists of atomic computations of tiles interleaved with communication for the transmission of the results to neighboring processors. Since the tile space J^S has only the unitary dependence vectors (see §2.6.3 and §B.5), the optimal linear time schedule can be easily proven to be: $\Pi = [1 \;\; 1 \ldots 1]$ [HS98]. In Figure 2.16, the **non-overlapping execution policy** is shown.

A possible implementation of this execution model can be summarized by the following pseudocode:

```
foracross (t_1=l_1^S; t_1 ≤ u_1^S; t_1++)
  ...
  foracross (t_{n-1}=l_{n-1}^S; t_{n-1} ≤ u_{n-1}^S; t_{n-1}++)
    /*Sequential execution of tiles assigned to this CPU*/
    for (t_n=l_n; t_n ≤ u_n; t_n++){
        Receive data from neighboring tiles
        Compute this tile
        Send data to neighboring tiles
    }
```

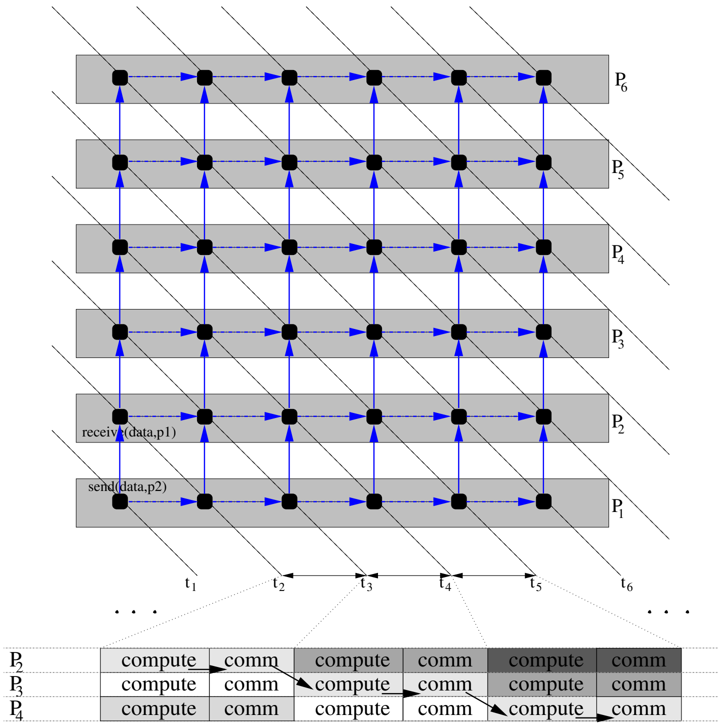


Figure 2.16: Non-overlapping Execution Policy

for a tile space, using six processors. We see that the overall schedule has computation subphases interleaved with communication ones.

2.7.2 Overlapping Execution Policy

The previous quite straightforward model of execution results in very good execution times, since it exploits all inherent parallelism at the tile level. However, one of its important drawbacks is that each processor has to wait for essential data before starting the computation of a certain tile, and wait for the transmission of the results to its neighbors, thus resulting in significant idle processor time. It would be ideal if a node was able to receive, compute and send data at the same time. Modern network interfaces (NICs) have DMA engines that enable them to work in parallel with the CPU. This means that some communication work can be overlapped with actual CPU cycles. In fact, even some part of the non-blocking communication needs the CPU, i.e. DMA initialization. Nevertheless, all subsequent data transferring actions can be ideally overlapped with useful computation.

However, what really imposes such inefficient processor utilization, is the data flow between successive time steps. Specifically, it seems that computations and respective communication substeps for each time step should be serialized to preserve the correct execution order. Every processor should first receive data, then compute and finally send the results to be used at the next time step by its neighbor. A much more thorough look at the correct data flow in the non-overlapping case, reveals the following interesting property: If we slightly modify the initial linear schedule, then we could overlap some communication time with computations. This means that, in each time step, the processor should send and receive data that is not directly dependent to the data computed at this step. A valid time execution policy would be for a processor to receive data from all neighbors to use them at $k+1$ time step, send data produced at previous time step $(k-1)$ and compute its results (Figure 2.17). In this case, every processor computes a tile and, at the same time, sends data produced in the previous step and receives data needed in next one. In Figure 2.17 the **overlapping execution policy** is shown. A more detailed description of this schedule can be found in [GSK01], [STK02], [Sot04].

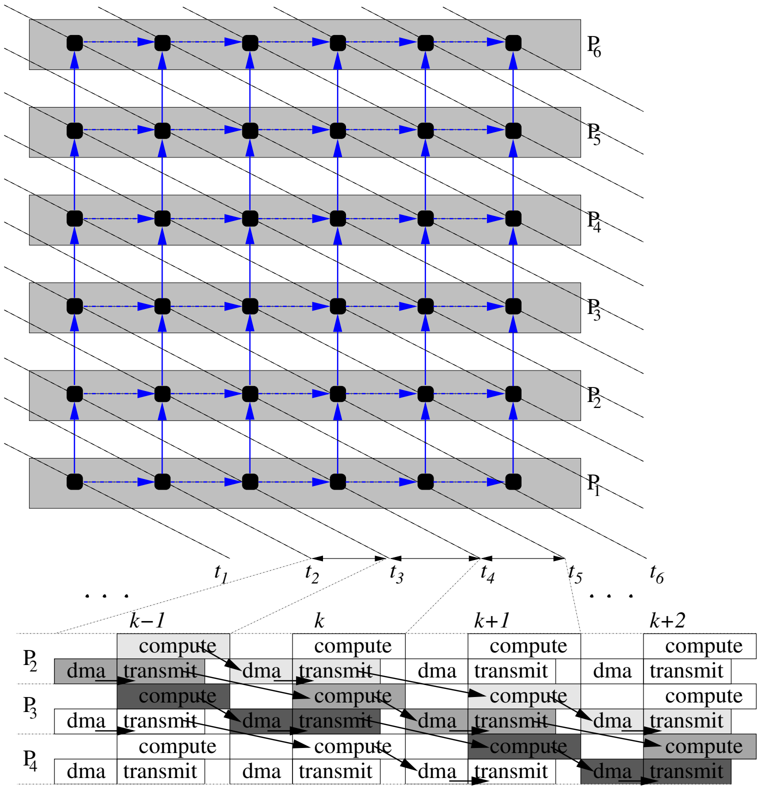


Figure 2.17: Overlapping Execution Policy.
Consider, for example, processor P_3 at k time step: while it computes a tile, it concurrently performs the following: sends the results produced during $k-1$ time step and receives data from neighbors, to be used during the computation of the next tile at $k+1$ time step. Note the arcs shown in this figure. They depict the actual flow of data between successive time steps (computes-sends-receives) in a pipelined way. The outcome of this schedule is to have successive computations overlapped with communication phases, thus 100% processor utilization.

If we implement the overlapping of computation and communication, then we will have the following scheme: A processor first initiates all the non-blocking send operations and then performs the actual atomic tile computations. While the processor performs computations, the NIC is receiving data from neighbors and sending previously computed data to others as well. When communication work is finished, the processor receives an interrupt.

A possible implementation of this execution model can be summarized by the following pseudocode:

```
foracross (t_1=l_1^S; t_1 ≤ u_1^S; t_1++)
  ...
  foracross (t_{n-1}=l_{n-1}^S; t_{n-1} ≤ u_{n-1}^S; t_{n-1}++)
    /*Sequential execution of tiles assigned to this CPU*/
    for (t_n=l_n; t_n ≤ u_n; t_n++){
        Initialize DMA card
        Compute this tile
        Wait for send & receive to complete
        Synchronize with neighbors
    }
```

According to the previous properties, the total execution time for the overlapping schedule, as deduced from Figure 2.17, is given by:

$$T_{overlap} = \wp(t_{start\_dma} + \max(t_{comp}, t_{comm\_dma}) + t_{synchro}), \tag{2.11}$$

where $\wp$ is the number of time steps of the parallel execution (makespan). The time needed to initiate the DMA engine is t_{start_dma}, t_{comp} is the tile execution time, t_{comm_dma} is the communication time which can be overlapped with computation and $t_{synchro}$ is the required synchronization time between successive time steps. In correlation to the parameters used in equation (2.10), it holds that: $t_{init\_dma} + t_{comm\_dma} + t_{synchro} = t_{comm}$

Since the concept of overlapping of actions is crucial, it should be noted that the actions initiated by a non-blocking call are overlapped with the actions initiated by calls following the non-blocking call. On the contrary, a blocking call implies no overlapping of actions, since a following call can be initiated only after the blocking call has completed.

In order to achieve actual overlapping of computation and communication, hardware should assist. The CPU and the NIC must be able to work simultaneously on different tasks. The most important issue is support from DMA, which should exist and be enabled to the NIC. Another aspect is that the invocation of DMA communication should be done in user level (User-Level DMA), without kernel intervention. Furthermore, zero-copy communications should be used and finally, the software packetization process involved in every communication must be avoided. All these prerequisites are discussed in the following section.

2.8 Hardware High Performance Features

Recent advances in high speed networks and improved microprocessor performance are making clusters of workstations an appealing vehicle for cost effective parallel computing. The trend in parallel computing is to move away from custom-designed platforms of the established HPC industry to general purpose systems consisting of loosely coupled components built up from single or multi-processor workstations or PCs.

The de-facto 100Mbps networking of commodity clusters can be a bottleneck for many applications, when scaling beyond a small number of nodes. The last years, new networking technologies such as SCI [Hel99], Myrinet and Gigabit Ethernet offer increased bandwidth and low startup latencies, which however, are never efficiently utilized by user applications. Therefore, high-performance clusters are introduced, which provide the computationally intensive applications with increased performance using special communication primitives, such as Zero-Copy Protocols and DMA transfers.

2.8.1 Zero-Copy Protocols

Network protocol stacks, such as TCP/IP, aggravate the communication procedure with the extra copying of data sent or received, to and from kernel space, respectively. As Figure 2.18 depicts, when sending data from an application (user space) buffer to the network, data must be initially copied from the application buffer to kernel buffers. TCP, IP and network headers must be added and then, as a packet, transferred to NIC's buffer for transmission. A respective procedure takes place when data reach the receiving node.

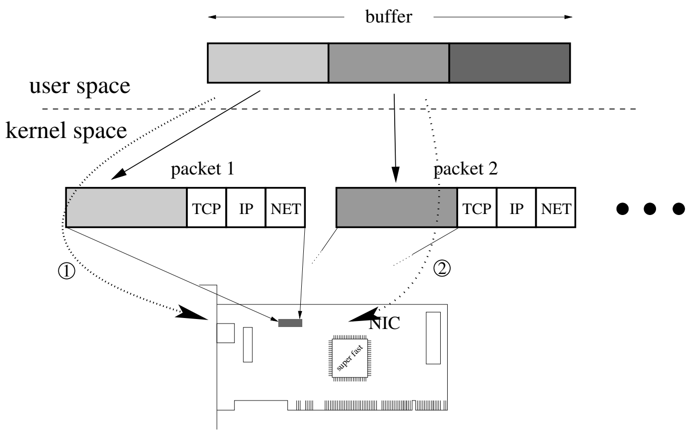


Figure 2.18: Single-Copy Protocol and packetization process

The previous sequence of actions is unavoidable when using legacy network technologies, but could be avoided when novel communication technologies are used. SCI achieves Zero-Copy Communication, since it supports a Distributed Shared Memory approach, which is implemented using kernel area memory mapped regions for communication. An SCI communication scenario involves the following stages: A process in an SCI node exports a memory segment, which is imported by a process that resides in another SCI node. Every imported memory segment is directly mapped to the PCI I/O space of the PCI-SCI NIC. It is part of the importer's (process) virtual memory through the prior invocation of an `SCIConnectSegment()` driver call. When the importing node needs to send data, it just writes them directly to the imported memory segment (thus, no kernel copies). Data are transferred to the exporter's memory and communication is performed, without any kernel intervention. No other data processing is needed within each send.

2.8.2 DMA transfers

Message data can be usually transferred in two ways: Programmed I/O (PIO) mode and DMA mode. In PIO mode, CPU handles data transferring completely, word by word. For example, data transferring of 1Kwords involves the initial copying of these words from main memory to the NIC's buffers with the aid of CPU. From a parallel application's point of view, these are considered "lost" CPU cycles, since useful calculations could have been executed instead. On the contrary, using DMA mode, CPU just programs the NIC's DMA engine with the information of which data to transfer from main memory and where to send it. CPU is not used (or blocked from a program's perspective) during the transfer and can perform other (useful) tasks.

The DSM feature of SCI allows the efficient use of its DMA capabilities. Using special SCI driver calls, the system returns physically contiguous allocated memory. This is performed using the `__get_free_pages()` kernel routine. The allocated memory is first "pinned down" and then mapped to user's virtual memory (Figure 2.19). User is able to read/write that memory region like the ordinary memory regions returned by LIBC `malloc()`. Despite the fact that DMA transfer is only invoked as a kernel system call, the complete transfer of the specific memory area will be performed with only one DMA invocation. On the contrary, even if the NIC in Figure 2.18 was DMA enabled, a new DMA invocation should take place for each {data,TCP,IP,NET} packet, which would be time consuming.

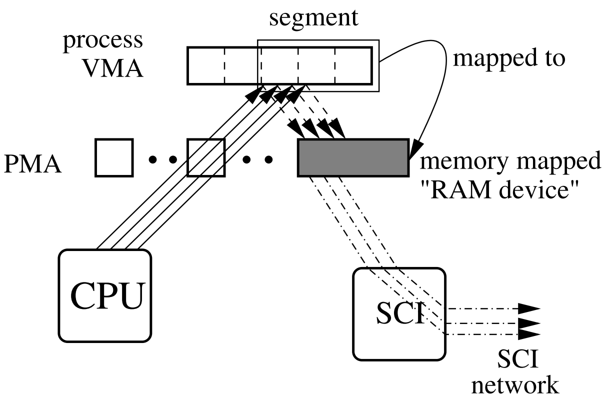


Figure 2.19: Locked and memory mapped "RAM device" for SCI communications

3

Automatic parallel code generation for tiled nested loops

In this chapter, we briefly describe an approach for the problem of automatically generating parallel code for tiled nested loops. Our method is applied to general parallelepiped tiles and non-rectangular space boundaries as well. It consists of two steps:

1. *generating sequential tiled code*
2. *parallelizing the sequential tiled code*

In order to generate sequential code efficiently, the original problem is divided into the subproblems of enumerating the tiles and sweeping the points inside every tile. In order to parallelize the sequential tiled code, we address issues such as data distribution, iteration distribution and automatic message passing.

3.1 Introduction

The tiling transformation, as described in §2.6.2, has been used in literature in two different contexts:

- in order to ensure the locality of data references and reduce the overall execution time through an efficient utilization of cache memory levels [Jim99]
- in order to parallelize the execution of a nested loop code segment with dense dependences, as described in §2.2 and §2.6.2 of this thesis.

A lot of research has been conducted, concerning the selection of optimal tile size and shape, that reduce the communication cost [BDRR94], [Xue97a], or the time processors remain idle [HS02], [HCF99], [XC02]. However, the parallelizing compilers community has been pessimistic about using non-rectangular tiling transformations to execute nested loops in distributed memory machines. General parallelepiped tiling has not been used in either commercial or research compilers ([AMC97], [AL93], [CMZ92], [FHK+91], [SLR+95]). This is due to the fact that a significant overhead is imposed by non-rectangular tiling to both compile time and run time of the final parallel code. Apart from [ACN+00], [XC02], that present some experimental results for 2-dimensional spaces, all previous research on non-rectangular tiling is purely theoretical. All complete frameworks for the automatic generation of parallel tiled code, such as the one presented in [TX00], can be applied only for rectangular tiling. In this chapter, as in [GAK03], we present a method for automatically producing non-rectangular tiled code without imposing a prohibitive overhead either at compile or at run time.

The parallelization of a nested loop code segment, as depicted in Figure 3.1, consists of the following three steps at minimum:

1. A dependence analysis is conducted [Ban88], [Pug92], so as to determine the optimal tiling transformation, which minimizes the communication overhead among processors [BDRR94], [Xue97a], or the time processors remain idle waiting for the data needed to arrive from neighboring processors [HS02], [HCF99], [XC02].
2. The initial code segment is converted to serial tiled code, according to the tiling transformation selected in the previous step, as described in [GAK02b], [GAK03]. This conversion is consisted of two substeps:
   (a) Producing the bounds of the tile space from the bounds of the iteration space and
   (b) Producing the appropriate boundary expressions for traversing the internal of each tile, as well as determining the incremental steps of each loop index.
3. Parallelizing the serial tiled code, as described in [GDAK02a]. This step consists of
   (a) the distribution of data and computations among processors and

(b) the automatic generation of the message passing primitives

Figure 3.1: Automatic parallel code generation for tiled iteration spaces.
After selecting the optimal tiling transformation, the initial untiled code segment should be converted into serial tiled code. Then, the serial tiled code should be parallelized.

3.2 Generation of Serial Tiled Code

In this section, we elaborate on generating tiled code that will traverse an iteration space J^n transformed by a tiling transformation. We call this code **sequential tiled code**. By applying tiling to J^n, we obtain the tile space J^S, the tile iteration space TIS and the tile origin space TOS. In §2.6.2, it was shown that tiling transformation is a $Z^n \longrightarrow Z^{2n}$ transformation, which means that a point $\vec{j} \in J^n$ is transformed into a tuple of n-dimensional factors $(\vec{j_a}, \vec{j_b})$, where $\vec{j_a}$ identifies the tile that the original point belongs to ($\vec{j_a} \in J^S$) and $\vec{j_b}$ identifies the coordinates of the point relevant to the tile origin ($\vec{j_b} \in TIS$). The sequential tiled code reorders the execution of indices enforced by their lexicographic order, resulting in an execution order described by the following scheme:

```
FOR (EVERY tile IN tile space J^S) TRAVERSE THE POINTS IN ITS INTERIOR
```

According to the above, the sequential tiled code consists of a $2n$-dimensional nested loop. The n outermost loops traverse the tile space J^S, using indices $j_1^S, j_2^S, \ldots, j_n^S$, and the n innermost loops traverse the points within tile $(j_1^S, j_2^S, \ldots, j_n^S)$, using indices $j'_1, j'_2, \ldots, j'_n$. We denote l_k^S, u_k^S the lower and upper bounds of index j_k^S, respectively. Similarly, we denote l'_k, u'_k the lower and upper bounds of index j'_k. In all cases, lower bounds (l_k^S or l'_k) are of the form: $max(l_{k,0}, l_{k,1}, \ldots)$ and upper bounds (u_k^S or u'_k) of the form: $min(u_{k,0}, u_{k,1}, \ldots)$, where $l_{k,j}$, $u_{k,j}$ are affine functions of the outermost indices. The calculation of factors $l_1^S, \ldots, l_n^S$ and $u_1^S, \ldots, u_n^S$ corresponds to substep 2a of §3.1, while the calculation of factors $l'_1, \ldots, l'_n$ and $u'_1, \ldots, u'_n$ corresponds to substep 2b.

3.2.1 Enumerating the tiles

A conventional approach

Ancourt and Irigoin in [AI91] dealt with the subproblem of traversing the tile space, by constructing an appropriate set of inequalities. According to their approach, a tile $\vec{j^S}$ belongs to the tile space J^S ($\vec{j^S} \in J^S$), iff there is an iteration $\vec{j}$, which fulfills both criteria:

1. It belongs to the iteration space J^n. That is, $\vec{j} \in J^n \Leftrightarrow$

$$B\vec{j} \leq \vec{b}$$

(recall formula (2.1)).

2. It belongs to tile $\vec{j^S}$ with origin iteration $\vec{j_0} = H^{-1}\vec{j^S}$ (recall formula (2.6)). Note that, according to the definitions given in §2.6.2, a point $\vec{j}$ belongs to a tile with tile origin $\vec{j_0}$, iff it satisfies the set of inequalities: $S(\vec{j} - \vec{j_0}) \leq \vec{s}$. Replacing in this set $\vec{j_0} = H^{-1}\vec{j^S}$, it can be equivalently written as:

$$\begin{pmatrix} -gI & gH \\ gI & -gH \end{pmatrix} \begin{pmatrix} \vec{j^S} \\ \vec{j} \end{pmatrix} \leq \vec{s}$$

Combining the above systems, we obtain the final system of inequalities:

$$\begin{pmatrix} 0 & B \\ -gI & gH \\ gI & -gH \end{pmatrix} \begin{pmatrix} \vec{j^S} \\ \vec{j} \end{pmatrix} \leq \begin{pmatrix} \vec{b} \\ \vec{s} \end{pmatrix} \tag{3.1}$$

Ancourt and Irigoin propose the application of Fourier-Motzkin elimination method to the above system in order to obtain proper formulas for the lower and upper bounds of the $2n$-dimensional loop that will traverse the tiled space. Note that the n outermost loop boundaries produced are appropriate for traversing the tile space. The n innermost loop boundaries are appropriate for scanning the interior of tiles and can be presently ignored.

Example 3.1: Consider the following nested loop code segment:

```
for (j1 = 0; j1 <= 39)
  for (j2 = 0; j2 <= 29){
    A[j1,j2]=A[j1 - 1,j2 - 2]+A[j1 - 3,j2 - 1];
  }
```

The corresponding iteration space J^n is: $J^n = \{(j_1, j_2)|0 \leq j_1 \leq 39, 0 \leq j_2 \leq 29\}$. Let us apply a tiling transformation defined by matrix

$$H = \begin{bmatrix} \frac{1}{5} & -\frac{1}{10} \\ -\frac{1}{20} & \frac{3}{20} \end{bmatrix} \text{ or, equivalently, by } P = \begin{bmatrix} 6 & 4 \\ 2 & 8 \end{bmatrix}$$

which is legal [RS92] (since $HD \geq 0$) and has both communication and scheduling-optimal shape ([BDRR94], [HS98], [HS02], [HCF97], [Xue97a]), for the specific problem. Then, as shown in

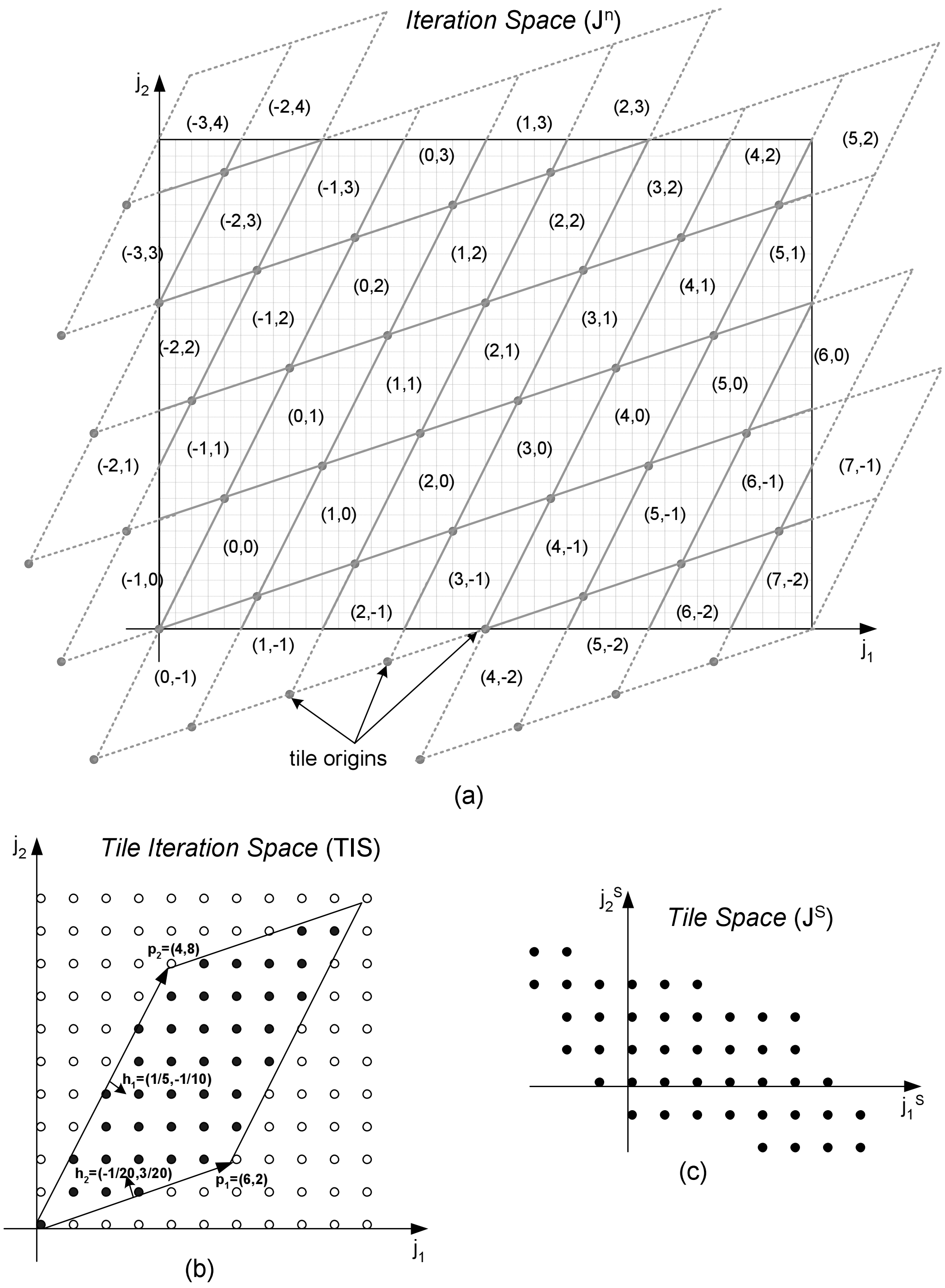


Figure 3.2: Example 3.1: Representation of the spaces used.

(a) The initial iteration space is partitioned into identical parallelogram tiles, which are identified by a unique vector indicated inside each tile. The origin of each tile has been illustrated by a grey dot. Some of the origins may not belong to the initial iteration space J^n. (b) The tile iteration space includes all iterations of tile $(0, 0)$, which starts at the axes origin. (c) The tile space J^S is derived from the iteration space by formula (2.4). All iterations of the the same tile in subfigure (a) are mapped to only one point in J^S of subfigure (c).

Figure 3.2b, TIS contains the points $\{(0,0), (1,1), (1,2), (2,1), (2,2), (2,3), (2,4), \ldots, (7,5), (7,6), (7,7), (7,8), (8,7), (8,8), (8,9), (9,9)\}$. In addition, as shown in Figure 3.2c, J^n is transformed by matrix H to the tile space $J^S = \{(-3,3), (-3,4), (-2,1), (-2,2), (-2,3), (-2,4), \ldots, (6,-2), (6,-1), (6,0), (7,-2), (7,-1)\}$. In the sequel, as shown by the grey dots in Figure 3.2a, the tile space J^S is transformed by matrix P to $TOS = \{(-6,18), (-2,26), (-8,4), (-4,12), (0,20), (4,28), \ldots, (28,-4), (32,-4), (36,12), (34,-2), (38,6)\}$.

The set of inequalities describing the iteration space J^n is:

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \\ -1 & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} j_1 \\ j_2 \end{pmatrix} \leq \begin{pmatrix} 39 \\ 29 \\ 0 \\ 0 \end{pmatrix}$$

The system of inequalities $S(\vec{j} - \vec{j_0}) \leq \vec{s}$ (see formulas (2.8), (2.9)) describing a tile is (since $g = 20$):

$$\begin{pmatrix} 4 & -2 \\ -1 & 3 \\ -4 & 2 \\ 1 & -3 \end{pmatrix} \begin{pmatrix} j_1 - j_{01} \\ j_2 - j_{02} \end{pmatrix} \leq \begin{pmatrix} 19 \\ 19 \\ 0 \\ 0 \end{pmatrix}$$

Thus, according to formula (3.1), the final system proposed my Ancourt and Irigoin for the calculation of loop indices is:

$$\begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \\ -20 & 0 & 4 & -2 \\ 0 & -20 & -1 & 3 \\ 20 & 0 & -4 & 2 \\ 0 & 20 & 1 & -3 \end{pmatrix} \begin{pmatrix} \vec{j^S} \\ \vec{j} \end{pmatrix} \leq \begin{pmatrix} 39 \\ 29 \\ 0 \\ 0 \\ 19 \\ 19 \\ 0 \\ 0 \end{pmatrix}$$

This system of inequalities is not suitable for a nested loop code segment, since it contains no inequalities for the expressions of outer loop boundaries of j_1^S and j_2^S. An application of the Fourier-Motzkin elimination method (see §2.4) can convert it to the equivalent system of inequalities:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 1 & 4 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 3 & 2 & 0 & 0 \\ -1 & -4 & 0 & 0 \\ -3 & -2 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ -5 & 0 & 1 & 0 \\ 0 & 20 & 1 & 0 \\ -6 & -4 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & -20 & -1 & 0 \\ 5 & 0 & -1 & 0 \\ 6 & 4 & -1 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & -20 & -1 & 3 \\ 10 & 0 & -2 & 1 \\ 0 & 0 & 0 & -1 \\ -10 & 0 & 2 & -1 \\ 0 & 20 & 1 & -3 \end{pmatrix} \begin{pmatrix} \vec{j^S} \\ \vec{j} \end{pmatrix} \leq \begin{pmatrix} 7 \\ 3 \\ 14 \\ 4 \\ 19 \\ 4 \\ 4 \\ 2 \\ 19 \\ 87 \\ 9 \\ 39 \\ 19 \\ 0 \\ 0 \\ 0 \\ 29 \\ 19 \\ 0 \\ 0 \\ 9 \\ 0 \end{pmatrix} \tag{3.2}$$

Only the eight first rows of this system are useful for traversing the tile space J^S. We may cut them off and go on with the system of inequalities:

$$\begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 1 & 4 \\ 0 & 1 \\ 3 & 2 \\ -1 & -4 \\ -3 & -2 \\ 0 & -1 \end{pmatrix} \vec{j^S} \leq \begin{pmatrix} 7 \\ 3 \\ 14 \\ 4 \\ 19 \\ 4 \\ 4 \\ 2 \end{pmatrix}$$

An application of the ad-hoc simplification method [BW95] can detect and eliminate two redundant inequalities. Finally, the simplified system

$$\begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 1 & 4 \\ 3 & 2 \\ -1 & -4 \\ -3 & -2 \end{pmatrix} \vec{j^S} \leq \begin{pmatrix} 7 \\ 3 \\ 14 \\ 19 \\ 4 \\ 4 \end{pmatrix}$$

may be used for automatically producing the code, which scans the tile space:

```
for(j_1^S=-3; j_1^S <=7; j_1^S++)
  for(j_2^S=max(⌈(-4-j_1^S)/4⌉, ⌈(-4-3j_1^S)/2⌉); j_2^S <=min(⌊(14-j_1^S)/4⌋,⌊(19-3j_1^S)/2⌋); j_2^S++){
    Execute tile (j_1^S, j_2^S)
  }
```

Reducing the compile time overhead of tiling

In order to reduce the overhead imposed at compile time by tiling, we should primarily reduce the complexity of the Fourier-Motzkin elimination method used. Recall from §2.4 that it depends doubly exponentially on the number of loops involved. Thus, in order to decrease the compile time overhead, we should first of all examine whether we may reduce the number of loop indices involved in the set of inequalities (3.1).

The subproblem of traversing the tile space J^S has been considered by many authors as an example of applying the non-unimodular tiling transformation to the original iteration space. More specifically, Ramanujam in [Ram92] and [Ram95] applied the non-unimodular tiling transformation to the set of inequalities $B\vec{j} \leq \vec{b}$ describing the iteration space, as follows: $B\vec{j} \leq \vec{b} \Rightarrow BH^{-1}H\vec{j} \leq \vec{b} \Rightarrow$

$$BP\vec{j^S} \leq \vec{b} \tag{3.3}$$

Here again, the application of Fourier-Motzkin elimination method to the derived system of inequalities is proposed, in order to obtain closed form formulas for tile bounds $l_1^S, \ldots, l_n^S$ and $u_1^S, \ldots, u_n^S$.

Unfortunately, the previous approach fails to enumerate tiles exactly. This is because the system of inequalities in (3.3) is satisfied by points in the tile space J^S, whose tile origins belong to J^n. However, as stated in §2.6.2, there exist some points in TOS that do not belong to J^n.

Although these points do not satisfy the preceding systems of inequalities, they must be traversed as well. In Figure 3.2a, tiles in the lower boundaries, such as (-3,3), (-2,1), (4,-2) and others, are not scanned by this method, because their origins do not belong to the original iteration space J^n. Consequently, a modification is required, so that Fourier-Motzkin elimination method can scan all tiles correctly. As shown in Figure 3.5, what is needed is a proper reduction of the lower bounds and/or a proper increase of the upper bounds of our space, in order to include all tile origins. Lemma 3.1 determines how much we must expand space bounds, in order to include all points of TOS.

Lemma 3.1 *If we apply tiling transformation P to an iteration space J^n, whose bounds are expressed by the system of inequalities $B\vec{j} \leq \vec{b}$, then for all tile origins $\vec{j_0} \in TOS$, it holds:*

$$B\vec{j_0} \leq \vec{b'}, \tag{3.4}$$

where $\vec{b'}$ is determined by the expression:

$$b'_i = b_i + \frac{g-1}{g}\sum_{r=1}^{n}(\vec{\beta_i} \cdot \vec{p_r})^-, i = 1, \ldots, n \tag{3.5}$$

where $\vec{\beta_i}$ is the i-th row of matrix B, $\vec{p_r}$ is the r-th column of matrix P and $(\vec{\beta_i} \cdot \vec{p_r})^- = max(-\vec{\beta_i} \cdot \vec{p_r}, 0)$.

Proof: *We suppose that point $\vec{j} \in J^n$ belongs to tile with origin $\vec{j_0}$. Since P consists of n linearly independent vectors, $\vec{j}$ can be expressed as the sum of $\vec{j_0}$ and a linear combination of the column-vectors of the tiling matrix P:*

$$\vec{j} = \vec{j_0} + \sum_{l=1}^{n}\lambda_l\vec{p_l} \tag{3.6}$$

In addition, as in formula (2.8), the following system of inequalities holds: $0 \leq gH(\vec{j} - \vec{j_0}) \leq (g-1)$. The i-th row of this inequality can be rewritten as follows: $0 \leq \vec{h_i} \cdot (\vec{j} - \vec{j_0}) \leq \frac{g-1}{g}$, where $\vec{h_i}$ is the i-th row-vector of matrix $H = P^{-1}$. Replacing in this expression by (3.6), we get:

$$0 \leq \vec{h_i} \cdot \sum_{l=1}^{n}\lambda_l\vec{p_l} \leq \frac{g-1}{g}$$

As $P = H^{-1}$ it holds that $\vec{h_i} \cdot \vec{p_i} = 1$ and $\vec{h_i} \cdot \vec{p_l} = 0$ if $i \neq l$. Consequently, the last formula can be rewritten as follows:

$$0 \leq \lambda_i \leq \frac{g-1}{g}$$

for all $i = 1, \ldots, n$. If multiplied by $\vec{\beta_k} \cdot \vec{p_i}$, this inequality gives:

1. *If $\vec{\beta_k} \cdot \vec{p_i} \geq 0$: $\lambda_i\vec{\beta_k} \cdot \vec{p_i} \geq 0$*

2. If $\vec{\beta_k} \cdot \vec{p_i} < 0$: $\lambda_i \vec{\beta_k} \cdot \vec{p_i} \geq \frac{g-1}{g} \vec{\beta_k} \cdot \vec{p_i}$

According to the definitions of the symbol $(\vec{\beta_k} \cdot \vec{p_i})^- = max(-\vec{\beta_k} \cdot \vec{p_i}, 0)$, the previous inequalities can in every case be rewritten as follows: $\lambda_i \vec{\beta_k} \cdot \vec{p_i} \geq -\frac{g-1}{g}(\vec{\beta_k} \cdot \vec{p_i})^- \Rightarrow -\lambda_i \vec{\beta_k} \cdot \vec{p_i} \leq \frac{g-1}{g}(\vec{\beta_k} \cdot \vec{p_i})^-$. If added for $i = 1, \ldots, n$, this inequality gives:

$$-\sum_{i=1}^{n} \lambda_i \vec{\beta_k} \cdot \vec{p_i} \leq \frac{g-1}{g} \sum_{i=1}^{n} (\vec{\beta_k} \cdot \vec{p_i})^- \tag{3.7}$$

For each $\vec{j} \in J^n$ the system of inequalities $B\vec{j} = \vec{b}$ holds. The k-th row of this system can be written as follows: $\vec{\beta_k} \cdot \vec{j} \leq b_k$. We can replace $\vec{j}$ in this inequality, using formula (3.6) as follows: $\vec{\beta_k} \cdot (\vec{j_0} + \sum_{i=1}^{n} \lambda_i \vec{p_i}) \leq b_k \Rightarrow \vec{\beta_k} \cdot \vec{j_0} \leq b_k - \vec{\beta_k} \cdot (\sum_{i=1}^{n} \lambda_i \vec{p_i})$

$$\Rightarrow \vec{\beta_k} \cdot \vec{j_0} \leq b_k - \sum_{i=1}^{n} \lambda_i (\vec{\beta_k} \cdot \vec{p_i})$$

If we combine this inequality with (3.7), we conclude that $\vec{\beta_k} \cdot \vec{j_0} \leq b_k + \frac{g-1}{g} \sum_{i=1}^{n} (\vec{\beta_k} \cdot \vec{p_i})^-$. Thus, for each tile with origin $\vec{j_0}$, which has at least one point in the initial iteration space, it holds that $B\vec{j_0} \leq \vec{b'}$, where the vector $\vec{b'}$ is constructed so as its k-th element is given by the form: $b'_k = b_k + \frac{g-1}{g} \sum_{i=1}^{n} (\vec{\beta_k} \cdot \vec{p_i})^-$. ⊣

If we work with the tile space J^S and take into account that $\vec{j_0} = P\vec{j^S}$, we equivalently get the system of inequalities:

$$BP\vec{j^S} \leq \vec{b'} \tag{3.8}$$

If it is given that matrix B consists of only integer elements, $\vec{b'}$, can be determined by the expression:

$$b'_i = b_i + \lfloor \frac{g-1}{g} \sum_{r=1}^{n} (\vec{\beta_i} \cdot \vec{p_r})^- \rfloor, i = 1, \ldots, n \tag{3.9}$$

Geometrical interpretation: The term added to each element of $\vec{b}$ expresses a parallel shift of the corresponding bound of the initial space. In Figure 3.3, we present an example of our method. Each row $\vec{\beta_i}$ of matrix B expresses a vector vertical to the corresponding bound of the iteration space with its direction outwards. The equation of this boundary surface is $\vec{\beta_i} \cdot \vec{x} = b_i$. A parallel shift of this surface by a vector $\vec{x_0}$ is expressed by the equation $\vec{\beta_i} \cdot (\vec{x} - \vec{x_0}) = b_i \Leftrightarrow \vec{\beta_i} \cdot \vec{x} = b_i + \vec{\beta_i} \cdot \vec{x_0}$. As shown in Figure 3.3, we shift a boundary surface by vector $-\vec{p_r}$, iff the tile edge-vector $\vec{p_r}$ forms an angle greater than 90^o with vector $\vec{\beta_i}$ (as the angles between the vectors $\vec{\beta_1}$ and $\vec{p_1}$, $\vec{\beta_1}$ and $\vec{p_2}$, $\vec{\beta_3}$ and $\vec{p_1}$, $\vec{\beta_3}$ and $\vec{p_2}$, $\vec{\beta_4}$ and $\vec{p_1}$ of Figure 3.3), or, equivalently, iff $\vec{p_r} \cdot \vec{\beta_i} < 0$.

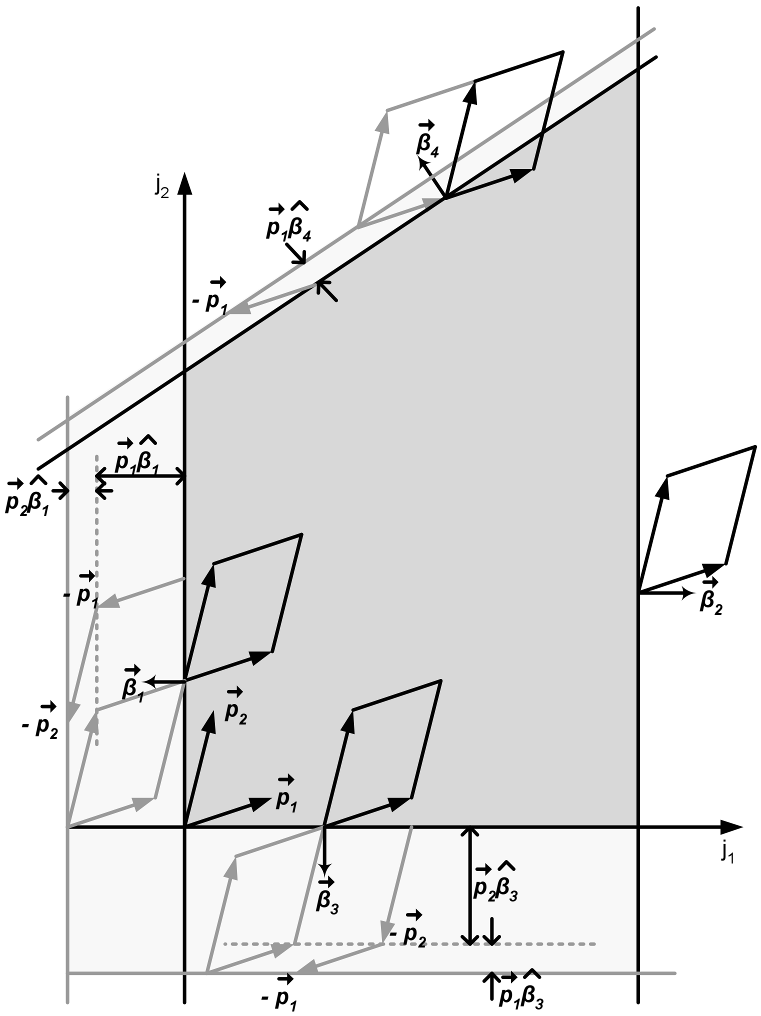


Figure 3.3: Expanding iteration space bounds to include all tile origins.
The dark grey area corresponds to the initial iteration space area. The light grey area indicates the expansion of the iteration space, in order to include all tile origins. It is shown that an iteration space boundary is shifted, iff there is an inverse tiling vector $\vec{p_r}$, which traverses this boundary outside $\rightarrow$ inside.

This fact can be expressed as follows: if the dot product of $\vec{p_r}$ (one of the columns of the matrix P) and $\vec{\beta_i}$ (a row of B) is negative, then this dot product is subtracted from the constant b_i. Equivalently, in formula (3.5) the term $(\vec{\beta_i} \cdot \vec{p_r})^-$ is added to the constant b_i for all vectors $\vec{p_r}$. The multiplying factor $\frac{g-1}{g}$ expresses the fact that a tile is a semiopen hyperparallelepiped and thus we need not contain in the tile space the tiles which just touch the initial iteration space.

Note, however, it was proven that the expanded space includes *all* origins of tiles in J^S. It was not proven that it contains *only* origins of tiles in J^S. In other words, this expansion of bounds may include some redundant tiles, whose origins belong to the extended space, but their internal points remain outside the original iteration space. These tiles will be accessed, but their internal points will not be swept, as it will be shown next, thus imposing little computation

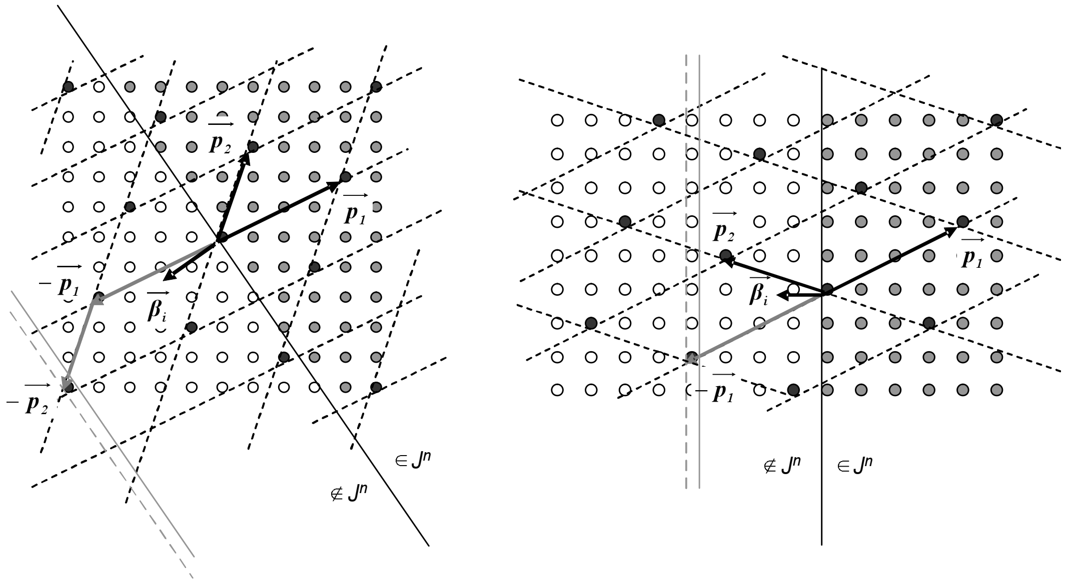


Figure 3.4: Expanding iteration space bounds to include all tile origins.
The grey dots correspond to iteration inside J^n, while the white dots correspond to iterations outside J^n. This figure indicates that the expansion of the iteration space should be less than the dot product of vectors $\vec{\beta_i}$ and $\vec{p_r}$, so as not to include tiled that just touch the initial iteration space boundaries, with no integer points inside J^n. The dashed grey lines corresponds to the expansion of bounds, according to the dot product of vectors $\vec{\beta_i}$ and $\vec{p_r}$. The solid grey lines correspond to the final expansion, so as not to include a lot of redundant tiles.

overhead in the execution of the sequential tiled code.

Example 3.2: We will now enumerate the tiles generated by the tiling transformation described in Example 3.1, using the method described just above. Following our approach, we should construct the system of inequalities in (3.8) making use of the expression in (3.9). Expression (3.9) in our case gives $\vec{b'} = \begin{pmatrix} 39 & 29 & 9 & 9 \end{pmatrix}^T$ and thus, the system in (3.8) becomes:

$$\begin{pmatrix} 6 & 4 \\ 2 & 8 \\ -6 & -4 \\ -2 & -8 \end{pmatrix} \begin{pmatrix} j_1^S \\ j_2^S \end{pmatrix} \leq \begin{pmatrix} 39 \\ 29 \\ 9 \\ 9 \end{pmatrix}$$

The expansion of bounds for this example is shown in Figure 3.5. An application of the Fourier-

Motzkin elimination method can convert this system to its equivalent:

$$\begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 3 & 2 \\ 1 & 4 \\ -3 & -2 \\ -1 & -4 \end{pmatrix} \begin{pmatrix} j_1^S \\ j_2^S \end{pmatrix} \leq \begin{pmatrix} 8 \\ 4 \\ 19 \\ 14 \\ 4 \\ 4 \end{pmatrix}$$

Note that the implementation used for the Fourier-Motzkin elimination method can take into account that index variables can only be integer, and further simplify the final expressions, applying the floor or ceiling functions where appropriate. Consequently, a loop that enumerates the tiles in our case has the form:

```
for (j1S = -4; j1S <= 8; j1S++)
  for (j2S =max(⌈(-4-3j1S)/2⌉,⌈(-4-j1S)/4⌉); j2S <=min(⌊(19-3j1S)/2⌋,⌊(14-j1S)/4⌋); j2S++) {
    Execute tile (j1S, j2S)
  }
```

Note that tiles $(8, -3)$ and $(-4, 4)$ are redundant (Figure 3.5).

3.2.2 Scanning the points within a tile

A conventional approach

In order to traverse the internal points of every tile, one can use the n innermost loop indices of the system of inequalities produced when applying the Fourier-Motzkin elimination method to the system (3.1). However, it is more efficient to separately apply the Fourier-Motzkin elimination method to the systems:

1.

$$B\vec{j} \leq \vec{b} \tag{3.10}$$

Recall from formula (2.1) that this systems indicates that iteration $\vec{j}$ belongs to the iteration space.

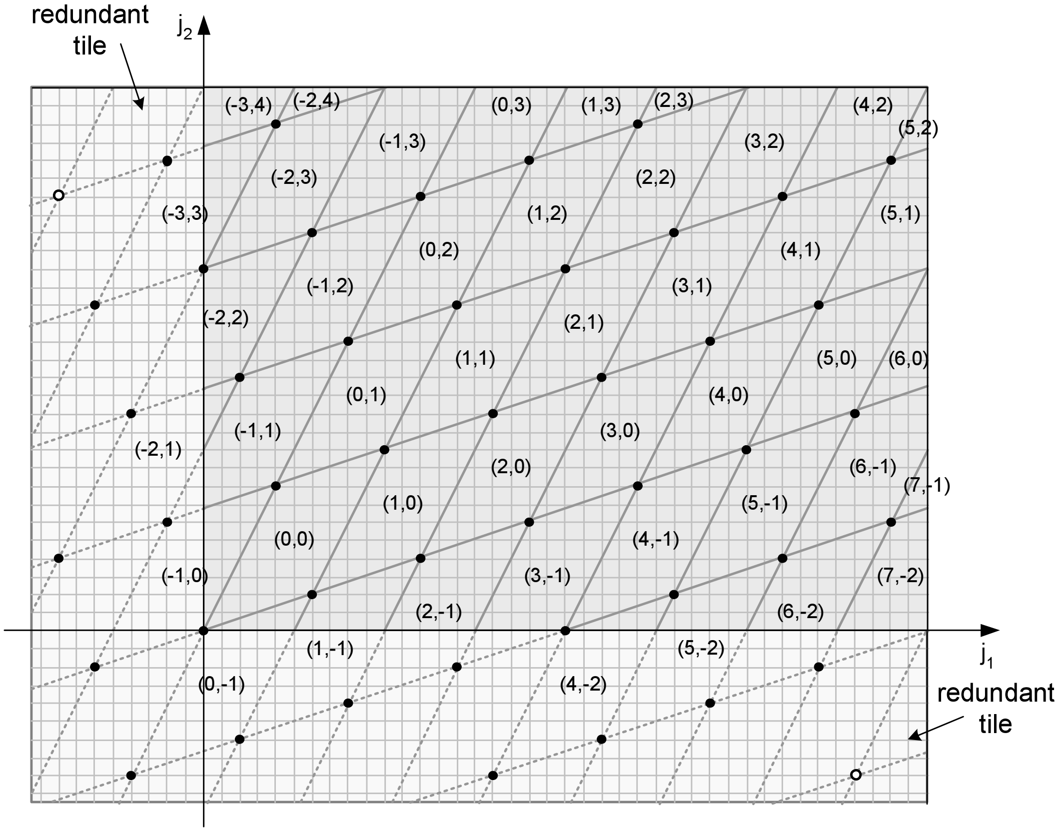


Figure 3.5: Example 3.2: Expanding iteration space bounds to include all tile origins. The dark grey area corresponds to the iteration space area. The light grey area indicates the expansion of the iteration space, in order to include all tile origins, according to formulas (3.4), (3.5). Unfortunately, the expanded area contains also two tile origins, which do not correspond to a tile in J^S. Fortunately, they may be located only in near the edges of the expanded iteration space. Thus, their number is negligible in comparison to the number of tiles of J^S.

2.

$$\begin{pmatrix} gH \\ -gH \end{pmatrix} \begin{pmatrix} \vec{j} - \vec{j_0} \end{pmatrix} \leq \begin{pmatrix} (g-1)\vec{1} \\ \vec{0} \end{pmatrix} \tag{3.11}$$

Recall from formulas (2.8), (2.9) that this system indicates that iteration $\vec{j}$ belongs to tile with origin iteration $\vec{j_0} = H^{-1}\vec{j^S}$.

This modification is used in our implementation for automatically producing tiled code. As deduced during our experimentation, it results to reducing both compile and run time of the final code.

Compile time is reduced because there is no more need for applying the ad-Hoc and exact simplification methods to the whole system produced by (3.1), but only to its subsystem corresponding to the n outer loop indices.

Run time is reduced because the combination of inequalities produced by (3.10) and (3.11) are less than inequalities produced by (3.1). This is partly due to the fact that the exact simplification method may not be able to detect the redundancy of an inequality in Z^n if it is not redundant in R^n. On the other hand, inequalities originating from different systems (3.10) and (3.11) are rarely redundant in respect to each other. Thus, it is almost improbable to have an extra inequality in the final system due to not applying the simplification methods to the combination of systems (3.10), (3.11).

In addition, when this modification is used, it is possible to check even less inequalities for tiles that are not located near a boundary of the iteration space, at run time. If a tile crosses the iteration space boundaries, then all inequalities produced by (3.10), (3.11) should be checked during the scan of the interior of the tile. Otherwise, if a tile does not cross any iteration boundary, only inequalities derived from (3.11) may be checked at run time. This simplification presupposes the use of a method for distinguishing tiles into internal and boundary. As internal we may characterize a tile with all its vertices in J^n.

Lemma 3.2 *If all 2^n vertices of a tile ($\vec{c} = \vec{j_0} + \sum_{i=1}^{n} x_i \frac{g-1}{g} \vec{p_i}$ for $x_i \in \{0,1\}$, $i = 1, \ldots, n$) belong to the convex iteration space J^n, then all iterations of this tile belong to J^n.*

Proof: *According to Lemma C.1, in order to prove this lemma, we may only prove that every iteration inside a tile $\vec{j^S}$ may be calculated by an expression of the form (C.1).*

In the proof of Lemma 3.1, we have written that every iteration $\vec{j}$ can be expressed as the sum of its tile origin $\vec{j_0}$ and a linear combination of the column-vectors of the inverse tiling matrix P:

$$\vec{j} = \vec{j_0} + \sum_{i=1}^{n} \lambda_i \vec{p_i} \tag{3.12}$$

where $0 \leq \lambda_i \leq \frac{g-1}{g}$ for all $i = 1, \ldots, n$. Equation (3.12) can be equivalently rewritten as follows

$$\vec{j} = \sum_{\substack{\forall x_i \in \{0,1\} \\ i=1..n}} \left[\prod_{i=1}^{n} \left[\left(1 - \frac{\lambda_i g}{g-1}\right)(1 - x_i) + \frac{\lambda_i g}{g-1} x_i \right] \left(\vec{j_0} + \sum_{i=1}^{n} x_i \frac{g-1}{g} \vec{p_i} \right) \right] \tag{3.13}$$

since

1. *The total multiplying factor of $\vec{j_0}$ equals to 1.*

$$\sum_{\substack{\forall x_i \in \{0,1\} \\ i=1..n}} \prod_{i=1}^{n} \left[\left(1 - \frac{\lambda_i g}{g-1}\right)(1 - x_i) + \frac{\lambda_i g}{g-1} x_i \right] =$$

$$\sum_{\substack{\forall x_i \in \{0,1\} \\ i=1..n-1 \\ x_n = 0}} \prod_{i=1}^{n} \left[\left(1 - \frac{\lambda_i g}{g-1}\right)(1 - x_i) + \frac{\lambda_i g}{g-1} x_i \right] +$$

$$\sum_{\substack{\forall x_i \in \{0,1\} \\ i=1..n-1 \\ x_n = 1}} \prod_{i=1}^{n} \left[\left(1-\tfrac{\lambda_i g}{g-1}\right)(1-x_i)+\tfrac{\lambda_i g}{g-1}x_i\right] =$$

$$\sum_{\substack{\forall x_i \in \{0,1\} \\ i=1..n-1}} \prod_{i=1}^{n-1} \left[\left(1-\tfrac{\lambda_i g}{g-1}\right)(1-x_i)+\tfrac{\lambda_i g}{g-1}x_i\right]\left(1-\tfrac{\lambda_n g}{g-1}\right) +$$

$$\sum_{\substack{\forall x_i \in \{0,1\} \\ i=1..n-1}} \prod_{i=1}^{n-1} \left[\left(1-\tfrac{\lambda_i g}{g-1}\right)(1-x_i)+\tfrac{\lambda_i g}{g-1}x_i\right]\tfrac{\lambda_n g}{g-1} =$$

$$\sum_{\substack{\forall x_i \in \{0,1\} \\ i=1..n-1}} \prod_{i=1}^{n-1} \left[\left(1-\tfrac{\lambda_i g}{g-1}\right)(1-x_i)+\tfrac{\lambda_i g}{g-1}x_i\right]$$

Eliminating this way the rest of the variable x_i, $i = 1, \ldots, n-1$, we conclude that

$$\sum_{\substack{\forall x_i \in \{0,1\} \\ i=1..n}} \prod_{i=1}^{n} \left[\left(1-\frac{\lambda_i g}{g-1}\right)(1-x_i)+\frac{\lambda_i g}{g-1}x_i\right] = 1 \tag{3.14}$$

2. *The total multiplying factor of $\vec{p_l}$, ($l = 1, \ldots, n$) equals to λ_l.*

$$\sum_{\substack{\forall x_i \in \{0,1\} \\ i=1..n}} \prod_{i=1}^{n} \left[\left(1-\tfrac{\lambda_i g}{g-1}\right)(1-x_i)+\tfrac{\lambda_i g}{g-1}x_i\right] x_l \tfrac{g-1}{g} =$$

$$\sum_{\substack{\forall x_i \in \{0,1\} \\ i=1..n, i\neq l \\ x_l = 0}} \prod_{i=1}^{n} \left[\left(1-\tfrac{\lambda_i g}{g-1}\right)(1-x_i)+\tfrac{\lambda_i g}{g-1}x_i\right] x_l \tfrac{g-1}{g} +$$

$$\sum_{\substack{\forall x_i \in \{0,1\} \\ i=1..n, i\neq l \\ x_l = 1}} \prod_{i=1}^{n} \left[\left(1-\tfrac{\lambda_i g}{g-1}\right)(1-x_i)+\tfrac{\lambda_i g}{g-1}x_i\right] x_l \tfrac{g-1}{g} =$$

$$0 + \sum_{\substack{\forall x_i \in \{0,1\} \\ i=1..n, i\neq l}} \prod_{i=1..n, i\neq l} \left[\left(1-\tfrac{\lambda_i g}{g-1}\right)(1-x_i)+\tfrac{\lambda_i g}{g-1}x_i\right]\tfrac{\lambda_l g}{g-1}\tfrac{g-1}{g} \overset{(3.14)}{=} \lambda_l$$

From (3.13), (3.14), we conclude that iteration $\vec{j}$ can be expressed in respect to vertices $\vec{c}$ by a formula of the type (C.1). Thus, if all vertices $\vec{c}$ belong to J^n, then iteration $\vec{j}$ of this tile also belongs to J^n

$\dashv$

Example 3.3: In order to scan the tiles enumerated by the code produced in Example 3.1, we may use the 14 remaining inequalities of the system (3.2). Otherwise, we may use a combination of systems

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \\ -1 & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} j_1 \\ j_2 \end{pmatrix} \leq \begin{pmatrix} 39 \\ 29 \\ 0 \\ 0 \end{pmatrix}$$

corresponding to formula (3.10) and

$$\begin{pmatrix} 4 & -2 \\ -1 & 3 \\ -4 & 2 \\ 1 & -3 \end{pmatrix} \begin{pmatrix} j_1 - j_{01} \\ j_2 - j_{02} \end{pmatrix} \leq \begin{pmatrix} 19 \\ 19 \\ 0 \\ 0 \end{pmatrix}$$

corresponding to formula (3.11). The former system of inequalities has already the required form and need not be converted through a Fourier-Motzkin elimination. An application of the Fourier-Motzkin elimination method to the latter system of inequalities results to the equivalent system:

$$\begin{pmatrix} 1 & 0 \\ -1 & 0 \\ -1 & 3 \\ -2 & 1 \\ 2 & -1 \\ 1 & -3 \end{pmatrix} \begin{pmatrix} j_1 - j_{01} \\ j_2 - j_{02} \end{pmatrix} \leq \begin{pmatrix} 9 \\ 0 \\ 19 \\ 0 \\ 9 \\ 0 \end{pmatrix}$$

Note that this way only $4 + 6 = 10$ inequalities should be checked for each iteration, instead of 14, as deduced from formula (3.1) in Example 3.1. In sequel, one can fill in the missing part of the code produced in Example 3.1, according to the systems of inequalities described just above.

```
for(j1^S=-3; j1^S <=7; j1^S++)
  for(j2^S=max(⌈(-4-j1^S)/4⌉, ⌈(-4-3j1^S)/2⌉); j2^S <=min(⌊(14-j1^S)/4⌋,⌊(19-3j1^S)/2⌋); j2^S++){
    /* Execute tile (j1^S, j2^S) */
    j01=6j1^S+4j2^S; /* Calculate j0 = P j^S */
    j02=2j1^S+8j2^S;
    for(j1=max(0, j01); j1 <=min(39, j01+9); j1++)
      for(j2=max(0, j02-9+2(j1 - j01), j02+⌈(j1-j01)/3⌉);
            j2 <=min(29, j02+⌊(19+(j1-j01))/3⌋, j02+2(j1 - j01)); j2++){
        /* Execute iteration (j1, j2) */
        A[j1, j2]=A[j1 - 1, j2 - 2]+A[j1 - 3, j2 - 1];
    }
  }
```

A reduction of the run time can be achieved by distinguishing the tiles into internal and boundary, according to Lemma 3.2. Such a discrimination implies a check whether all vertices of the tile belong to J^n. This check is necessary to be conducted once for all 2^n vertices of each tile, while without this discrimination, the iteration space boundaries are checked once for each iteration. Thus, the above code segment can be rewritten as follows;

```
for(j_1^S=-3; j_1^S ≤7; j_1^S++)
  for(j_2^S=max(⌈(-4-j_1^S)/4⌉, ⌈(-4-3j_1^S)/2⌉); j_2^S ≤min(⌊(14-j_1^S)/4⌋,⌊(19-3j_1^S)/2⌋); j_2^S++){
    /* Execute tile (j_1^S, j_2^S) */
    j_01=6j_1^S+4j_2^S; /* Calculate j⃗_0 = P j⃗^S */
    j_02=2j_1^S+8j_2^S;
    /* Check whether tile (j_1^S, j_2^S) crosses the iteration space */
    /* boundaries */
    check=TILE_IN;
    for(x_1=0; x_1 ≤1; x_1++)
      for(x_2=0; x_2 ≤1; x_2++){
        /* Calculate vertex c⃗ = j⃗_0 + Σ_{i=1}^{n} x_i p⃗_i for all x_i ∈ {0,1} */
        c_1=j_01+6x_1+4x_2;
        c_2=j_02+2x_1+8x_2;
        /* Check whether c⃗ ∈ J^n */
        if(c_1<0 || c_1>39 || c_2<0 || c_2>29){
          check=TILE_CROSS;
          break;
        }
        if(check==TILE_CROSS) break;
      }
    if(check==TILE_CROSS) {
       /* Execute tile (j_1^S, j_2^S) in case it may cross */
       /* the iteration space boundaries */
      for(j_1=max(0, j_01); j_1 ≤min(39, j_01+9); j_1++)
        for(j_2=max(0, j_02-9+2(j_1 - j_01), j_02+⌈(j_1-j_01)/3⌉);
            j_2 ≤min(29, j_02+⌊(19+(j_1-j_01))/3⌋, j_02+2(j_1 - j_01)); j_2++){
          /* Execute iteration (j_1, j_2) */
          A[j_1, j_2]=A[j_1 - 1, j_2 - 2]+A[j_1 - 3, j_2 - 1];
        }
    }
    else {
       /* Execute tile (j_1^S, j_2^S) in case it does not cross */
       /* the iteration space boundaries */
      for(j_1=j_01; j_1 ≤ j_01+9; j_1++)
        for(j_2=max(j_02-9+2(j_1 - j_01), j_02+⌈(j_1-j_01)/3⌉);
            j_2 ≤min(j_02+⌊(19+(j_1-j_01))/3⌋, j_02+2(j_1 - j_01)); j_2++){
          /* Execute iteration (j_1, j_2) */
          A[j_1, j_2]=A[j_1 - 1, j_2 - 2]+A[j_1 - 3, j_2 - 1];
        }
    }
  }
```

Note that the generation of the above code segment is completely automated, when the initial iteration space and the tiling transformation are given. In addition, the loop bounds generated in this example for the n innermost loop indices can be also combined with the loop bounds generated for the n outermost loop indices in Example 3.2.

Reducing the run time overhead of tiling

In order to achieve a reduced run time complexity of the code generated automatically, as seen in Example 3.3, one should reduce the complexity of the loop bounds, which are checked for all tiles. That is, one should reduce the complexity of inequalities generated from formula (3.11). It is achieved by applying a linear transformation to the initial iteration space, so as to transform non-rectangular tiles into rectangular ones.

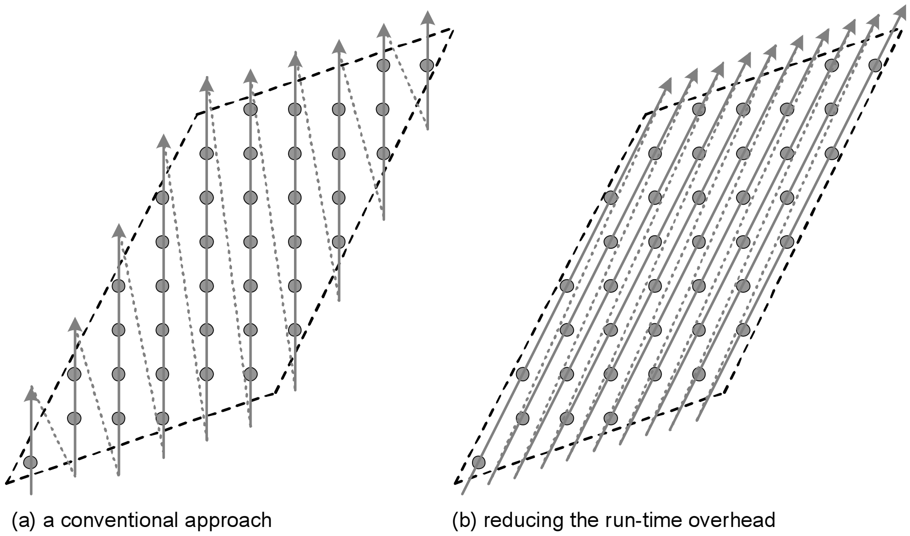


Figure 3.6: Scanning the iterations of a tile.
(a) Iterations of a tile are executed according to their lexicographic order, parallely to the axes.
(b) Iterations are scanned in such an order that traces to be parallel to the tile edges.

The method is based on the use of a non-unimodular transformation. The final goal is to traverse the TIS and then slide the points of TIS properly, so as to scan all points of J^n. In order to achieve this, the TIS is transformed to a rectangular space, called the transformed tile iteration space ($TTIS$). The $TTIS$ is traversed with an n-dimensional nested loop and then the indices of the loop are transformed, so as to return to the proper points of the TIS.

In other words, there is needed a transformation pair (P', H'): $TTIS \xrightarrow{P'} TIS$ and $TIS \xrightarrow{H'} TTIS$ (Fig. 3.7). Intuitively, P' should be parallel to the tile sides, that is, the column vectors of P' should be parallel to the column vectors of P. This is equivalent to the row vectors of H' being parallel to the row vectors of H. In addition to this, we demand the lattice of H' to be an integer space for integer loop indices to be able to traverse it. Formally, an n-dimensional transformation $H' : H' = VH$ must be found, where V is an $n \times n$ diagonal matrix and $\mathcal{L}(H') \subseteq Z^n$. The following lemma proves that the second requirement is satisfied if and only if H' is integral.

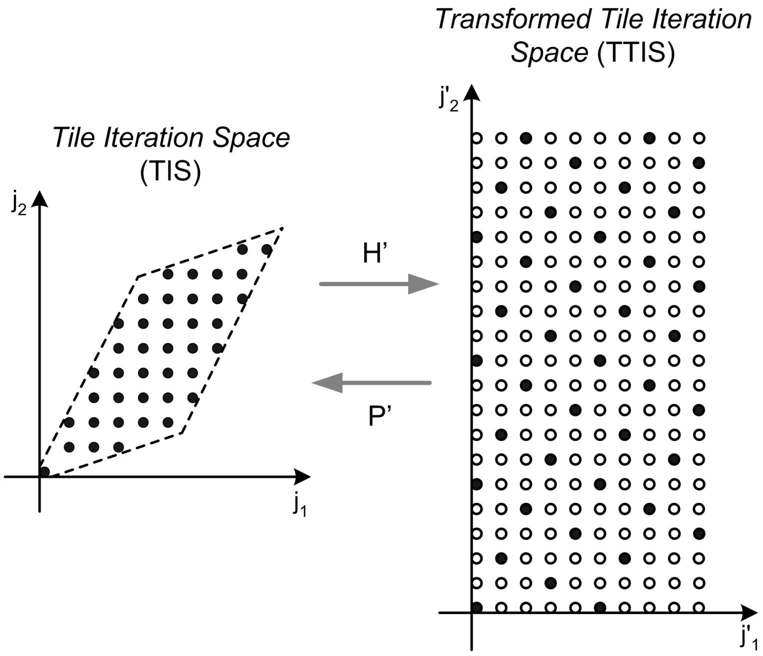


Figure 3.7: Traverse the TIS with a non-unimodular transformation.
In order to traverse the tile iteration space parallely to the tile edges, as indicated in Figure 3.6(b), the non-rectangular tile iteration space should be transformed into a rectangular one, using a non-unimodular transformation matrix H'. Since H' is not unimodular, the transformed space may include integer points with no integer coefficient in the initial space. They are depicted by white dots.

Lemma 3.3 *$\vec{j'} = A\vec{j} \in Z^n \ \forall \vec{j} \in Z^n$ iff A is integral.*

Proof: *If A is integral, it is clear that $\vec{j'} \in Z^n \forall \vec{j} \in Z^n$.*
Suppose that $\vec{j'} \in Z^n \forall \vec{j} \in Z^n$. We shall prove that A is integral:
It holds $\vec{j'} \in Z^n$ for $\vec{j} = \hat{u}_k$, where $\hat{u}_k$ is the k-th unitary vector,

$$\hat{u}_k = (u_{k1}, \ldots, u_{kn}), u_{kk} = 1, u_{ki} = 0, i \neq k$$

Thus,

$$\vec{j'} = A\hat{u}_k = \left(\sum_{i=1}^{n} a_{1i}u_{ki}, \sum_{i=1}^{n} a_{2i}u_{ki}, \ldots, \sum_{i=1}^{n} a_{ni}u_{ki} \right)^T = [a_{1k}, a_{2k}, \ldots, a_{nk}]^T \in Z^n$$

This holds for all $\hat{u}_k$, $k = 1 \ldots n$, therefore all elements of A are integer numbers. ⊣

Let us construct V in the following way: Every diagonal element v_{kk} is the smallest integer such that $v_{kk}\vec{h_k}$ is integral, where $\vec{h_k}$ is the k-th row of matrix H. Thus, both requirements for H' are satisfied. It is obvious that H' is a non-unimodular transformation. This means that the transformed tile iteration space contains holes. In Figure 3.7, the holes in the $TTIS$ are depicted with white dots, while the actual points are depicted with black ones. So, in order to traverse the TIS, we have to scan all actual points of the $TTIS$ and then transform them back using matrix P'. We can apply any of the methods presented in [Ram92], [Ram95], [Xue94], [Li93], [FLV95] to traverse the $TTIS$. However, we will avoid the application of Fourier-Motzkin elimination method by taking advantage of the tile shape regularity.

We use an n-dimensional nested loop with iterations indexed by $\vec{j'} = (j'_1, j'_2, \ldots, j'_n)$, in order to traverse the actual points of the $TTIS$. Replacing $\vec{j} = P'\vec{j'}$ in formula (2.7), the boundaries of TTIS are given by the system of inequalities: $0 \le HP'\vec{j'} < 1 \Leftrightarrow 0 \le V^{-1}\vec{j'} < 1 \Leftrightarrow$

$$0 \le j'_k \le v_{kk} - 1, \text{ for all } k = 1, \ldots, n \tag{3.15}$$

The bounds of the indices j'_k are determined by formulas (3.15), without applying the Fourier-Motzkin elimination method to the system of inequalities (3.11).

However, the increment step c_k of an index j'_k is not necessarily 1. In addition to this, if index j'_k is incremented by c_k, indices $j'_{k+1}, \ldots, j'_n$ should not be initialized at 0. Suppose that for a certain index vector $\vec{j'}$, it holds $P'\vec{j'} \in Z^n$. The first question is how much to increment the innermost index j'_n so that the next swept point is also integral. Formally, we search the minimum $c_n \in Z$ such that $P' \begin{pmatrix} j'_1 & j'_2 & \ldots & j'_n + c_n \end{pmatrix}^T \in Z^n$. After determining c_n, the next step is to calculate the increment step of index j'_{n-1} so that the next swept point is also integral. In this case, it is possible that index j'_n should also be incremented by an offset $a_{n(n-1)} : 0 \le a_{n(n-1)} < c_n$. In the general case of index j'_k we need to determine $c_k, a_{(k+1)k}, \ldots, a_{nk}$ such that: $P' \begin{pmatrix} j'_1 & \ldots & j'_k + c_k & j'_{k+1} + a_{(k+1)k} & \ldots & j'_n + a_{nk} \end{pmatrix}^T \in Z^n$. Every index j'_k has $k-1$ different incremental offsets a_{ki}, depending on each of the increment steps c_i of the $k-1$ outer indices j'_i. These offsets are $a_{k1}, \ldots, a_{k(k-1)}$. The following lemma proves that increment steps c_k and offsets a_{kl}, ($k = 1 \ldots n$ and $l = 1 \ldots k-1$), are directly obtained from the hermite normal form of matrix H', denoted $\widetilde{H'}$.

Lemma 3.4 *If $\widetilde{H'}$ is the column HNF of H' and $\vec{j'} = (j'_1, j'_2, \ldots, j'_n)$ is the index vector used to traverse the actual points of $\mathcal{L}(H')$, then the increment step (stride) for index j'_k is $c_k = \widetilde{h'}_{kk}$ and the incremental offsets are $a_{kl} = \widetilde{h'}_{kl}$, ($k = 1 \ldots n$ and $l = 1 \ldots k-1$).*

Proof: *It holds $\mathcal{L}(H') = \mathcal{L}(\widetilde{H'})$. Thus, $\vec{0} \in \mathcal{L}(H')$ and the columns of $\widetilde{H'}$ belong to $\mathcal{L}(H')$. Suppose $\vec{x} \in Z^n/\{\vec{0}\}$ with the following properties: $x_i = 0$ for $i < k$ and $0 \le x_i \le \widetilde{h'}_{ik}$ for $k \le i \le n$. It suffices to prove that $\vec{x} = \vec{h_k}$.*

Suppose that $\vec{x} \in \mathcal{L}(H')$*, which means that* $\exists \vec{j} \in Z^n : \widetilde{H'}\vec{j} = \vec{x}$*.* $\widetilde{H'}$ *is a lower triangular non-negative matrix and thus it holds:* $x_1 = \widetilde{h'}_{11} j_1 = 0 \Rightarrow j_1 = 0$*. Similarly,* $j_i = 0$ *for* $i < k$*. In the sequel, it holds:* $x_k = \widetilde{h'}_{kk} j_k$*. According to the above, it holds:* $0 \leq x_k = \widetilde{h'}_{kk} j_k \leq \widetilde{h'}_{kk} \Rightarrow 0 \leq j_k \leq 1$*. In addition,* $0 \leq x_{k+1} = \widetilde{h'}_{(k+1)k} j_k + \widetilde{h'}_{(k+1)(k+1)} j_{k+1} \leq \widetilde{h'}_{(k+1)k}$*. Since* $\widetilde{h'}_{(k+1)(k+1)} > \widetilde{h'}_{(k+1)k} \Rightarrow j_{k+1} = 0$*. Similarly,* $j_i = 0$ *for* $i > k+1$*. Consequently, since* $\vec{x} \neq \vec{0}$*,* $\vec{x}$ *is the* $k-th$ *column of* $\widetilde{H'}$*.* $\dashv$

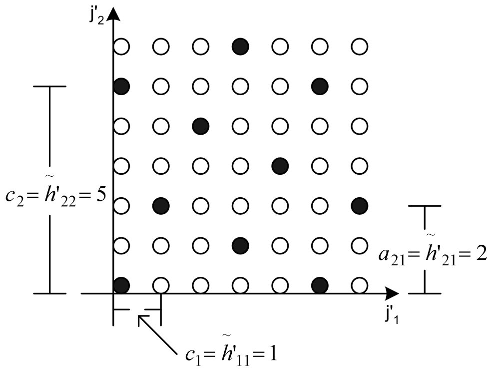


Figure 3.8: Steps and initial offsets in $TTIS$ derived from matrix $\widetilde{H'}$

According to the above analysis, the point that will be traversed using the next instantiation of indices is calculated from the current instantiation, since steps and incremental offsets are added to the current indices. Special care is taken so that every time the index vector $\vec{j'} = (j'_1, \ldots, j'_n)$ is to be modified, the new index vector $\vec{j'}$ is calculated as a sum of current $\vec{j'}$ and a multiple of a column-vector of $\widetilde{H'}$. Thus, assuming that the current instantiation $\vec{j'} \in \mathcal{L}(H')$, we ensure that the next point to be traversed remains in $\mathcal{L}(H')$.

Theorem 3.1 *The following* n*-dimensional nested loop traverses all points* $\vec{j'} \in TTIS$

for(j'_1=0, ..., j'_n=0; $j'_1 \leq v_{11}$-1; j'_1+=$\widetilde{h'}_{11}$, ..., j'_n+=$\widetilde{h'}_{n1}$)
  for($j'_n += \lceil \frac{-j'_2}{\widetilde{h'}_{22}} \rceil * \widetilde{h'}_{n2}, \ldots, j'_2 += \lceil \frac{-j'_2}{\widetilde{h'}_{22}} \rceil * \widetilde{h'}_{22}$; $j'_2 \leq v_{22} - 1$;
$j'_2 += \widetilde{h'}_{22}, \ldots, j'_n += \widetilde{h'}_{n2}$)
    ...
    for($j'_n += \lceil \frac{-j'_n}{\widetilde{h'}_{nn}} \rceil * \widetilde{h'}_{nn}$; $j'_n \leq v_{nn} - 1$; $j'_n += \widetilde{h'}_{nn}$){
      Loop body
    }

We now need to adjust the above loop, which sweeps all points in $TTIS$, in order to traverse the internal points of any tile in J^S. If $\vec{j'} \in TTIS$ is the point that is derived from the indices of the former loop and $\vec{j^S} \in J^S$ is the tile, whose internal points $\vec{j} \in J^n$ we want to traverse, it will hold: $\vec{j} = \vec{j_0} + P'\vec{j'} = P\vec{j^S} + P'\vec{j'}$, where $\vec{j_0} = P\vec{j^S} \in TOS$ is the tile origin, and $P'\vec{j'} \in TIS$ is the corresponding to $\vec{j'}$ point in TIS. Since $P = VP'$, the last equality can be equivalently rewritten as follows:

$$\vec{j} = P'(V\vec{j^S} + \vec{j'}) \tag{3.16}$$

Special attention also needs to be paid so that the points traversed do not overcome the original space boundaries. As we have mentioned before, a point $\vec{j} \in J^n$ satisfies the following set of inequalities: $B\vec{j} \leq \vec{b}$. Replacing $\vec{j}$ by the above equation (3.16), we have:

$$BP'(V\vec{j^S} + \vec{j'}) \leq \vec{b} \tag{3.17}$$

By applying the Fourier-Motzkin elimination method to this set of inequalities, we obtain proper expressions for $\vec{j'}$, so that we do not cross the original space boundaries. As deduced for systems (3.10), (3.11), system (3.17) should be used in combination with inequalities (3.15).

Example 3.4: Let us consider the same algorithm as in the previous examples. We will now sweep the internal points of tiles with the use of the method described just above. We need the following matrices: $H' = \begin{bmatrix} 2 & -1 \\ -1 & 3 \end{bmatrix}$ and $V = \begin{bmatrix} 10 & 0 \\ 0 & 20 \end{bmatrix}$. Accordingly, $P' = \begin{bmatrix} \frac{3}{5} & \frac{1}{5} \\ \frac{1}{5} & \frac{2}{5} \end{bmatrix}$. The Hermite Normal Form of matrix H' is $\widetilde{H'} = \begin{bmatrix} 1 & 0 \\ 2 & 5 \end{bmatrix} = \begin{bmatrix} 2 & -1 \\ -1 & 3 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 2 \end{bmatrix}$ and thus, as shown in Figure 3.8, $c_1 = \widetilde{h'}_{11} = 1$, $c_2 = \widetilde{h'}_{22} = 5$, $a_{21} = \widetilde{h'}_{21} = 2$. Consequently, the code that traverses the indices inside every internal tile, according to Theorem 3.1, is:

```
/* Calculate j'_0 = V j^S */
j'_01=10 j^S_1;
j'_02=20 j^S_2;
for (j'_1 = 0, j'_2 = 0; j'_1 <= 9; j'_1 += 1, j'_2 += 2)
   for (j'_2 += ⌈-j'_2/5⌉ * 5; j'_2 <= 19; j'_2 += 5) {
      /* Calculate j = P'(V j^S + j') */
      j1=3/5(j'_01 + j'_1)+1/5(j'_02 + j'_2);
      j2=1/5(j'_01 + j'_1)+2/5(j'_02 + j'_2);
      /* Execute iteration (j1, j2) */
      A[j1, j2]=A[j1 - 1, j2 - 2]+A[j1 - 3, j2 - 1];
   }
```

In order to exactly scan the internal of boundary tiles, we construct matrix

$$[BP'|\vec{b}] = \left(\begin{array}{cc|c} \frac{3}{5} & \frac{1}{5} & 39 \\ \frac{1}{5} & \frac{2}{5} & 29 \\ -\frac{3}{5} & -\frac{1}{5} & 0 \\ -\frac{1}{5} & -\frac{2}{5} & 0 \end{array}\right)$$

The application of Fourier-Motzkin elimination method on this matrix gives:

$$\left(\begin{array}{rr|r} 1 & 0 & 78 \\ -1 & 0 & 29 \\ 3 & 1 & 195 \\ 1 & 2 & 145 \\ -3 & -1 & 0 \\ -1 & -2 & 0 \end{array}\right)$$

Consequently, the code that traverses the indices inside tiles, which cut the iteration space bounds, is:

```
/* Calculate $\vec{j'_0} = V\vec{j^S}$ */
$j'_{01}$=10$j^S_1$;
$j'_{02}$=20$j^S_2$;
$l'_1$ =max(0, $-29 - j'_{01}$);
$u'_1$ =min(9 /* $v_{11}$-1 */, $78 - j'_{01}$);
for ($j'_1 = l'_1, j'_2 = l'_1 * 2$; $j'_1 \le u'_1$; $j'_1 += 1, j'_2 += 2$) {
   $l'_2$ =max(0, $-3(j'_{01}+j'_1) - j'_{02}, \lceil\frac{-(j'_{01}+j'_1)}{2}\rceil - j'_{02}$);
   $u'_2$ =min(19 /* $v_{22}$-1 */, $195 - 3(j'_{01}+j'_1) - j'_{02}, \lfloor\frac{145-(j'_{01}+j'_1)}{2}\rfloor - j'_{02}$);
   for ($j'_2 += \lceil\frac{l'_2 - j'_2}{5}\rceil * 5$; $j'_2 \le' b_2$; $j'_2 += 5$) {
      /* Calculate $\vec{j} = P'(V\vec{j^S} + \vec{j'})$ */
      $j_1$=$\frac{3}{5}(j'_{01}+j'_1)$+$\frac{1}{5}(j'_{02}+j'_2)$;
      $j_1$=$\frac{1}{5}(j'_{01}+j'_1)$+$\frac{2}{5}(j'_{02}+j'_2)$;
      /* Execute iteration $(j_1, j_2)$ */
      A[$j_1, j_2$]=A[$j_1 - 1, j_2 - 2$]+A[$j_1 - 3, j_2 - 1$];
   }
}
```

Using the tile space boundaries calculated in Example 3.2, and combining the code segments produced just above for internal tiles and for tiles crossing the iteration space boundaries, we get the final code segment:

```
for ($j^S_1 = -4$; $j^S_1 \le 8$; $j^S_1$++)
   for ($j^S_2$ =max($\lceil\frac{-4-3j^S_1}{2}\rceil, \lceil\frac{-4-j^S_1}{4}\rceil$); $j^S_2$ $\le$min($\lfloor\frac{19-3j^S_1}{2}\rfloor, \lfloor\frac{14-j^S_1}{4}\rfloor$); $j^S_2$++) {
      /* Execute tile $(j^S_1, j^S_2)$ */
      /* Calculate $\vec{j'_0} = V\vec{j^S}$ */
      $j'_{01}$=10$j^S_1$; /* This line could be placed outside loop $j^S_2$ */
      $j'_{02}$=20$j^S_2$;

      /* Check whether tile $(j^S_1, j^S_2)$ crosses the iteration space */
      /* boundaries */
      check=TILE_IN;
      for($x_1$=0; $x_1$ $\le$1; $x_1$++){
         $c_1$=$j'_{01}$+9$x_1$;
```

```
    if($c_1$<-29 || $c_1$>78) { check=TILE_CROSS; break; }
    for($x_2$=0; $x_2$ ≤1; $x_2$++){
      $c_2$=$j'_{02}$+19$x_2$;
      /* Check whether $\vec{c} \in J^n$ */
      if($c_2$<max(-3$c_1$, $\lceil\frac{-c_1}{2}\rceil$) || $c_2$>min(195-3$c_1$, $\lfloor\frac{145-c_1}{2}\rfloor$)){
        check=TILE_CROSS; break;
      }
      if(check==TILE_CROSS) break;
    }
  }

  if(check==TILE_CROSS) {
     /* Execute tile $(j_1^S, j_2^S)$ in case it may cross */
     /* the iteration space boundaries */
    $l'_1$ =max$(0, -29 - j'_{01})$;
    $u'_1$ =min(9 /* $v_{11}$-1 */, $78 - j'_{01}$);
    for ($j'_1 = l'_1, j'_2 = l'_1 * 2$; $j'_1 \le u'_1$; $j'_1 += 1, j'_2 += 2$) {
      $l'_2$ =max$(0, -3(j'_{01} + j'_1) - j'_{02}, \lceil\frac{-(j'_{01}+j'_1)}{2}\rceil - j'_{02})$;
      $u'_2$ =min(19 /* $v_{22}$-1 */, $195 - 3(j'_{01} + j'_1) - j'_{02}, \lfloor\frac{145-(j'_{01}+j'_1)}{2}\rfloor - j'_{02}$);
      for ($j'_2 += \lceil\frac{l'_2 - j'_2}{5}\rceil * 5$; $j'_2 \le' b_2$; $j'_2 += 5$) {
        /* Calculate $\vec{j} = P'(V\vec{j}^S + \vec{j}')$ */
        $j_1$=$\frac{3}{5}(j'_{01} + j'_1)$+$\frac{1}{5}(j'_{02} + j'_2)$;
        $j_1$=$\frac{1}{5}(j'_{01} + j'_1)$+$\frac{2}{5}(j'_{02} + j'_2)$;
        /* Execute iteration $(j_1, j_2)$ */
        A$[j_1, j_2]$=A$[j_1 - 1, j_2 - 2]$+A$[j_1 - 3, j_2 - 1]$;
      }
    }
  }

  else {
    for ($j'_1 = 0, j'_2 = 0$; $j'_1 \le 9$; $j'_1 += 1, j'_2 += 2$)
      for ($j'_2 += \lceil\frac{-j'_2}{5}\rceil * 5$; $j'_2 \le 19$; $j'_2 += 5$) {
        /* Calculate $\vec{j} = P'(V\vec{j}^S + \vec{j}')$ */
        $j_1$=$\frac{3}{5}(j'_{01} + j'_1)$+$\frac{1}{5}(j'_{02} + j'_2)$;
        $j_1$=$\frac{1}{5}(j'_{01} + j'_1)$+$\frac{2}{5}(j'_{02} + j'_2)$;
        /* Execute iteration $(j_1, j_2)$ */
        A$[j_1, j_2]$=A$[j_1 - 1, j_2 - 2]$+A$[j_1 - 3, j_2 - 1]$;
      }
  }

}
```

3.2.3 Comparison – Experimental Results

Both our method (in the sequel denoted as RI - Reduced Inequalities) and the one described in [AI91] by Ancourt and Irigoin (denoted as AI), have been implemented as a software tool which automatically generates tiled C code using any tiling transformation P. In this section, we compare AI and RI methods both in terms of compilation time and generated code efficiency. We generated several random $2-D$ and $3-D$ problems and measured the following: compilation time, row operations performed by Fourier-Motzkin elimination and run time of the generated code. In the sequel, we applied both AI and RI methods to three real applications: SOR, Jacobi and ADI integration. We also applied the inequalities of AI method to the Omega calculator [KMP+95] and generated code for all problems. We then measured the compilation time and run time obtained by Omega (the results are denoted as AI-Omega) and compared them with the ones obtained by AI (using our tool) and RI. Table 3.1 shows the iteration spaces used as examples in $2-D$ and $3-D$ problems. We applied several tiling transformations, in which the non-zero elements of the tiling matrices were randomly generated. In $2-D$ spaces we applied three different tiling transformations (P_1, P_2, P_3) varying from the diagonal matrix P_1 to more complex ones. In $3-D$ spaces we applied seven different tiling transformations $(P_4, \ldots, P_{10})$, again here starting from the diagonal P_4 and adding non-zero elements (P_{10} contains no zero element). We performed our experiments on a PIII @ 800MHz processor with 128MB of RAM. The operating system is Linux with kernel 2.4.18. The generated tiled code was compiled using gcc v.2.95.4 with the -O3 optimization flag. We also experimented with lower optimization levels, where the execution times were slower, but the relative results for all methods remained the same.

Table 3.1: Example iteration spaces

	j_1		j_2		j_3		
	lower bound	**upper bound**	**lower bound**	**upper bound**	**lower bound**	**upper bound**	**# of iterations**
Space1	-1999	4999	-1999	4999	-	-	48986001
Space2	-1999	4999	-1999	$4999 + 2i_1$	-	-	69983001
Space3	-4999	4999	$-4999 + 3i_1$	$4999 + 2i_1$	-	-	99980001
Space4	0	399	0	399	0	399	64000000
Space5	0	399	0	$399 + i_1$	0	399	95920000
Space6	0	399	$-i_1$	$399 + i_1$	0	399	127840000
Space7	-99	149	$-99 - i_1$	$149 + i_1$	-99	$149 + 2i_2$	22904099
Space8	0	399	$-i_1$	$399 + i_1$	i_1	$79 + 2i_2$	117635018
Space9	-99	149	$-99 - i_1$	$149 + i_1$	$-99 - i_1$	$149 + i_1 + 2i_2$	31129399
Space10	0	59	$-i_1$	$59 + i_1$	$-i_1 - 3i_2$	$59 + i_1 + 2i_2$	1994462

Row Operations - Compilation Time

Tables 3.2-3.4 summarize the results (row operations and compilation time) from the compilations of all iteration spaces tiled with all candidate tiling matrices. We present here the number

Table 3.2: Fourier-Motzkin row operations and compilation time for 2D algorithms

		AI	RI	AI-Omega	AI	RI
		Row Operations		*Compilation Time (ms)*		
P_1	**Space1**	30	10	16.29	0.26	0.26
	Space2	30	10	19.53	0.27	0.26
	Space3	34	10	20.82	0.29	0.26
P_2	**Space1**	37	10	22.56	0.28	0.27
	Space2	33	10	21.56	0.28	0.27
	Space3	34	10	22.78	0.29	0.26
P_3	**Space1**	56	12	33.36	0.36	0.30
	Space2	55	12	39.40	0.37	0.30
	Space3	53	12	40.12	0.36	0.30
		Avg. Row Operations		*Avg. Compilation Time (ms)*		
P_1		31	10	18.88	0.27	0.26
P_2		35	10	22.30	0.28	0.27
P_3		55	12	37.63	0.36	0.3

of row operations and compilation times of each matrix for each iteration space and the average values of each matrix for all iteration spaces.

Run Time

In order to evaluate the run time overhead due to tiling, we executed all tiled codes of the previous problems and measured their run time. We also executed the original untiled serial code for each problem. We define the *tiling overhead factor (TOF)* as the fraction of the run time of the sequential tiled code to the run time of the untiled code: $TOF = \frac{\text{Run time of Sequential Tiled Code}}{\text{Run time of Untiled Code}}$. Note that, the loop body in each case is a simple array assignment statement and, thus, the run time measured is dominated by the time to compute the loop bounds. Since the array size was small (20×20) and the tile sizes were not chosen to be optimal for cache locality, the sequential tiled code does not present any improvement due to the exploitation of the memory hierarchy. Thus, TOF indicates the overhead imposed by the evaluation of the new loop bounds, due to tiling. If TOF is too large, it will aggravate the speedup obtained when we parallelize nested `for`-loops using tiling. Tables 3.5-3.6 summarize the tiling overhead factors. Again here we present the TOFs of all tiling matrices applied to each iteration space and the average TOFs of all matrices P across all iteration spaces. Figure 3.9 shows the TOF of $3-D$ problems as a function of the number of non-zero elements in tiling matrix P.

Real Applications

In our last set of experiments, we applied AI and RI methods to tile three real applications: SOR, Jacobi, and ADI integration. For the first two problems, there is a skewed and an unskewed version, and for each version there are four (communication and scheduling) optimal matrices as described in [HS02] and [Xue97a]. Table 3.7 summarizes the row operations, compilation times

Table 3.3: Fourier-Motzkin row operations and compilation time for 3D algorithms. In some cases the Fourier-Motzkin elimination method could not be completed in a reasonable time, or was interrupted due to lack of memory or an overflow exception. In these cases, we have denoted a – in the respective cells of the table.

		Row Operations		*Compilation Time (ms)*		
		AI	**RI**	**AI-Omega**	**AI**	**RI**
P_4	**Space4**	70	22	27	0.41	0.43
	Space5	70	22	30.7	0.42	0.43
	Space6	74	22	33.39	0.44	0.43
	Space7	80	22	42.5	0.49	0.44
	Space8	117	22	84.14	0.62	0.44
	Space9	87	20	55.8	0.53	0.43
	Space10	116	22	89.54	0.63	0.44
P_5	**Space4**	82	22	36.3	0.45	0.44
	Space5	86	22	45.58	0.48	0.43
	Space6	96	22	51	0.53	0.43
	Space7	95	22	51.52	0.55	0.44
	Space8	150	22	158.12	0.79	0.45
	Space9	110	20	55.56	0.62	0.43
	Space10	118	22	70.35	0.65	0.45
P_6	**Space4**	132	28	106	0.64	0.48
	Space5	159	34	167.63	0.77	0.51
	Space6	220	42	371.34	1.1	0.54
	Space7	199	38	213.76	1.03	0.54
	Space8	470	42	397.13	3.91	0.54
	Space9	316	38	284.81	1.91	0.54
	Space10	360	42	382.33	2.32	0.55
P_7	**Space4**	264	28	235.55	1.33	0.49
	Space5	578	34	367.78	6.0	0.52
	Space6	508	42	1,188.72	4.24	0.55
	Space7	1411	38	911.38	40.78	0.54
	Space8	1522	42	2,099.32	51.31	0.56
	Space9	379	38	370.47	2.61	0.55
	Space10	419	42	527.3	3.08	0.56
P_8	**Space4**	4,254	28	1,558.04	460.04	0.51
	Space5	14,012	34	2,891.19	7,607.2	0.52
	Space6	10,049	38	4,019.51	3,022.46	0.54
	Space7	1,752	36	1,846.78	73.16	0.54
	Space8	6,031	40	3,201.75	1,040.44	0.55
	Space9	637	36	3,889.58	7.27	0.54
	Space10	936	40	–	15.95	0.55
P_9	**Space4**	6,933	46	1,984.67	1,280.34	0.56
	Space5	10,569	42	2,775.25	3,234.86	0.56
	Space6	5,655	40	3,662.66	855.78	0.55
	Space7	751	40	5,132.84	9.77	0.55
	Space8	1,907	36	1,943.71	83.53	0.54
	Space9	259	22	2,308.23	1.37	0.51
	Space10	295	22	2,640.29	1.65	0.49
P_{10}	**Space4**	6,477	46	1,629.59	1,034.07	0.58
	Space5	27,763	44	2,612.24	45,342.36	0.56
	Space6	12,533	40	2,484.32	5,351.28	0.55
	Space7	95,712	40	2,428.64	638,417.48	0.56
	Space8	83,025	40	1,014.64	450,599.44	0.56
	Space9	71,119	40	3,215.22	328,971.3	0.57
	Space10	> 120,309	40	4,336.41	> 1,025,846.41	0.57

Table 3.4: Average row operations and compilation time for 3D algorithms

	Avg. Row Operations		*Avg. Compilation Time (ms)*		
	AI	**RI**	**AI-Omega**	**AI**	**RI**
P_4	88	22	51.87	0.51	0.43
P_5	105	22	67.2	0.58	0.44
P_6	265	38	276.14	1.67	0.53
P_7	726	38	814.36	15.62	0.54
P_8	5382	36	2,901.14	1,746.64	0.53
P_9	3767	35	2,921.1	781.04	0.53
P_{10}	59563	41	2,531.58	356,508.91	0.56

Table 3.5: Tiling overhead factors (TOF) for $2-D$ problems

		TOF (2D)			*Avg. TOF (2D)*		
		AI-Omega	**AI**	**RI**	**AI-Omega**	**AI**	**RI**
P_1	**Space1**	2.59	0.96	1.24	2.85	1.03	1.31
	Space2	2.73	1.01	1.27			
	Space3	3.22	1.13	1.43			
P_2	**Space1**	6.27	4.55	1.61	6.62	4.78	1.69
	Space2	6.12	4.62	1.63			
	Space3	7.45	5.16	1.82			
P_3	**Space1**	8.00	6.10	3.58	8.23	6.41	3.75
	Space2	7.75	6.21	3.63			
	Space3	8.95	6.92	4.04			

and TOFs for each case. Figure 3.10 shows the TOFs obtained by each method, in each case.

Overall Evaluation Comments

As far as compilation time is concerned, RI method clearly outperforms AI method. This is due to the fact that RI method feeds Fourier-Motzkin elimination with the system in (3.8), which consists of $2n$ inequalities with n variables, while AI method feeds Fourier-Motzkin elimination with the system in (3.1), which consists of $4n$ inequalities with $2n$ variables. Recall that Fourier-Motzkin elimination is a doubly exponential algorithm and thus the reduction in its input size

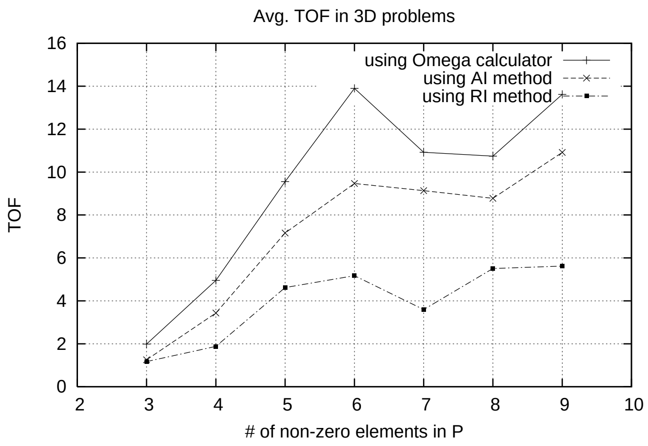


Figure 3.9: Average tiling overhead factors for $3-D$ problems

Table 3.6: Tiling overhead factors (TOF) for $3-D$ problems.
In case the compilation time could not be calculated in Table 3.3, then, the run time can not be calculated, either. These cases have been indicated by a – in the respective cells of this table.

		TOF (3D)			*Avg. TOF (3D)*		
		AI-Omega	**AI**	**RI**	**AI-Omega**	**AI**	**RI**
P_4	**Space4**	1.33	1.21	1.18	1.99	1.26	1.17
	Space5	1.36	1.23	1.17			
	Space6	1.39	1.23	1.17			
	Space7	2.19	1.21	1.11			
	Space8	2.45	1.16	1.19			
	Space9	2.44	1.28	1.10			
	Space10	2.75	1.48	1.30			
P_4	**Space4**	5.39	3.57	1.97	4.96	3.44	1.88
	Space5	5.57	3.59	1.98			
	Space6	5.72	3.63	1.91			
	Space7	4.33	3.20	1.77			
	Space8	4.57	3.20	1.85			
	Space9	4.51	3.33	1.77			
	Space10	4.61	3.53	1.90			
P_4	**Space4**	10.90	7.55	4.05	9.55	7.16	4.62
	Space5	10.77	7.52	4.38			
	Space6	11.17	7.65	4.51			
	Space7	8.33	6.67	4.13			
	Space8	8.44	6.68	4.01			
	Space9	8.52	6.89	4.61			
	Space10	8.75	7.18	6.67			
P_7	**Space4**	15.50	9.86	4.65	13.90	9.47	5.17
	Space5	16.09	10.05	5.14			
	Space6	16.20	10.10	5.29			
	Space7	12.67	9.04	4.80			
	Space8	12.72	8.92	4.65			
	Space9	11.80	8.95	4.84			
	Space10	12.29	9.38	6.84			
P_4	**Space4**	12.94	9.81	3.51	11.24	9.14	3.60
	Space5	12.40	9.88	3.61			
	Space6	12.27	9.92	3.68			
	Space7	9.87	8.39	3.29			
	Space8	10.08	8.36	3.16			
	Space9	9.87	8.60	3.48			
	Space10	–	8.98	4.46			
P_4	**Space4**	12.68	9.63	6.10	10.74	8.78	5.51
	Space5	12.52	9.61	6.05			
	Space6	12.68	9.75	6.09			
	Space7	9.21	7.96	5.05			
	Space8	9.75	7.89	4.39			
	Space9	9.51	8.15	4.57			
	Space10	8.86	8.46	6.33			
P_4	**Space4**	16.07	11.70	5.17	13.62	11.07	5.62
	Space5	16.55	11.75	5.04			
	Space6	16.24	11.57	5.09			
	Space7	12.30	10.48	5.11			
	Space8	11.20	10.14	3.83			
	Space9	11.26	10.77	5.67			
	Space10	11.72	–	9.44			

Table 3.7: Performance for real applications

		Row Operations		*Compilation Time (ms)*			*TOF*		
		AI	**RI**	**AI-Omega**	**AI**	**RI**	**AI-Omega**	**AI**	**RI**
SOR	P_1	99	22	53.03	0.50	0.42	1.47	1.20	1.05
	P_2	107	22	50.27	0.53	0.42	1.50	1.21	1.01
	P_3	118	22	49.01	0.57	0.42	1.75	1.63	1.05
	P_4	165	40	90.04	0.77	0.5	1.80	1.78	1.30
SOR skewed	P_1	99	22	42.09	0.53	0.41	1.59	1.29	1.06
	P_2	107	22	40.60	0.53	0.42	1.60	1.29	1.06
	P_3	118	22	57.9	0.57	0.42	1.90	1.73	1.12
	P_4	165	40	91.97	0.77	0.51	1.95	1.86	1.34
Jacobi	P_1	645	28	346.99	5.3	0.46	2.08	1.91	1.57
	P_2	645	28	347.96	5.26	0.47	2.09	1.92	1.60
	P_3	800	28	362.5	8.86	0.47	2.06	1.90	1.56
	P_4	3207	46	$1,353.55$	194.88	0.53	5.58	5.09	2.10
Jacobi skewed	P_1	645	28	251.885	4.93	0.48	1.99	1.88	1.44
	P_2	645	28	248.27	4.98	0.47	1.98	1.87	1.46
	P_3	800	28	229.34	8.19	0.48	2.02	1.89	1.45
	P_4	691	28	238.82	5.95	0.47	2.01	1.88	1.43
ADI	P_1	180	28	47.42	0.85	0.46	1.46	1.47	1.07

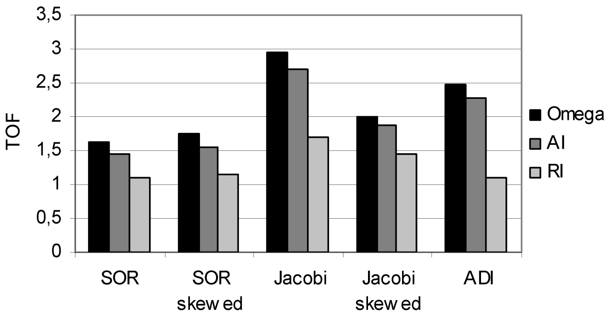


Figure 3.10: Tiling overhead factors for real applications

imposed by our method causes significant reduction in the method's execution steps, as clearly seen by the number of row operations. Note also that the exact simplification method of Fourier-Motzkin elimination was not applied in the presented experiments, since the gain in run time by the application of the method was inadequate to justify the vast increase in compilation times, especially in the case of AI method (3% average and 10% maximum gain in run time). In particular, while RI compilation times remained in the order of milliseconds when using exact simplification, AI compilation times increased dramatically (reached the order of an hour). This means that we can practically apply exact simplification to RI, in order to further improve the efficiency of the generated code.

Despite the reduction in compilation time imposed by RI, it seems that both AI and AI-Omega perform well in almost all $2-D$ and $3-D$ problems (compilation times are less than one second). However, in problems of larger dimensions, both AI and AI-Omega present several problems. We executed a number of randomly generated $4-D$ algorithms and observed that, at first, the compilation time of AI becomes impractical (several hours or even days). More importantly, AI failed to generate code for almost half of the problems due to lack of memory. Note that Fourier-Motzkin elimination is also doubly exponential in space, so in several $4-D$ problems even 1GB of virtual memory was not sufficient to cover the needs of the method. On the other hand, AI-Omega also faced some problems with memory space (to a smaller extent than AI) but here again, in almost half of the problems, the system rose an overflow exception. Apparently, after a large number of row operations in $4-D$ algorithms, some coefficients exceeded the system's `MAXINT`. In all cases RI method succeeded in generating code, within some seconds in the worst case.

Note, also, that, since we do not know all details about the implementation of Omega, we cannot be sure why the AI-Omega implementation gives higher implementation times than our implementation. However, as deduced by tiling matrices P_8, P_{10}, Omega is more stable and one can more accurately predict the time needed for the generation of serial tiled code.

As far as run time is concerned, RI also exhibits a significant improvement in performance in all problems. In particular, as shown in Figure 3.9, as the number of non-zero elements in matrix P increases, the improvement of RI method becomes much more obvious. This means that RI method performs very well in complex problems where the tiling matrices contain many non-zero elements and the iteration spaces are non-rectangular. In addition, as shown in Figure 3.10, RI's performance is nearly optimal in simpler algorithms such as SOR, Jacobi and ADI, since the TOF in these cases is very close to one. Thus, RI performs very well in easy problems and sustains a remarkably good performance even when the tiling transformations and the shape of the iteration spaces become increasingly complex.

The improvement in the quality of the generated code caused by RI, is due to the fact that, although the code to enumerate the tiles is essentially similar in AI and RI, the code to traverse the internal points of the tiles is completely different. Our tool makes a distinction between boundary and internal tiles and generates different code to scan the internal points for both AI and RI (as in Examples 3.3, 3.4). In the case of boundary tiles, RI method results in fewer inequalities for the bounds of the tile space. Consequently, fewer bound calculations are executed during run time. In the case of internal tiles, which are the vast majority in most problems, the code of RI consists of a loop with constant bounds $0 \leq j'_i \leq v_{ii}-1$ for $i = 1, \ldots, n$ (see formula (3.15)), while the code of AI includes a loop whose bounds are derived from the application of Fourier-Motzkin elimination to the system $\begin{pmatrix} gH \\ -gH \end{pmatrix} (\vec{j} - \vec{j_0}) \leq \begin{pmatrix} (g-1)\vec{1} \\ \vec{0} \end{pmatrix}$ (see formula (3.11)). It is clear that the calculation of loop bounds in the first case is much more efficient. Finally, note that the enumeration of some redundant tiles does not impose any significant overhead, since the number of redundant tiles is negligible. The same holds for the non-unimodular transformation used to access the internal points of the tiles. In this case, the additional operations due to the transformation are simple integer multiplications, while operations on extra variables are integer additions and assignment statements, which are all efficiently executed by modern processors and optimized by any back-end compiler like gcc.

Note, also, that, the run time overhead imposed by Omega, in comparison to our implementation for AI inequalities, is due to the fact that Omega is a general purpose code generation tool, while our implementation is aimed at tiled nested loops. Thus, Omega cannot take into account the discrimination of internal and boundary tiles, as in Examples 3.3, 3.4. It uses the system of inequalities (3.1) for both enumerating the tiles and scanning their interior. Although the optimization described in this section could not be incorporated into Omega, the respective columns have been used in Tables 3.2-3.7 as a measure of efficiency of the code produced by our tool.

Summarizing, the compilation time reduction is due to the method used to enumerate the tiles of the tile space, while the run time reduction is mainly due to the transformation of a non-rectangular tile to a rectangular one.

3.3 Parallelization

In this section, we refer to some parallelization aspects of the sequential tiled code. Recall from Figure 3.1 that the parallelization of an arbitrarily tiled algorithm involves two separate tasks: first, the generation of the sequential tiled code and, second, the parallelization of this code. §3.2 focused on the first task. This section will focus on the second one. Parallelization can be separated in sub-tasks such as iteration distribution, data distribution and data transferring code generation. Tang and Xue in [TX00] addressed the same issues for rectangularly tiled iteration spaces. In this section, as in [GDAK02a], [Gou03], efficient data parallel code for non-rectangular tiles will be discussed, without imposing any further complexity.

When executing an algorithm on a distributed memory machine, the original data space of the algorithm is distributed to the local memories of the processing nodes. The local data space of each node is in general a non-rectangular subset of the original data space, even if rectangular tiling is applied [AKN95]. However, applying the transformations proposed in §3.2.2, each processor can iterate over a rectangular local iteration space (TTIS) and access rectangular data spaces as well. In this way, each processor can allocate exactly the required amount of memory. Rectangular data spaces also allow for straightforward addressing schemes of array elements and thus a direct way of sweeping data by the generated code.

Another very important benefit in parallelization using rectangular local iteration spaces (TTIS) is the convenient determination of the communication sets. Each communication set contains the communication points, i.e. the points that are written in the local memory of a processing node and are needed by another. The communication points have the following property: if we add one dependence vector to them, then the resulting point lies in a tile assigned to a different node. Figure 3.11 shows the communication points and sets when determined in

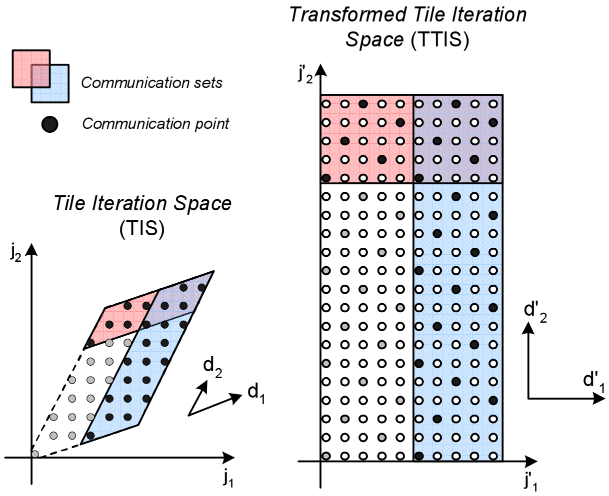


Figure 3.11: Determining communication sets in the TIS and TTIS.
In case the following tile along a dimension has been assigned to a different processing node, then the data calculated by the iterations of the corresponding grey area should be transferred to it.

the TIS and in the TTIS. $\vec{d_1}$ and $\vec{d_2}$ are the dependences of the original algorithm, while $\vec{d'_1}$ and $\vec{d'_2}$ are the transformed dependences in the $TTIS$. It is obvious that, when working with the rectangular $TTIS$, the communication sets are much more easily determined since they are rectangular as well. Note that these sets, indicated by grey areas should be transferred at run-time only in case the following tile is assigned to a different node, according to the allocation schemes that will be explored in detail in Chapters 4, 5.

3.3.1 Some more algorithmic assumptions

In addition to the restriction imposed by our algorithmic model in §2.2 and summarized in Appendix B, in this section we also consider that the body of the perfectly nested loops is consisted of a statement of the form:

$$A[f_w(\vec{j})] := F(A[f_w(\vec{j} - \vec{d_1})], \ldots, A[f_w(\vec{j} - \vec{d_q})]);$$

where:

1. $\vec{j} = (j_1, \ldots, j_n)$ is the current iteration
2. $\vec{d_i} = (d_{i1}, \ldots, d_{in})$, $i = 1, \ldots, q$ are the uniform and constant dependences of this code segment and
3. F, f_w are functions.

In order to simplify the model, single assignment statements with one array variable have been considered. Note, however, that this is only a notational restriction, since all of the techniques presented in this section can be adapted to multiple statements on multiple arrays. In addition to previously defined spaces, in this section we shall use the data space, denoted DS, defined as:

$$DS = \{f_w(\vec{j}) | \vec{j} \in J^n\}$$

where f_w is the write array reference.

The underlying architecture is considered a $(n-1)$-dimensional processor mesh. Thus, each processor is identified by a $(n-1)$-dimensional vector denoted $\vec{pid}$. Note, however, that this is not a physical restriction, but a convention for processor labelling. More generally, a bi-level parallel architecture may be considered as a $(n-1)$-dimensional mesh of SMP nodes (Symmetric Multi-Processors). Each SMP node is identified by a $(n-1)$-dimensional vector denoted $\vec{smp\_id}$. In addition, we consider that each SMP node is consisted of a $(n-1)$-dimensional mesh of CPUs (processors) with m_x CPUs along the x-th dimension. Each CPU is identified by a $(n-1)$-dimensional vector denoted $\vec{cpu\_id}$ $(0 \leq cpu\_id_x \leq m_x - 1)$. Apparently, there is an one-to-one correspondence between the global labels of processors and their labels inside a node. It holds that

$$pid_x = cpu\_id_x + smp\_id_x m_x$$

Inversely, it holds that

$$cpu\_id_x = pid_x \% m_x$$

$$smp\_id_x = \lfloor pid_x / m_x \rfloor$$

The memory is physically distributed among nodes. Processors perform computations on local data. In order to use data calculated by a different processor,

1. if they reside in the same node, they should only synchronize with each other in order to make sure that the data neede have already been written to shared memory before used, or

2. if they reside in different nodes, they should communicate with each other via message passing or remote DMA, in order to exchange data that reside to remote memories.

The general intuition in the presented approach is that, since the iteration space is transformed by H and H' into a space of rectangular tiles, each processor can work on its local share of *rectangular* tiles and, following a proper memory allocation scheme, perform operations on rectangular data spaces as well. After all computations have been completed, locally computed data can be written back to the appropriate locations of the global data space. In this way, each processor essentially works on iteration and data spaces that are both rectangular, and properly translates from its local data space to the global one.

3.3.2 Computation Distribution

Computation distribution determines which computations of the sequential tiled code will be assigned to which processor. The n innermost loops of the sequential tiled code that access the internal points of a tile will not be parallelized, and thus parallelization only involves the distribution of tiles (traversed by the outermost n-dimensional loop) to processors. Hodzic and Shang in [HS98] mapped all tiles along a specific dimension to the same processor and used hyperplane $\Pi = [1, \ldots, 1]$ as time scheduling vector. In addition to this, previous work [AKPT99] in the field of UET-UCT task graphs has shown that if we map all tiles along the dimension with the maximum length (i.e. maximum number of tiles) to the same processor, then the overall scheduling is optimal, as long as the computation to communication ratio is one. This conclusion will also be verified in §4.4.4 for a bi-level parallel architecture. However, all research works resulting to this conclusion have assumed the existence of an infinite number of processors. We will keep on this assumption in this section also. In Chapter 5 we shall propose some allocation schemes in case there are fewer processors available than needed.

Let us denote the i-th dimension as the one with the maximum total length. According to the above, all tiles indexed by $\vec{j^S} = (j_1^S, \ldots, j_i^S, \ldots, j_n^S)$, where $j_k^S = const$, $k = 1, \ldots, i-1, i+1, \ldots, n$ and $l_i^S \leq j_i^S \leq u_i^S$ are executed by the same processor. The $n-1$ coordinates of a tile (excluding j_i^S) will identify the processor that a tile is going to be mapped to ($\vec{pid}$). All tiles along j_i^S are sequentially executed by the same processor, one after the other, in an order specified by a linear time schedule. This means that, after the selection of index j_i^S with the maximum trip count, we reorder all indices so that j_i^S becomes the innermost index. This corresponds to loop index interchange or permutation. Since all dependence vectors $\vec{d^S}$ in J^S are considered lexicographically positive, the interchanging or reordering of indices is valid (see also [PW86]). The boundaries of the reordered loop indices, in case of a non-rectangular tile space, can be calculated by an application of the Fourier-Motzkin elimination method [BW95].

3.3.3 Data Distribution

In a NUMA architecture, the data space of the original algorithm is distributed to the local memories of the various nodes forming the global data space. Data distribution decisions affect the communication volume, since data that reside in one node may be needed for the computation in another. In our approach we follow the **computer-owns** rule, which dictates that a processor owns the data it writes. It means that data computed by a processor are directly written to the local memory of the respective node. Communication occurs when a processor residing in another node needs to read data computed in the former one. Substantially, the memory space allocated by a node represents the space where computed data are to be stored. This means that the processors of each node iterate over a number of transformed rectangular tiles ($TTIS$) and can locally store their computed data to a rectangular data space. At the end of all their computations, the locally computed data can be placed to the appropriate positions of the global data space (DS). Thus, concerning the data writes, we can distinguish the following phases:

1. Data (initial and boundary values) are distributed to the local memories of the nodes, according to the computer-owns rule.

2. Data are locally computed by the processors of each node. Communication is interleaved between the execution of two tiles in order to receive data from neighboring nodes needed during the execution of subsequent tiles. The data received are locally stored.

3. At the end of all computations, locally computed data are written to the global data space (DS).

A simplified version of this procedure, concerning single CPU nodes, is extensively described in [GDAK02a], [Gou03].

The data space computed by a tile could be an exact image of the $TTIS$, but in this case the holes of the $TTIS$ would correspond to unused extra space. In addition to the space storing the computed data, each node needs to allocate extra space for communication, that is memory space to store the data it receives from its neighbors. This means that we need to

1. condense the actual points of the $TTIS$ and

2. provide further space for receiving data.

Since, after all transformations, we finally work with rectangular sets, this local data space (denoted LDS) allocated by a node, is given by the following definition.

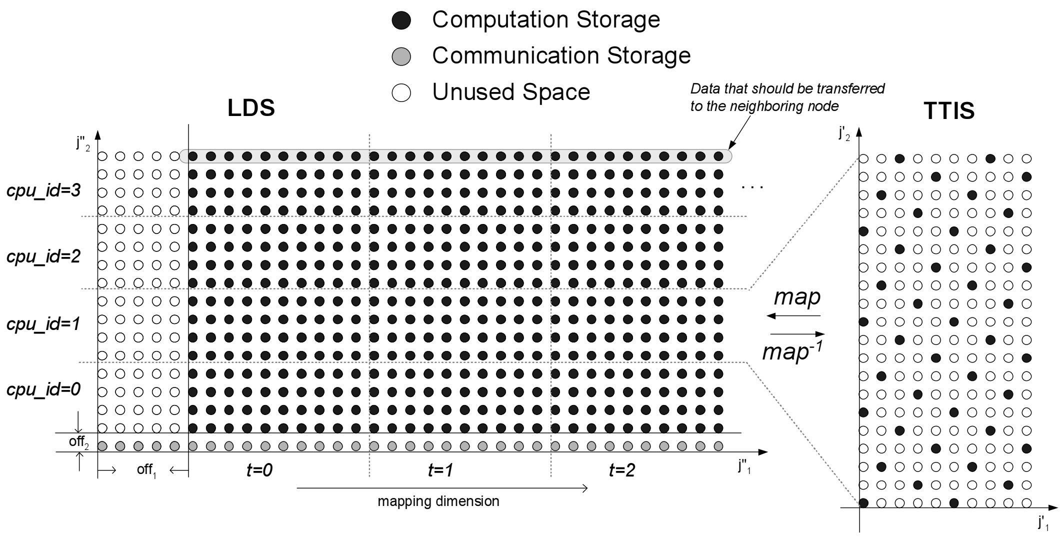


Figure 3.12: Local data space LDS and transformed tile iteration space $TTIS$

Definition 3.1 *The local data space (LDS) is defined as:*

$$LDS = \left\{ \vec{j''} \in Z^n | \begin{array}{c} 0 \leq j''_k < off_k + m_k v_{kk}/\widetilde{h'}_{kk}, k = 1, \ldots, n, k \neq i \\ \wedge 0 \leq j''_i < off_i + |t| v_{ii}/\widetilde{h'}_{ii} \end{array} \right\}$$

where $|t|$ denotes the maximum number of tiles assigned to a processor of the particular node.

As shown in Figure 3.12, the LDS of a processor consists of the memory space required for packing computed data (black dots) and for unpacking received data (grey dots) of a tile, multiplied by the number of tiles assigned to the particular processor. White dots depict unused data. The offset off_k, which expands the space to store received data, derives from the communication criteria of the algorithm, as shown in §3.3.4. Recall that each processor iterates over the $TTIS$ for as many times as the number of tiles assigned to that processor. Lemma 3.5 determines the translation function from $TTIS$ to LDS, while Lemma 3.6 determines the inverse translation function from LDS to $TTIS$.

Lemma 3.5 *If $\vec{j'} \in TTIS$, then its corresponding point in LDS is given by the following expressions:*

$$j''_k = \lfloor (cpu\_id_k v_{kk} + j'_k)/\widetilde{h'}_{kk} \rfloor + off_k, k \neq i$$

$$j''_i = \lfloor (tv_{ii} + j'_i)/\widetilde{h'}_{ii} \rfloor + off_i$$

where t is the current tile. We call this transformation function as map(): $\vec{j''} = map(\vec{j'}, t)$.

Proof: *In order to prove the validity of this transformation, we need to prove that the resulting point $\vec{j''} \in LDS$.*

1. *For each $k \neq i$ it holds that $0 \leq j'_k < v_{kk} \Rightarrow 0 \leq \lfloor \frac{j'_k}{\widetilde{h}'_{kk}} \rfloor < \frac{v_{kk}}{\widetilde{h}'_{kk}} \Rightarrow \frac{cpu\_id_k v_{kk}}{\widetilde{h}'_{kk}} + off_k \leq \frac{cpu\_id_k v_{kk}}{\widetilde{h}'_{kk}} + \lfloor \frac{j'_k}{\widetilde{h}'_{kk}} \rfloor + off_k < \frac{(cpu\_id_k+1) v_{kk}}{\widetilde{h}'_{kk}} + off_k$. Taking into account that $0 \leq cpu\_id_k \leq m_k - 1$, the previous inequality gives $off_k \leq \frac{cpu\_id_k v_{kk}}{\widetilde{h}'_{kk}} + off_k \leq j''_k < \frac{(cpu\_id_k+1) v_{kk}}{\widetilde{h}'_{kk}} + off_k \leq \frac{m_k v_{kk}}{\widetilde{h}'_{kk}} + off_k$.*
2. *In addition, $0 \leq j'_i < v_{ii} \Rightarrow 0 \leq \lfloor \frac{j'_i}{\widetilde{h}'_{ii}} \rfloor < \frac{v_{ii}}{\widetilde{h}'_{ii}} \Rightarrow \frac{tv_{ii}}{\widetilde{h}'_{ii}} + off_i \leq \frac{tv_{ii}}{\widetilde{h}'_{ii}} + \lfloor \frac{j'_i}{\widetilde{h}'_{ii}} \rfloor + off_i < \frac{(t+1)v_{ii}}{\widetilde{h}'_{ii}} + off_i$. Taking into account that $0 \leq t \leq |t| - 1$, the previous inequality gives $off_i \leq \frac{tv_{ii}}{\widetilde{h}'_{ii}} + off_i \leq j''_i < \frac{(t+1)v_{ii}}{\widetilde{h}'_{ii}} + off_i \leq \frac{|t| v_{ii}}{\widetilde{h}'_{ii}} + off_i$.*

Therefore, it holds that $\vec{j''} = map(\vec{j'}, t) \in LDS$. In addition, the proof of item (1) gives that the corresponding parts of LDS for each CPU of a node have no common elements, but they are neighboring iff CPUs are neighboring. The proof of item (2) gives that the corresponding parts of LDS for each tile of a processor have no common elements, but they are neighboring iff tiles are neighboring. ⊣

Lemma 3.6 *If $\vec{j''} \in LDS$, then its corresponding point in TTIS is given by the following expression:*

$$\vec{j'} = \widetilde{H'}\vec{x}$$

where $\vec{x}$ is given by:

$$x_k = j''_k - off_k - cpu\_id_k v_{kk}/\widetilde{h'}_{kk} - \lfloor(\sum_{l=1}^{k-1} x_l \widetilde{h'}_{kl})/\widetilde{h'}_{kk}\rfloor,\ k \neq i$$

$$x_i = j''_i - off_i - tv_{ii}/\widetilde{h'}_{ii} - \lfloor(\sum_{l=1}^{i-1} x_l \widetilde{h'}_{il})/\widetilde{h'}_{ii}\rfloor$$

where $t = \lfloor(j''_i - off_i)\widetilde{h'}_{ii}/v_{ii}\rfloor$ is the current tile. We call this transformation function as $map^{-1}()$: $(\vec{j'}, t) = map^{-1}(\vec{j''})$.

Proof: *We need to prove that map and map^{-1} are indeed inverse functions. Equivalently, we should prove that*

1. $(\vec{j'}, t) = map^{-1}(map(\vec{j'}, t))$ *and*
2. $\vec{j''} = map(map^{-1}(\vec{j''}))$.

1. $(\vec{j'}, t) \stackrel{?}{=} map^{-1}(map(\vec{j'}, t)) \Leftrightarrow \left\{ \begin{array}{ll} t \stackrel{?}{=} \lfloor \frac{((\lfloor \frac{tv_{ii}+j'_i}{h'_{ii}} \rfloor + off_i) - off_i)\widetilde{h}'_{ii}}{v_{ii}} \rfloor & (a) \\ \wedge & \\ j'_l \stackrel{?}{=} \sum_{k=1}^{l} \widetilde{h}'_{lk} y_k & (b) \end{array} \right.,$

where: $\left\{ \begin{array}{l} y_k = ((\lfloor \frac{cpu\_id_k v_{kk} + j'_k}{\widetilde{h}'_{kk}} \rfloor + off_k) - off_k - \frac{cpu\_id_k v_{kk}}{\widetilde{h'}_{kk}}) - \lfloor \frac{\sum_{l=1}^{k-1} \widetilde{h}'_{kl} y_l}{\widetilde{h}'_{kk}} \rfloor, k \neq i \\ y_i = ((\lfloor \frac{tv_{ii}+j'_i}{\widetilde{h}'_{ii}} \rfloor + off_i) - off_i - \frac{tv_{ii}}{\widetilde{h}'_{ii}}) - \lfloor \frac{\sum_{l=1}^{i-1} \widetilde{h}'_{il} y_l}{\widetilde{h}'_{ii}} \rfloor \end{array} \right\}$

$\Leftrightarrow y_k = \lfloor \frac{j'_k}{\widetilde{h}'_{kk}} \rfloor - \lfloor \frac{\sum_{l=1}^{k-1} \widetilde{h}'_{kl} y_l}{\widetilde{h}'_{kk}} \rfloor \ \forall k$

(a) *However,* $t \stackrel{?}{=} \lfloor \frac{((\lfloor \frac{tv_{ii}+j'_i}{h'_{ii}} \rfloor + off_i) - off_i)\widetilde{h}'_{ii}}{v_{ii}} \rfloor \Leftrightarrow t \stackrel{?}{=} t + \lfloor \frac{\lfloor \frac{j'_i}{h'_{ii}} \rfloor \widetilde{h}'_{ii}}{v_{ii}} \rfloor$. *From* $0 \leq j'_i < v_{ii} \Rightarrow 0 \leq \lfloor \frac{j'_i}{\widetilde{h}'_{ii}} \rfloor < \frac{v_{ii}}{\widetilde{h}'_{ii}} \Rightarrow 0 \leq \lfloor \frac{j'_i}{\widetilde{h}'_{ii}} \rfloor \widetilde{h}'_{ii} < v_{ii} \Rightarrow 0 \leq \lfloor \frac{\lfloor \frac{j'_i}{h'_{ii}} \rfloor \widetilde{h}'_{ii}}{v_{ii}} \rfloor < 1 \Rightarrow \lfloor \frac{\lfloor \frac{j'_i}{h'_{ii}} \rfloor \widetilde{h}'_{ii}}{v_{ii}} \rfloor = 0$. *Thus,* $t \stackrel{?}{=} t + \lfloor \frac{\lfloor \frac{j'_i}{h'_{ii}} \rfloor \widetilde{h}'_{ii}}{v_{ii}} \rfloor \Leftrightarrow t \stackrel{?}{=} t + 0$, *which is always valid.*

(b) *In addition, from* $y_k = \lfloor \frac{j'_k}{\widetilde{h}'_{kk}} \rfloor - \lfloor \frac{\sum_{l=1}^{k-1} \widetilde{h}'_{kl} y_l}{\widetilde{h}'_{kk}} \rfloor \Rightarrow \lfloor \frac{j'_k}{\widetilde{h}'_{kk}} \rfloor = y_k + \lfloor \frac{\sum_{l=1}^{k-1} \widetilde{h}'_{kl} y_l}{\widetilde{h}'_{kk}} \rfloor = \lfloor \frac{\sum_{l=1}^{k} \widetilde{h}'_{kl} y_l}{\widetilde{h}'_{kk}} \rfloor \Rightarrow \widetilde{h}'_{kk} \lfloor \frac{\sum_{l=1}^{k} \widetilde{h}'_{kl} y_l}{\widetilde{h}'_{kk}} \rfloor \leq j'_k \leq \widetilde{h}'_{kk} \lfloor \frac{\sum_{l=1}^{k} \widetilde{h}'_{kl} y_l}{\widetilde{h}'_{kk}} \rfloor + \widetilde{h}'_{kk} - 1$. *In this interval,* *there is exactly one actual point j'_k (as $\widetilde{h}'_{kk}$ is the step of j'_k in order to meet another actual point), which is $\sum_{l=1}^{k} \widetilde{h}'_{kl} y_l$. Therefore, it holds that $j'_k = \sum_{l=1}^{k} \widetilde{h}'_{kl} y_l$.*

2.

$$\vec{j''} \stackrel{?}{=} map(map^{-1}(\vec{j''})) \Leftrightarrow \left\{ \begin{array}{c} j''_k \stackrel{?}{=} \lfloor \frac{cpu\_id_k v_{kk} + \sum_{l=1}^{k} \widetilde{h}'_{kl} z_l}{\widetilde{h}'_{kk}} \rfloor + off_k, k \neq i \\ \wedge \\ j''_i \stackrel{?}{=} \lfloor \frac{tv_{ii} + \sum_{l=1}^{i} \widetilde{h}'_{il} z_l}{\widetilde{h}'_{ii}} \rfloor + off_i \end{array} \right\} \quad (3.18)$$

$$\textit{where:} \left\{ \begin{array}{c} z_l = j''_l - off_l - \frac{cpu\_id_l v_{ll}}{\widetilde{h'}_{ll}} - \lfloor \frac{\sum_{k=1}^{l-1} \widetilde{h}'_{lk} z_k}{\widetilde{h}'_{ll}} \rfloor, l \neq i \\ z_i = j''_i - off_i - \frac{tv_{ii}}{\widetilde{h}'_{ii}} - \lfloor \frac{\sum_{k=1}^{i-1} \widetilde{h}'_{ik} z_k}{\widetilde{h}'_{ii}} \rfloor \end{array} \right\} \Rightarrow$$

$$\left\{ \begin{array}{c} j''_l = off_l + \frac{cpu\_id_l v_{ll}}{\widetilde{h'}_{ll}} + \lfloor \frac{\sum_{k=1}^{l} \widetilde{h}'_{lk} z_k}{\widetilde{h}'_{ll}} \rfloor, l \neq i \\ j''_i = off_i + \frac{tv_{ii}}{\widetilde{h}'_{ii}} + \lfloor \frac{\sum_{k=1}^{i} \widetilde{h}'_{ik} z_k}{\widetilde{h}'_{ii}} \rfloor \end{array} \right. \quad (3.19)$$

Therefore, (3.18) $\stackrel{(3.19)}{\Longleftrightarrow}$

$$\left\{ \begin{array}{c} off_k + \frac{cpu\_id_k v_{kk}}{\widetilde{h'}_{kk}} + \lfloor \frac{\sum_{l=1}^{k} \widetilde{h}'_{kl} z_l}{\widetilde{h}'_{kk}} \rfloor \stackrel{?}{=} \lfloor \frac{cpu\_id_k v_{kk} + \sum_{l=1}^{k} \widetilde{h}'_{kl} z_l}{\widetilde{h}'_{kk}} \rfloor + off_k, k \neq i \\ \wedge \\ off_i + \frac{tv_{ii}}{\widetilde{h}'_{ii}} + \lfloor \frac{\sum_{k=1}^{i} \widetilde{h}'_{ik} z_k}{\widetilde{h}'_{ii}} \rfloor \stackrel{?}{=} \lfloor \frac{tv_{ii} + \sum_{l=1}^{i} \widetilde{h}'_{il} z_l}{\widetilde{h}'_{ii}} \rfloor + off_i \end{array} \right.$$

which is apparently always valid, taking into account that v_{kk} is always a multiple of $\widetilde{h}'_{kk}$, $\forall k = 1, \ldots, n$.

After proving both claims (1) and (2), it turns out that this lemma is always valid. ⊣

Function $map(\vec{j'}, t)$ determines, according to Lemma 3.5, the memory location in LDS where computation for iteration $\vec{j'} \in TTIS$ is to be stored (Figure 3.12). Function $loc(\vec{j})$ in Table 3.8 uses $map(\vec{j'}, t)$ in order to locate the processor $\vec{pid}$ and the memory location $\vec{j''} \in LDS$, where the computed data of iteration point $\vec{j} \in J^n$ is to be stored. Inversely, Table 3.9 shows the series of steps in order to locate the corresponding $\vec{j} \in J^n$ for a point $\vec{j''} \in LDS$ of processor $\vec{pid}$. Thus, $loc^{-1}()$ is called by a processor of each node at the end of the node's computations in order to transit from their LDS to the original iteration space J^n. In the sequel, the corresponding point in the data space DS is found via f_w (Figure 3.13).

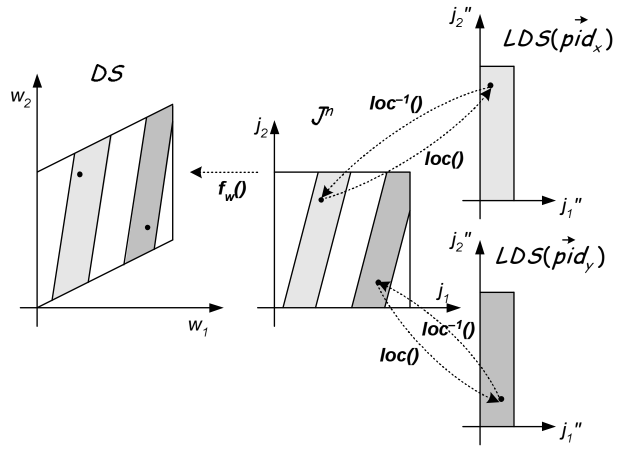


Figure 3.13: Relations between DS, J^n and LDS

Table 3.8: Using function $loc()$ to locate $\vec{j} \in J^n$ in the LDS of a processor

$\vec{j''} = map(\vec{j'}, t)$:
$j''_k = \lfloor (cpu\_id_k v_{kk} + j'_k)/\widetilde{h'}_{kk} \rfloor + off_k, k \neq i$ $j''_i = \lfloor (tv_{ii} + j'_i)/\widetilde{h'}_{ii} \rfloor + off_i$

$(\vec{j''}, \vec{pid}) = loc(\vec{j})$:
$\vec{j^S} = \lfloor H\vec{j} \rfloor$ $\vec{j'} = H'(\vec{j} - P\vec{j^S})$ $\vec{j''} = map(\vec{j'}, j^S_i - min\{l^S_i\})$ $\vec{pid} = (j^S_1, \ldots, j^S_{i-1}, j^S_{i+1}, \ldots, j^S_n)$

Under this scheme, each node allocates exactly the amount of local memory needed for computation and communication (minor over-allocation occurs in the few boundary tiles). Note that direct allocation of a node's share in the original DS would lead to a waste of memory space, since this generally non-rectangular share would lead to the allocation of the minimum enclosing rectangular memory space. Note, also, that each node's share in the original DS (the footprint of a tile because of f_w) is in general non-rectangular, even if a rectangular tiling transformation is applied. This method, however, forces the local data space of each node to be rectangular, allowing thus more efficient memory management. In addition, if we also take into account that data spaces for common computationally intensive algorithms are very large, and will probably not fit in each node's memory, the compression of the local space to the LDS is in most cases necessary. Eventually, this leads to a trade-off between computational complexity and allocated memory space, since extra expressions are needed to address the LDS, but this minor overhead does not significantly affect performance, as indicated by the experimental verification of [GDAK02a]. Finally, note that storing data accessed by a non-rectangular tile to a dense rectangular data space also exploits cache locality.

Table 3.9: Using function $loc^{-1}()$ to locate $\vec{j''} \in LDS$ of processor $\vec{pid}$ in J^n

$(\vec{j'}, t) = map^{-1}(\vec{j''})$:
$t = \lfloor (j''_i - off_i)\widetilde{h'}_{ii}/v_{ii} \rfloor$ $x_k = j''_k - off_k - cpu\_id_k v_{kk}/\widetilde{h'}_{kk} - \lfloor (\sum_{l=1}^{k-1} x_l \widetilde{h'}_{kl})/\widetilde{h'}_{kk} \rfloor, k \neq i$ $x_i = j''_i - off_i - tv_{ii}/\widetilde{h'}_{ii} - \lfloor (\sum_{l=1}^{i-1} x_l \widetilde{h'}_{il})/\widetilde{h'}_{ii} \rfloor$ $\vec{j'} = \widetilde{H'}\vec{x}$

$\vec{j} = loc^{-1}(\vec{j''}, \vec{pid})$:
$\vec{j'} = map^{-1}(\vec{j''})$ $\vec{j^S} = (pid_1, \ldots, pid_{i-1}, t + min\{l_i^S\}, pid_{i+1}, \ldots, pid_n)$ $\vec{j} = P'(V\vec{j^S} + \vec{j'})$

3.3.4 Communication sets

Using the iteration and data distribution schemes described before, data that reside in the local memory of one node may be needed by another due to algorithmic dependences. In this case, the nodes need to communicate via message passing or remote DMA. The two fundamental issues that need to be addressed regarding communication are

1. the specification of the processors each processor needs to communicate with, and
2. the determination of the data that need to be transferred.

As far as the first issue is concerned, each processor needs to exchange data with its neighbors only in case they reside in a different node. That is, processors with $cpu\_id_x = 0 \Leftrightarrow pid_x \% m_x = 0$ need to receive data from processors with $pid'_x = pid_x - 1$. Similarly, processors with $cpu\_id_x = m_x - 1 \Leftrightarrow pid_x \% m_x = m_x - 1$ should send data to neighboring processors with $pid'_x = pid_x + 1$ (see Figure 3.14). When neighboring processors reside in the same node, they should only synchronize with each other, in order to make sure that data have been written to the shared memory of the node before used.

As far as the communication data are concerned, we focus on the communication points, as defined below:

Definition 3.2 *Let i be the mapping dimension. Let $\vec{d^S} \in D^S$ be a tile dependence that implies processor dependence, that is $\exists l \neq i : d_l^S \neq 0$. A point $\vec{j'} \in TTIS$ is considered a communication point respective to $\vec{d^S}$ iff the computed data at iteration $\vec{j} = P'(V\vec{j^S} + \vec{j'})$ is needed by tile $\vec{j^S} + \vec{d^S}$, where $\vec{j^S} \in J^S$ and $\vec{j^S} + \vec{d^S} \in J^S$, and $\vec{j^S} + \vec{d^S}$ has been allocated to a different node than $\vec{j^S}$.*

Note that a communication point is only defined in respect to a specific tile dependence $\vec{d^S}$. In other words, communication points in the $TTIS$ correspond to iterations at which data are computed by one node and need to be sent to another node in tile direction $\vec{d^S}$.

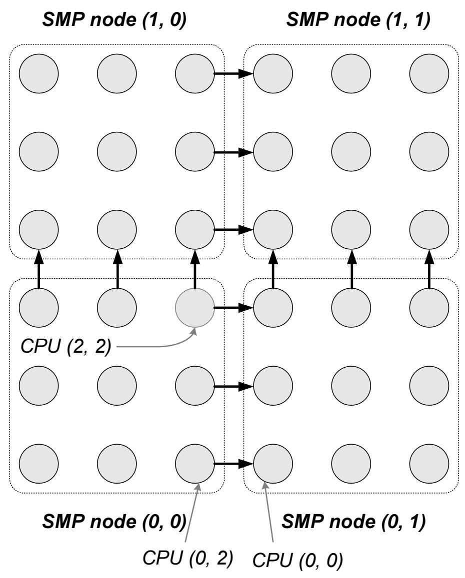


Figure 3.14: Communication among processors.
Only processors with neighbors in a different node need to transfer data among them. Neighboring processors within the same node should only synchronize with each other, in order to make sure that data have been written to the shared memory of the node before used.

We further exploit the regularity of the $TTIS$ and LDS to deduce simple criteria for the communication points at compile time. The following lemma is useful:

Lemma 3.7 *A point $\vec{j'} = (j'_1, \ldots, j'_n) \in TTIS$ corresponds to a communication point respective to a tile dependence $\vec{d^S} = (d^S_1, \ldots, d^S_n) \in D^S$ iff it holds:*

$$j'_k \geq d^S_k(v_{kk} - \max_{\vec{d'} \in D'}\{d'_k\})$$

where $k = 1, \ldots, n, \vec{d'} \in D', D' = H'D$, and tile $\vec{j^S} + \vec{d^S}$ has been allocated to a different node than $\vec{j^S}$.

Proof: *For $\vec{j'}$ to be a communication point according to the k-th dimension, we distinguish two cases:*

1. *$d^S_k = 0$. Since no tile dependence is enforced in this case, no limitation for j'_k is defined. So it holds $0 \leq j'_k \leq v_{kk} - 1$.*
2. *$d^S_k = 1$. In this case, there must exist a data dependence in the $TTIS$ $\vec{d'} \in D'$ such, that the k-th component of $\vec{j'} + \vec{d'}$ exceeds the respective bound of the $TTIS$, thus incurring need for communication according to the k-th dimension. According to the above, it must hold*

$$j'_k + d'_k > v_{kk} - 1 \Rightarrow j'_k + d'_k \geq v_{kk} \Rightarrow j'_k \geq v_{kk} - d'_k$$

for some $\vec{d'} \in D'$ *or equivalently*

$$j'_k \geq v_{kk} - \max_{\vec{d'} \in D'} \{d'_k\}$$

The unification of both cases leads to the given condition. ⊣

Thus, it is advantageous to identify the communication data in the $TTIS$, as opposed to the other possible alternatives (e.g. the initial iteration space, the TIS etc.) which would complicate the communication procedure. Also, note that the offsets in LDS referenced in §3.3.3 can easily arise as follows:

$$off_k = \lceil \max_{\vec{d'} \in D'} \{d'_k\} / \widetilde{h'}_{kk} \rceil, \ \forall k = 1, \ldots, n \tag{3.20}$$

The instances of LDS corresponding to the communication points, as defined by Lemma 3.7, can be calculated by the expression:

$$j''_k \geq m_k v_{kk} / \widetilde{h'}_{kk} \tag{3.21}$$

for each tile dependence $\vec{d^S}$ with $d^S_k \neq 0$.

Example 3.5: Continuing Example 3.4, we consider that the tiled nested loops will be executed by a cluster of SMP nodes with 4 processors each. According to Figure 3.2, the maximum total length corresponds to dimension j^S_1. Thus, according to §3.3.2, j^S_1 should be selected as the mapping dimension of this example.

Since $D' = H'D = \begin{bmatrix} 2 & -1 \\ -1 & 3 \end{bmatrix} \begin{bmatrix} 3 & 1 \\ 1 & 2 \end{bmatrix} = \begin{bmatrix} 5 & 0 \\ 0 & 5 \end{bmatrix}$, the offset parameters of LDS are given by formula (3.20) as follows:

$$off_1 = \lceil \max_{\vec{d'} \in D'} \{d'_1\} / \widetilde{h'}_{11} \rceil = \lceil 5/1 \rceil = 5$$

$$off_2 = \lceil \max_{\vec{d'} \in D'} \{d'_2\} / \widetilde{h'}_{22} \rceil = \lceil 5/5 \rceil = 1$$

According to Definition 3.1, as depicted in Figure 3.12, the local data space LDS is defined as follows:

$$LDS = \{\vec{j''} \in Z^n | 0 \leq j''_1 < 5 + |t|10/1 = 5 + 10|t| \wedge 0 \leq j''_2 < 1 + 4 \cdot 20/5 = 17\}$$

where $|t|$ denotes the maximum number of tiles assigned to a processor of the particular node. According to formula (3.21), as indicated in Figure 3.12, the data that are computed in this node and should be transferred to a neighboring one, reside in the positions of LDS with $j''_2 \geq m_2 v_{22}/\widetilde{h}'_{22} = 4 \cdot 20/5 = 16$.

4

Execution of tiles onto clusters of Symmetric Multiprocessors (SMP nodes)

In this chapter, the execution policies of non-overlapping and overlapping communication with computation, are generalized, in order to be applied onto PC clusters with more than one CPUs each. In order to achieve this generalization, we introduce the technique of grouping, which is a tiling transformation applied onto tiles. Afterwards, we produce a linear time scheduling of groups, which seems to be optimal, while any linear scheduling of tiles would be suboptimal, since the communication requirements among tiles are different. We also indicate how computation tasks should be allocated to the processors and we determine the guidelines for the selection of the grouping parameters. Finally, we theoretically and experimentally validate the techniques proposed.

4.1 An Intuitive Approach

Before starting with the full demonstration of the proposed techniques, we will intuitively illustrate the basic concepts of our method, using an example. Let us consider the following scenario: A 2-dimensional nested loop is to be executed onto a cluster of 3 identical single CPU nodes. We tile the iteration space of the code segment and assign each row of tiles to a CPU node. In order to achieve an easy allocation of tiles to CPUs, the size and shape of tiles should be selected so that the iteration space is partitioned into 3 rows of tiles (since 3 CPUs are available). Then, the tiles can be computed using either the overlapping, or the non-overlapping scheme presented in §2.7.

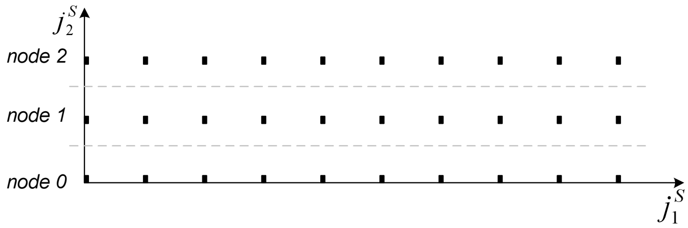


Figure 4.1: Execution of tiles on single-CPU nodes.
If the cluster consists of 3 single-CPU nodes, the initial iteration space is partitioned into 3 rows of tiles.

In the sequence, each single CPU node is replaced by an SMP node, with 2 CPUs. The first solution one may think of, is tiling the initial iteration space from scratch, selecting the tile size so as to get six rows of tiles. Then, a row of tiles may be assigned to each CPU and executed as if there were six single CPU nodes. This would mean that even CPUs inside the same SMP node should communicate with each other via message passing, in order to exchange the data needed. The result of such a consideration may be unnecessary transfers from the processing unit to the network card and vice versa, which will consume a portion of the intra-node communication bandwidth. In the best case, when the compiler can detect and prevent such unnecessary communication between the processor and the network card, it will not evict unnecessary transfers among the shared and private space of threads inside the same SMP node [DK04]. In fact, they can simply write and read the data needed directly to and from shared memory. Then, they should only synchronize with each other using a barrier or a semaphore.

The above consideration leads to the conclusion that iterations assigned to the same SMP node should be more tightly connected to each other, than simply being mapped to neighboring tiles. Maybe they can belong to the same tile, or to an entity inheriting some properties of tiling.

In order to adjust the tile space of Figure 4.1 to this computing architecture, we can split each tile into two subtiles and assign each subtile to one of the CPUs of the corresponding SMP node, as indicated in Figure 4.2. Then, one may schedule tiles as if they were unsplit and take care so as to execute subtiles of a tile at the same time.

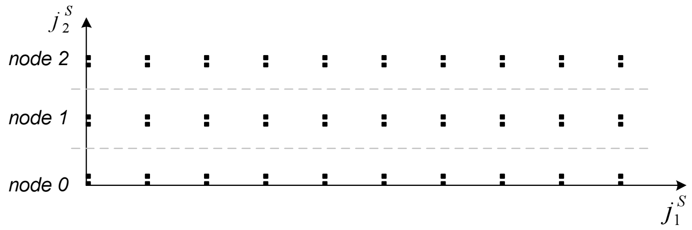


Figure 4.2: Execution of tiles on SMP nodes with 2 CPUs each.
Each tile of Figure 4.1 is divided into 2 subtiles and each CPU undertakes a subtile during each time step.

Equivalently, the initial iteration space may be tiled from scratch, selecting the size of tiles so as to form six rows of tiles. Then, one row of tiles is assigned to each CPU of the SMP nodes neighboring tiles, assigned to the same SMP node, are grouped together, as in Figure 4.3. Because of tile dependences, the tiles grouped together by this scheme cannot be simultaneously executed, unless they are split into subtiles. Thus, additional synchronization overhead is necessary due to dependences among subtiles, which have been assigned to different CPUs of the same node, but should be executed during the same time step.

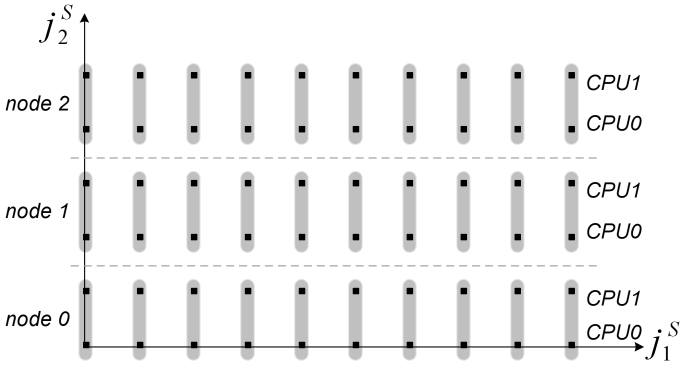


Figure 4.3: Vertical grouping.
Neighboring tiles should be executed at the same time by CPUs of the same node. There are dependences among tiles executed during the same time step.

A more efficient scheme can be obtained if the tiles assigned to the same SMP nodes are grouped as indicated in Figure 4.4. Then, both tiles belonging to the same group can be simultaneously executed by the CPUs of an SMP node, without a need for communication or synchronization. Only one synchronization per tile is required, in order to certify that the data needed are located in the shared memory. This synchronization (implemented by a barrier or a semaphore) can be contemporary with the communication with CPUs of different SMP nodes.

In the rest of this thesis, we shall call this grouping scheme as **hyperplane grouping**. On the contrary, any other grouping scheme along a specific dimension, such as the one pre-

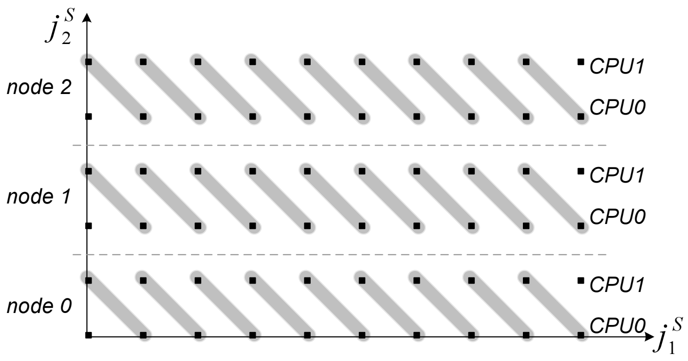


Figure 4.4: Hyperplane grouping.
There are no dependences among tiles executed during the same time step.

sented in Figure 4.3, will be called **vertical grouping**. Vertical grouping imposes additional synchronization overhead, due to dependences among tiles of the same group.

4.2 Grouping Transformation

As shown in §4.1, efficient scheduling of tiled iteration spaces onto a parallel architecture consisting of SMP nodes, is not a straightforward task. In order to generate an appropriate time schedule, we need to group together the tiles of J^S that can be concurrently executed by the CPUs of the same SMP node. It can be achieved by applying an additional supernode, or tiling transformation to the tile space J^S. We name this supernode transformation as **grouping transformation**.

Thus, from the tile space J^S we produce the **group space**

$$J^G = \{\vec{j^G} | \vec{j^G} = \lfloor H^G \vec{j^S} \rfloor, \vec{j^S} \in J^S\} \tag{4.1}$$

in correspondence to formula (2.4) for tiling. This grouping transformation is defined by the $n \times n$ non-singular matrix H^G (similarly to matrix H defining tiling transformation). In correspondence to the tiling matrix H, the $n \times n$ matrix H^G is called **grouping matrix**. Each row-vector of H^G is perpendicular to one of the families of hyperplanes that define the boundaries of the groups in J^S. The $n \times n$ matrix $P^G = (H^G)^{-1}$ is called **inverse grouping matrix**. The matrix P^G should consist only of integer elements and its column-vectors are parallel and equal in size to the edges of a group-hyperparallelepiped in J^S.

In order to be valid, a grouping transformation should preserve the constraint of atomicity of groups ($H^G D^S \geq 0$ in correspondence to $HD \geq 0$ for tiling). In addition, since within a group all tiles are concurrently executed by the CPUs of an SMP node, in order to preserve data consistency, there should be no direct or indirect dependence among them. Equivalently, for each dependence vector $\vec{d_i^S}$ in the tile space, vector $H^G\vec{d_i^S}$ should have at least one element greater than or equal to 1.

4.3 Intuition of our algorithm

Thus, just as tiling transformation is used to summon iteration points into tiles, grouping transformation is applied after tiling transformation, in order to form suitable groups of tiles. A desirable tiling transformation is the one that minimizes communication overhead [Xue97a], [AKN95], [RR02], [RS92], [BDRR94], or total execution time [HCF97], [HCF99], [DDRR97] [XC02]. Respectively, in the following paragraphs, we shall define the criteria for an efficient grouping transformation and we shall propose a theory for determining it.

Let us consider a 3-dimensional tile space J^S. We want to assign all tiles along dimension j_1^S to the same CPU of an SMP node. Since all CPUs within a node have access to the shared memory, neighboring rows of tiles, which exchange data, are assigned to the CPUs of the same node. In this way, the part of the tile space assigned to a node will be of a rectangular shape, as depicted in Figure 4.5.

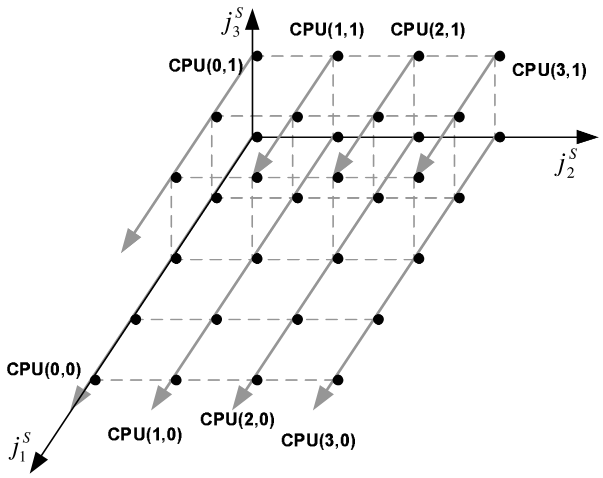


Figure 4.5: Set of tiles assigned to an SMP node.
All dots along a grey arrow correspond to tiles, which are assigned to the same CPU of the SMP node. They are executed one after the other, during consecutive time steps.

We seek for an appropriate transformation matrix that will group together the tiles of Figure 4.5, which can be executed simultaneously by different CPUs. The execution of the portion of the tile space, which has been assigned to an SMP node, resembles the execution of a UET grid, as described in [AKPT99]. According to [AKPT99], the optimal valid linear scheduling vector for an iteration space (or tile space) with unitary dependence vectors (as imposed by §B.5), is $(1,1,1)$, when the time required for communication is minimal. In our example, the communication among CPUs of a node recoils to a synchronization. Thus, it may be considered conformal to the UET communication model. So, we shall group together the tiles that belong to the same plane which is perpendicular to the vector $(1, 1, 1)$, as indicated in Figure 4.6.

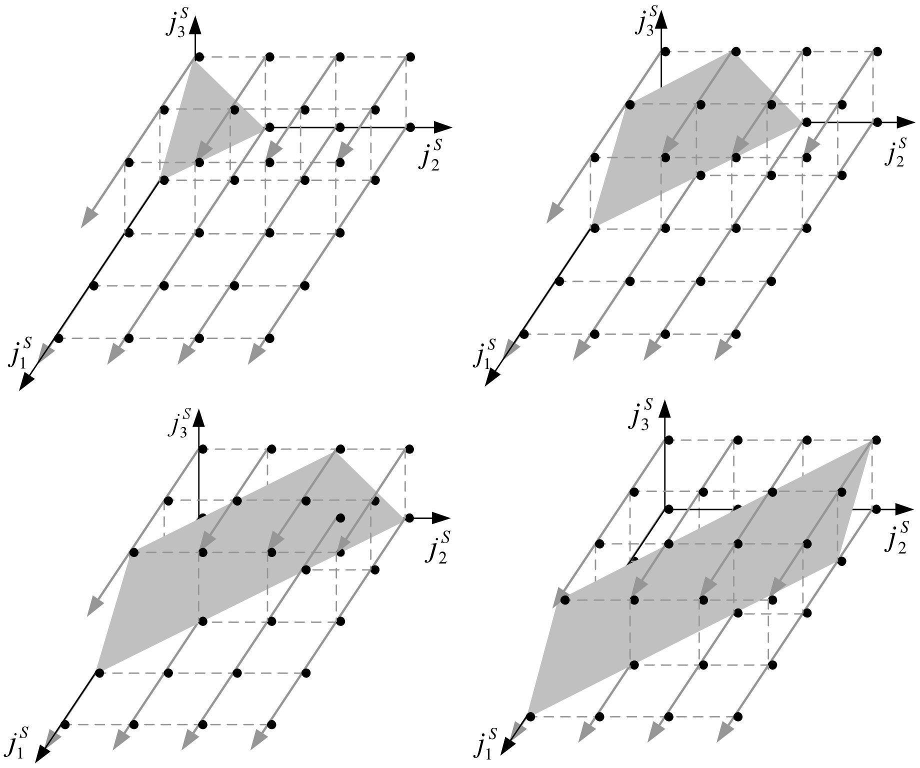


Figure 4.6: Groups of tiles executed simultaneously in an SMP node. The tiles of the same grey plane belong to the same group and will be executed at the same time by different CPUs of the same node. Subfigures correspond to consecutive time steps.

The column-vectors of the inverse grouping matrix P^G define a hyper-parallelepiped (in general) that contains the tiles of a group, similar to the way the columns of P define a tile. Thus, vectors $\vec{p_2^G}$ and $\vec{p_3^G}$ should be parallel to the plane $j_1^S + j_2^S + j_3^S = const$ and, at the same time, they should be parallel to one of the planes defining the bounds of the set allocated to this SMP node. That is, they should be parallel to the planes $j_3^S = 0$ and $j_2^S = 0$ respectively. Therefore, the appropriate vectors are

$$\vec{p_2^G} = \lambda(-1, 1, 0) \text{ and } \vec{p_3^G} = \mu(-1, 0, 1)$$

(In Figures 4.5-4.7 it holds $\lambda = 4, \mu = 2$.) In addition, in order to cover exactly the part of the tile space allocated to an SMP node using a series of successive groups, vector $\vec{p_1^G}$ should be constructed parallel to both the planes $j_2^S = 0$ and $j_3^S = 0$. Therefore, the appropriate vector is

$$\vec{p_1^G} = (1, 0, 0)$$

Thus, the appropriate inverse grouping matrix is

$$P^G = \begin{bmatrix} 1 & -\lambda & -\mu \\ 0 & \lambda & 0 \\ 0 & 0 & \mu \end{bmatrix}$$

where $\lambda, \mu \in N$. The maximum number of tiles grouped together will be $det(P^G) = \lambda\mu$ and this product must be equal to the number of CPUs inside a node, so as to assign one tile to each CPU during each time step.

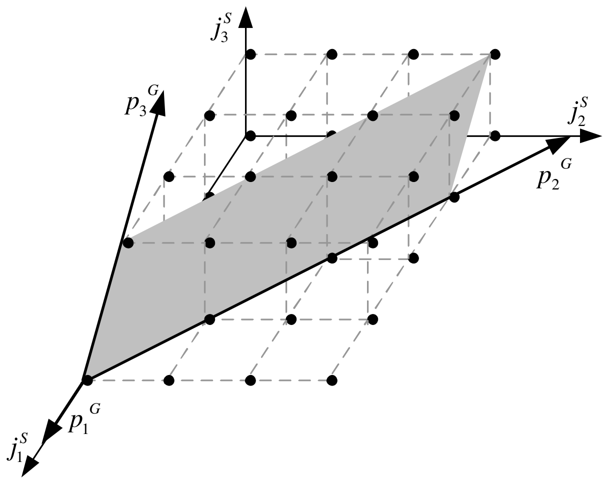


Figure 4.7: Constructing the inverse grouping matrix.

Vectors p_i^G should be parallel to the edges of a group-parallelepiped. Their norm should be equal to the length of the corresponding edge.

4.4 Determining P^G according to the number of CPUs within an SMP node

Consider now the general case: We have an n-dimensional tiled iteration space and an homogeneous cluster of identical SMP nodes, each with m processors inside. Our objective is to assign the tiles of J^S along the first dimension to the same CPU of an SMP node. The natural number m can be written as $m = m_2 \times m_3 \times \cdots \times m_n$, where $m_2, m_3, \ldots, m_n \in N$. The grouping matrices are selected to be

$$P^G = \begin{bmatrix} 1 & -m_2 & \ldots & -m_n \\ 0 & m_2 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & m_n \end{bmatrix} \text{ and } H^G = (P^G)^{-1} = \begin{bmatrix} 1 & 1 & \ldots & 1 \\ 0 & \frac{1}{m_2} & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & \frac{1}{m_n} \end{bmatrix} \quad (4.2)$$

The maximum number of tiles contained inside a group is $det(P^G) = m$, exactly equal to the number of CPUs inside each SMP node.

Theorem 4.1 *In the algorithmic model, which is summarized in Appendix B, matrix H^G, defined by formula (4.2), defines a legal grouping transformation.*

Proof: *In order to prove that H^G defines a legal grouping transformation, it suffices to prove that*

1. *$H^G D^S \geq 0$, where D^S is the dependence matrix of the tile space J^S*
2. *any two tiles $\vec{j^S}, \vec{j^S}'$ within the same group are independent.*

We have assumed (see §2.6.3 and restriction B.5) that the dependence matrix D^S contains only 0's and 1's. Consequently, the first condition is apparently valid.

In order to prove the second condition, we assume that the dependence matrix D^S is equal to the unitary matrix. Even if there is a dependence vector with more than one 1's, it is the sum of more than one unitary dependence vectors. So it will be included in the following proof as an indirect dependence:

If tiles $\vec{j^S}, \vec{j^S}' \in J^S$ belong to the same group $\vec{j^G}$, then it holds that:

$$\lfloor H^G \vec{j^S} \rfloor = \lfloor H^G \vec{j^S}' \rfloor \Rightarrow \begin{pmatrix} j_1^S + j_2^S + \cdots + j_n^S \\ \lfloor \frac{j_2^S}{m_2} \rfloor \\ \vdots \\ \lfloor \frac{j_n^S}{m_n} \rfloor \end{pmatrix} = \begin{pmatrix} j_1^{S'} + j_2^{S'} + \cdots + j_n^{S'} \\ \lfloor \frac{j_2^{S'}}{m_2} \rfloor \\ \vdots \\ \lfloor \frac{j_n^{S'}}{m_n} \rfloor \end{pmatrix} \Rightarrow$$

$$j_1^S + j_2^S + \cdots + j_{n-1}^S + j_n^S = j_1^{S'} + j_2^{S'} + \cdots + j_{n-1}^{S}{}' + j_n^{S'}$$

In addition, if there is a direct or an indirect dependence from $\vec{j^S}$ to $\vec{j^S}'$, it holds that

$$\vec{j^S}' = \vec{j^S} + \sum_{i=1}^{n} \lambda_i \vec{d_i},$$

where $\lambda_i \in N$ and $\vec{d_i}$ is a unitary dependence vector. The previous equality can be rewritten as follows: $\vec{j^S}' = \vec{j^S} + \vec{\lambda}$, where $\vec{\lambda} = (\lambda_1, \ldots, \lambda_n)$. Thus,

$$j_i^{S'} = j_i^S + \lambda_i, \ i = 1, \ldots, n$$

Therefore, the equality $j_1^S + j_2^S + \cdots + j_{n-1}^S + j_n^S = j_1^{S'} + j_2^{S'} + \cdots + j_{n-1}^{S}{}' + j_n^{S'}$ can be rewritten as follows:

$$\lambda_1 + \lambda_2 + \cdots + \lambda_n = 0$$

As $\lambda_1, \ldots, \lambda_n \in N$, it holds that

$$\lambda_1 = \cdots = \lambda_n = 0$$

Consequently, there is no direct or indirect dependence between two tiles belonging to the same group $\vec{j^G} \in J^G$ and all tiles of a group in J^G can be computed simultaneously by the CPUs of an SMP node. Thus, the above grouping transformation is valid according to our algorithmic model. ⊣

Example 4.1: We afford a cluster of SMP nodes with 2 CPUs and one NIC (Network Interface Card) each. The NICs provide the facility of Direct Memory Access (DMA). Thus, the overlapping execution policy can be implemented. We assume a 2-dimensional rectangular tile space J^S. Let us assign the tiles along dimension j_1^S to the same CPU, as indicated in Figure 4.8 by the grey arrows. The CPUs of the same SMP node will process two neighboring rows of tiles.

Then, during the time step t=0, CPU 0 of SMP node 0 computes tile $(0,0)$. During the time step $t=1$, CPU 0 of node 0 computes tile $(1,0)$, while CPU 1 of the same SMP node computes tile $(0,1)$. Similarly, during the time step $t=2$, CPU 0 computes tile $(2,0)$, while CPU 1 computes tile $(1,1)$. At the same time, the data computed in tile $(0,1)$, which are necessary for the computation of tile $(0,2)$, can be sent to node 1. During the time step t=3, the CPUs of node 0 can continue the execution as above, while the CPUs of node 1 start executing the same routine with the rows of tiles $(\bullet,2)$ and $(\bullet,3)$.

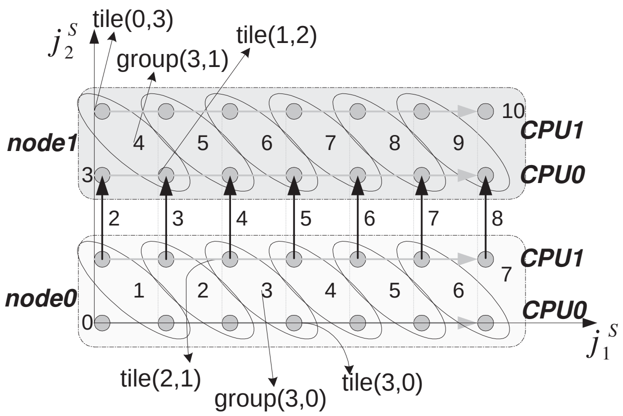


Figure 4.8: Example 4.1 - Tile space.

Grey dots correspond to tiles. Tiles along the same grey arrow will be executed by the same CPU during consecutive time steps. The grey rounded rectangles indicate which tiles will be executes by the CPUs of the same SMP node. The ovals indicate tiles that are grouped together and will be executed by different CPUs of the same node, during the same time step. The black arrows indicate dependences between tiles that will be executed in different SMP nodes and, thus, require a data transfer. The labels in the ovals-groups or besides black arrows-dependences indicate during which time step each group will be executed and each data transfer will take place, according to the overlapping execution policy.

In order to construct a time schedule for this example, we group together the tiles that should be concurrently executed by the same SMP node. In particular, we apply *grouping* to the tile space J^S, as indicated in Figure 4.8 and derive the group space J^G (Figure 4.9). The appropriate grouping matrices, according to formula (4.2), for this case are

$$P^G = \begin{bmatrix} 1 & -2 \\ 0 & 2 \end{bmatrix} \text{ and } H^G = (P^G)^{-1} = \begin{bmatrix} 1 & 1 \\ 0 & \frac{1}{2} \end{bmatrix}$$

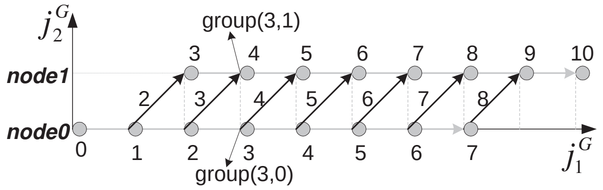


Figure 4.9: Example 4.1 - Group space.

Grey dots correspond to groups arising when applying the selected grouping transformation to the tile space of Figure 4.8. Groups along the same grey arrow will be executed in the same SMP node during consecutive time steps. As in Figure 4.8, the black arrows indicate dependences between groups that will be executed in different SMP nodes and, thus, require a data transfer. The labels besides the dots-groups or black arrows-dependences indicate during which time step each group will be executed and each data transfer will take place, according to the overlapping execution policy.

In this way, tiles $(1,0)$ and $(0,1)$ which, as we have already mentioned, are simultaneously executed by the same SMP node, are grouped together in $\vec{j^G} = \lfloor H^G(1,0)^T \rfloor = \lfloor H^G(0,1)^T \rfloor = (1,0)^T$. Similarly, tiles $(2,0)$ and $(1,1)$ are grouped together in $\vec{j^G} = (2,0)^T$. In Figures 4.8-4.9, the time step, when each group will be computed, is shown, together with the time step, when each data transfer will take place.

Table 4.1: Example 4.1

The columns labelled as "CPU x" indicate which tile will be executed by each CPU of an SMP node during each time step, according to the overlapping execution policy. The columns labelled as "group" indicate the group corresponding to the tiles executed by both CPUs of an SMP node at the same time.

Time	node 0			node 1		
Step	**CPU 0**	**CPU 1**	**group**	**CPU 0**	**CPU 1**	**group**
0	$\begin{pmatrix}0\\0\end{pmatrix}$		$\begin{pmatrix}0\\0\end{pmatrix}$			
1	$\begin{pmatrix}1\\0\end{pmatrix}$	$\begin{pmatrix}0\\1\end{pmatrix}$	$\begin{pmatrix}1\\0\end{pmatrix}$			
2	$\begin{pmatrix}2\\0\end{pmatrix}$	$\begin{pmatrix}1\\1\end{pmatrix}$	$\begin{pmatrix}2\\0\end{pmatrix}$			
3	$\begin{pmatrix}3\\0\end{pmatrix}$	$\begin{pmatrix}2\\1\end{pmatrix}$	$\begin{pmatrix}3\\0\end{pmatrix}$	$\begin{pmatrix}0\\2\end{pmatrix}$		$\begin{pmatrix}2\\1\end{pmatrix}$
4	$\begin{pmatrix}4\\0\end{pmatrix}$	$\begin{pmatrix}3\\1\end{pmatrix}$	$\begin{pmatrix}4\\0\end{pmatrix}$	$\begin{pmatrix}1\\2\end{pmatrix}$	$\begin{pmatrix}0\\3\end{pmatrix}$	$\begin{pmatrix}3\\1\end{pmatrix}$
5	$\begin{pmatrix}5\\0\end{pmatrix}$	$\begin{pmatrix}4\\1\end{pmatrix}$	$\begin{pmatrix}5\\0\end{pmatrix}$	$\begin{pmatrix}2\\2\end{pmatrix}$	$\begin{pmatrix}1\\3\end{pmatrix}$	$\begin{pmatrix}4\\1\end{pmatrix}$
6	$\begin{pmatrix}6\\0\end{pmatrix}$	$\begin{pmatrix}5\\1\end{pmatrix}$	$\begin{pmatrix}6\\0\end{pmatrix}$	$\begin{pmatrix}3\\2\end{pmatrix}$	$\begin{pmatrix}2\\3\end{pmatrix}$	$\begin{pmatrix}5\\1\end{pmatrix}$

In Table 4.1, we indicate the tiles of the tile space J^S that will be executed by each CPU of the first 2 SMP nodes during a time step and their corresponding group coordinates. It can be easily deduced that a group $\vec{j^G} = (j_1^G, j_2^G) \in J^G$ will be executed during the time step $t(\vec{j^G}) = j_1^G + j_2^G$ in the SMP node j_2^G. Therefore, the linear time scheduling vector for this example is $\Pi^G = (1, 1)$.

Example 4.2: In case the NICs of our cluster do not support DMA, then Example 4.1 should be modified as follows: During the time step t=0, CPU 0 of the SMP node 0 computes tile $(0, 0)$. During the time step $t = 1$, CPU 0 of node 0 computes tile $(1, 0)$, while CPU 1 of the same SMP node computes tile $(0, 1)$. Just when the computation of both tiles is completed, data needed for the computation of tile $(2, 0)$, which have just been computed in node 0 are transferred to node 1. During the time step $t = 2$, the CPUs of node 0 can continue the execution as above, while the CPUs of node 1 start executing the same routine with the rows of tiles $(\bullet, 2)$ and $(\bullet, 3)$.

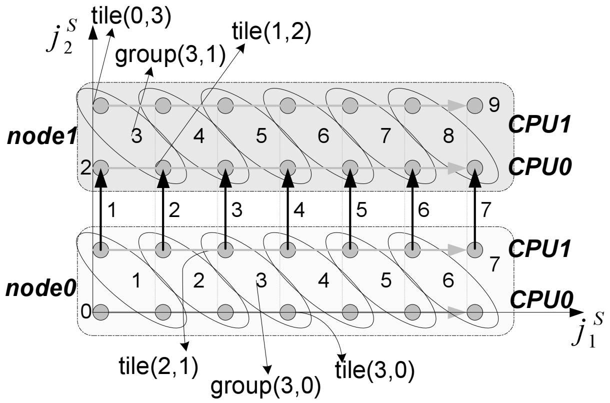


Figure 4.10: Example 4.2 - Tile space.
As in Figure 4.8, the labels in the ovals-groups or besides black arrows-dependences indicate during which time step each group will be executed and each data transfer will take place, according to the non-overlapping execution policy.

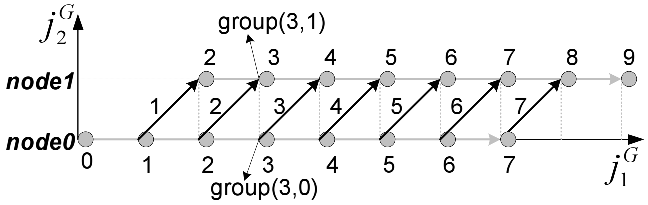


Figure 4.11: Example 4.2 - Group space.
As in Figure 4.9, the labels besides the dots-groups or black arrows-dependences indicate during which time step each group will be executed and each data transfer will take place, according to the non-overlapping execution policy.

In order to construct a time schedule for this example, as in Example 4.1, we group together the tiles that should be concurrently executed by the same SMP node. In particular, we apply *grouping* to the tile space J^S, as indicated in Figure 4.10 and derive the group space

J^G (Figure 4.11). The grouping matrices are identical to the ones used in Example 4.1. In Figures 4.10-4.11, the time step, when each group will be computed, is shown, together with the time step, when each data transfer will take place.

Table 4.2: Example 4.2
As in Table 4.1, the columns labelled as "CPU x" indicate which tile will be executed by each CPU of an SMP node during each time step, according to the non-overlapping execution policy. The columns labelled as "group" indicate the group corresponding to the tiles executed by both CPUs of an SMP node at the same time.

Time Step	node 0			node 1		
	CPU 0	CPU 1	group	CPU 0	CPU 1	group
0	$\begin{pmatrix}0\\0\end{pmatrix}$		$\begin{pmatrix}0\\0\end{pmatrix}$			
1	$\begin{pmatrix}1\\0\end{pmatrix}$	$\begin{pmatrix}0\\1\end{pmatrix}$	$\begin{pmatrix}1\\0\end{pmatrix}$			
2	$\begin{pmatrix}2\\0\end{pmatrix}$	$\begin{pmatrix}1\\1\end{pmatrix}$	$\begin{pmatrix}2\\0\end{pmatrix}$	$\begin{pmatrix}0\\2\end{pmatrix}$		$\begin{pmatrix}2\\1\end{pmatrix}$
3	$\begin{pmatrix}3\\0\end{pmatrix}$	$\begin{pmatrix}2\\1\end{pmatrix}$	$\begin{pmatrix}3\\0\end{pmatrix}$	$\begin{pmatrix}1\\2\end{pmatrix}$	$\begin{pmatrix}0\\3\end{pmatrix}$	$\begin{pmatrix}3\\1\end{pmatrix}$
4	$\begin{pmatrix}4\\0\end{pmatrix}$	$\begin{pmatrix}3\\1\end{pmatrix}$	$\begin{pmatrix}4\\0\end{pmatrix}$	$\begin{pmatrix}2\\2\end{pmatrix}$	$\begin{pmatrix}1\\3\end{pmatrix}$	$\begin{pmatrix}4\\1\end{pmatrix}$
3	$\begin{pmatrix}5\\0\end{pmatrix}$	$\begin{pmatrix}4\\1\end{pmatrix}$	$\begin{pmatrix}5\\0\end{pmatrix}$	$\begin{pmatrix}3\\2\end{pmatrix}$	$\begin{pmatrix}2\\3\end{pmatrix}$	$\begin{pmatrix}5\\1\end{pmatrix}$

In Table 4.2, we indicate the tiles of the tile space J^S that will be executed by each CPU of the first 2 SMP nodes during a time step and their corresponding group coordinates. It can be easily deduced that a group $\vec{j^G} = (j_1^G, j_2^G) \in J^G$ will be executed during the time step $t(\vec{j^G}) = j_1^G$ in the SMP node j_2^G. Therefore, the linear time scheduling vector for this example is $\Pi^G = (1, 0)$. Thus, we may equivalently schedule tiles, instead of groups, using the linear time scheduling vector $\Pi = (1, 1)$.

4.4.1 Linear time schedule

Theorem 4.2 *When applying the overlapping execution policy, the appropriate linear time scheduling vector for the group space derived by grouping, as defined in formula (4.2), is* $\Pi^G = (1, 1, \ldots, 1)$.

Proof: *Applying the grouping transformation defined by formula (4.2), the* 1*-st column-vector of the dependence matrix* $D^S = I$ *is transformed to the vector* $\vec{d_1^{G}}' = H^G \vec{d_1^S} = (1, 0, \ldots, 0)^T$. *In addition, the* j*-th column-vector of the dependence matrix* $D^S = I$, $j = 2, \ldots, n$, *is transformed to the vector*

$$H^G \vec{d_j^S} = (1, 0, \ldots, 0, \frac{1}{m_j}, 0, \ldots, 0)^T.$$

Thus, it imposes group dependences

$$(1,0,\ldots,0,\lfloor\frac{1}{m_i}\rfloor,0,\ldots,0)^T = (1,0,\ldots,0,0,0,\ldots,0)^T$$

and

$$(1,0,\ldots,0,\lceil\frac{1}{m_j}\rceil,0,\ldots,0)^T = (1,0,\ldots,0,1,0,\ldots,0)^T$$

Thus, the dependence matrix of the group space can be written as:

$$D^G = \begin{pmatrix} 1 & 1 & \ldots & 1 & 1 \\ 0 & 1 & \ldots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \ldots & 1 & 0 \\ 0 & 0 & \ldots & 0 & 1 \end{pmatrix}.$$

We are searching for an appropriate linear time scheduling vector $\Pi^G = (\pi_1^G,\ldots,\pi_n^G)$ *such that each group* $\vec{j^G} \in J^G$ *is computed during the time step* $t = \Pi^G \vec{j^G}$. *Consider the last* $(n-1)$ *coordinates of a group indicating which SMP node of the cluster will execute this group. Then, groups* $\vec{j^G} = (j_1^G,\ldots,j_n^G)$ *and* $\vec{j^{G'}} = (j_1^G+1, j_2^G,\ldots,j_n^G)$ *will be successively computed within the same SMP node. There is a dependence between them, as indicated by the first column of* D^G*, but there is no need for a communication step between their successive computation steps, because the necessary data are already located in the local shared memory of the SMP node. Consequently, their time distance* $\Pi^G\vec{j^{G'}} - \Pi^G\vec{j^G} = \pi_1^G$ *may be equal to* 1. *Thus,* $\pi_1^G = 1$. *In addition, the i-th column of* D^G $(i = 2,\ldots n)$ *imposes a dependence between groups* $\vec{j^G} = (j_1^G,\ldots,j_n^G)$ *and* $\vec{j^{G'}} = (j_1^G+1, j_2^G,\ldots,j_{i-1}^G, j_i^G+1, j_{i+1}^G,\ldots,j_n^G)$. *These groups are executed in neighboring SMP nodes, thus a communication step is required between their computation steps. It means that their time distance* $\Pi^G\vec{j^{G'}} - \Pi^G\vec{j^G} = \pi_1^G + \pi_i^G$ *must be equal to* 2. *Consequently,* $\pi_i^G = 1$, $i = 2,\ldots,n$. *So, the vector* $\Pi^G = (1,1,\ldots,1)$ *is selected for the linear time scheduling of our group space* J^G. ⊣

Notice that, in [GSK01], [STK02], for the single CPU pipelined schedule, Π was $(1,2,\ldots,2)$ according to the UET-UCT theory [AKPT99]. In other words, the optimal overlapping schedule could be achieved when we had equal computation to communication times, so that all communication could be hidden (overlapped) with the computation phase. Nevertheless, in the SMP case presented here, the labeling of coordinates of groups, that is the grouping transformation P^G, slightly skews the space (see Figure 4.8 and the resulting group space in Figure 4.9, the relative positions of groups $(3,0)$ and $(3,1)$). So the optimal overlapping schedule is achieved by $(1,1,\ldots,1)$. Notice, also, that this scheduling vector is not the same with Hodzic's [HS98] scheduling vector, since we are now referring to groups, while Hodzic was scheduling tiles.

Theorem 4.3 *When applying the non-overlapping execution policy, the appropriate linear time scheduling vector for the group space derived by grouping, as defined in formula (4.2), is* $\Pi^G = (1,0,\ldots,0)$.

Proof: *As in the proof of Theorem 4.2, the dependence matrix of the group space is:*

$$D^G = \begin{pmatrix} 1 & 1 & \dots & 1 & 1 \\ 0 & 1 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 1 & 0 \\ 0 & 0 & \dots & 0 & 1 \end{pmatrix}.$$

We are searching, again, for an appropriate linear time scheduling vector $\Pi^G = (\pi_1^G, \dots, \pi_n^G)$ *such that each group* $\vec{j^G} \in J^G$ *is computed during the time step* $t = \Pi^G \vec{j^G}$. *Consider the last* $(n-1)$ *coordinates of a group indicating which SMP node of the cluster will execute this group. Then, groups* $\vec{j^G} = (j_1^G, \dots, j_n^G)$ *and* $\vec{j^G}' = (j_1^G + 1, j_2^G, \dots, j_n^G)$ *will be successively computed within the same SMP node. Consequently, their time distance* $\Pi^G \vec{j^G}' - \Pi^G \vec{j^G} = \pi_1^G$ *may be equal to* 1. *Thus* $\pi_1^G = 1$. *In addition, the* i*-th column of* D^G $(i = 2, \dots n)$ *imposes a dependence between groups* $\vec{j^G} = (j_1^G, \dots, j_n^G)$ *and* $\vec{j^G}' = (j_1^G + 1, j_2^G, \dots, j_{i-1}^G, j_i^G + 1, j_{i+1}^G, \dots, j_n^G)$. *These groups are executed in neighboring SMP nodes, thus a data transfer should take place between the respective computations. In contrast to the overlapping execution policy, this data transfer may take place during the time step, when data are computed, just after the completion of computation. Thus, their time distance* $\Pi^G \vec{j^G}' - \Pi^G \vec{j^G} = \pi_1^G + \pi_i^G$ *may be equal to* 1. *Consequently,* $\pi_i^G = 0$, $i = 2, \dots, n$. *So, the vector* $\Pi^G = (1, 0, \dots, 0)$ *is selected for the linear time scheduling of our group space* J^G. ⊣

As in Example 4.2, notice that linear scheduling of groups, using vector $\Pi^G = (1, 0, \dots, 0)$, is equivalent to linear scheduling of tiles, using vector $\Pi = (1, 1, \dots, 1)$. Thus, the only reasons for grouping tiles, when an overlapping execution is not possible, or not desired, are

1. comparison with the overlapping execution
2. emphasizing the fact that data originating in the same group, albeit in different tiles, may be transferred in a single message.

Example 4.3: Consider a rectangular n-dimensional tile space J^S: $0 \leq j_i^S \leq u_i^S$, $i = 1, \dots, n$ and $u_1^S \geq u_i^S, i = 2, \dots, n$. We apply grouping transformation, according to the formula (4.2). Thus, tile $\vec{j^S}$ belongs to group $\vec{j^G} = (\sum_{i=1}^{n} j_i^S, \lfloor \frac{j_2^S}{m_2} \rfloor, \dots, \lfloor \frac{j_n^S}{m_n} \rfloor)^T$.

According to the overlapping execution policy, it will be executed during the time step $t(\vec{j^G}) = \sum_{i=1}^{n} j_i^G = \sum_{i=1}^{n} j_i^S + \sum_{i=2}^{n} \lfloor \frac{j_i^S}{m_i} \rfloor$ (according to the linear time scheduling vector $\Pi^G = (1, 1, \dots, 1)$). Group $(0, 0, 0)$ will be executed during the first time step $t_{min} = 0$. Group $(\sum_{i=1}^{n} u_i^S, \lfloor \frac{u_2^S}{m_2} \rfloor, \dots, \lfloor \frac{u_n^S}{m_n} \rfloor)$ will be computed during the last time step $t_{max} = \sum_{i=1}^{n} u_i^S + \sum_{i=2}^{n} \lfloor \frac{u_i^S}{m_i} \rfloor$. Thus, the number of time steps required for the completion of the execution (makespan), is:

$$\wp_{overlap} = 1 + t_{max} - t_{min} = \sum_{i=1}^{n} u_i^S + \sum_{i=2}^{n} \lfloor \frac{u_i^S}{m_i} \rfloor + 1 = \sum_{i=1}^{n} (w_i^S - 1) + \sum_{i=2}^{n} \lfloor \frac{w_i^S - 1}{m_i} \rfloor + 1 \overset{(C.4)}{\Rightarrow}$$

$$\wp_{overlap} = \sum_{i=1}^{n} w_i^S + \sum_{i=2}^{n} \lceil \frac{w_i^S}{m_i} \rceil - 2n + 2 \tag{4.3}$$

where $w_i^s = u_i^s + 1$, $i = 1, \ldots, n$ is the width of the tile space along dimension i.

Similarly, following the non-overlapping execution policy, group

$$\vec{j^G} = (\sum_{i=1}^{n} j_i^S, \lfloor \frac{j_2^S}{m_2} \rfloor, \ldots, \lfloor \frac{j_n^S}{m_n} \rfloor)^T$$

will be executed during the time step $t(\vec{j^G}) = j_1^G = \sum_{i=1}^{n} j_i^S$ (according to the linear time scheduling vector $\Pi^G = (1, 0, \ldots, 0)$). Group $(0, 0, 0)$ will be executed during the first time step $t_{min} = 0$. Group $(\sum_{i=1}^{n} u_i^S, \lfloor \frac{u_2^S}{m_2} \rfloor, \ldots, \lfloor \frac{u_n^S}{m_n} \rfloor)$ will be computed during the last time step $t_{max} = \sum_{i=1}^{n} u_i^S$. Thus, the number of time steps required for the completion of the execution (makespan), is: $\wp_{nonoverlap} = 1 + t_{max} - t_{min} = \sum_{i=1}^{n} u_i^S + 1 \Rightarrow$

$$\wp_{nonoverlap} = \sum_{i=1}^{n} w_i^S - n + 1 \tag{4.4}$$

4.4.2 Assigning Tiles to CPUs

For node labelling reasons, consider that the available SMP nodes form a virtual $(n-1)$-dimensional mesh. Thus, each node is identified by a $(n-1)$-dimensional vector. Note, however, that it is not a physical layout restriction, but a convention to give each node a unique tag. Then, the last $(n-1)$ coordinates of a group indicate the SMP into which it will be executed. The first coordinate affects only the time of its execution. Thus, a tile $\vec{j^S} = (j_1^S, \ldots, j_n^S)$, belonging to group $\vec{j^G} = (j_1^G, \ldots, j_n^G)$, will be executed in node $(j_2^G, \ldots, j_n^G) = (\lfloor \frac{j_2^S}{m_2} \rfloor, \ldots, \lfloor \frac{j_n^S}{m_n} \rfloor)$.

Similarly, inside each SMP node we consider a $(n-1)$-dimensional CPU virtual mesh containing labels $\{\vec{cpu} \in Z^{n-1} | 0 \leq cpu_x < m_{x+1}, 1 \leq x \leq n-1\}$. Then, a tile $\vec{j^S} = (j_1^S, \ldots, j_n^S)$ will be executed by CPU $(j_2^S \% m_2, \ldots, j_n^S \% m_n)$ of SMP node $(\lfloor \frac{j_2^S}{m_2} \rfloor, \ldots, \lfloor \frac{j_n^S}{m_n} \rfloor)$. So, apparently, only tiles with the same coordinate j_1^S will be assigned to the same CPU of the same node.

In addition, note that, if one of the diagonal elements of the inverse grouping matrix m_x equals to 1, then the corresponding coordinate of the CPU identification vector can be omitted, as it will always equal 0.

4.4.3 Generalization: Grouping tiles along an arbitrary dimension of J^S

If we want to assign the iterations along the i-th dimension of J^S to the same CPU of an SMP node, then it can be similarly proven that the appropriate grouping matrices are

$$
\begin{aligned}
P^G &= \begin{bmatrix}
m_1 & \dots & 0 & 0 & 0 & \dots & 0 \\
\vdots & \ddots & \vdots & \vdots & \vdots & & \vdots \\
0 & \dots & m_{i-1} & 0 & 0 & \dots & 0 \\
-m_1 & \dots & -m_{i-1} & 1 & -m_{i+1} & \dots & -m_n \\
0 & \dots & 0 & 0 & m_{i+1} & \dots & 0 \\
\vdots & & \vdots & \vdots & \vdots & \ddots & \vdots \\
0 & \dots & 0 & 0 & 0 & \dots & m_n
\end{bmatrix} \\
H^G = (P^G)^{-1} &= \begin{bmatrix}
\frac{1}{m_1} & \dots & 0 & 0 & 0 & \dots & 0 \\
\vdots & \ddots & \vdots & \vdots & \vdots & & \vdots \\
0 & \dots & \frac{1}{m_{i-1}} & 0 & 0 & \dots & 0 \\
1 & \dots & 1 & 1 & 1 & \dots & 1 \\
0 & \dots & 0 & 0 & \frac{1}{m_{i+1}} & \dots & 0 \\
\vdots & & \vdots & \vdots & \vdots & \ddots & \vdots \\
0 & \dots & 0 & 0 & 0 & \dots & \frac{1}{m_n}
\end{bmatrix}
\end{aligned}
\tag{4.5}
$$

where $m_1 \times \dots \times m_{i-1} \times m_{i+1} \times \dots \times m_n = m$. As previously, the time scheduling vector is $\Pi^G = (1, \dots, 1)$ if the overlapping execution policy is followed, or $\Pi^G = (0, \dots, 0, 1, 0, \dots, 0)$ otherwise. In addition, tile $\vec{j^S} = (j_1^S, \dots, j_n^S)$ belonging to group $\vec{j^G} = (j_1^G, \dots, j_n^G)$, will be executed within node $(j_1^G, \dots, j_{i-1}^G, j_{i+1}^G, \dots, j_n^G)$ by CPU $(j_1^S \% m_1, \dots, j_{i-1}^S \% m_{i-1}, j_{i+1}^S \% m_{i+1}, \dots, j_n^S \% m_n)$. As previously, if one of the diagonal elements of the inverse grouping matrix $m_x = 1, x \neq i$, then the corresponding coordinate of the CPU identification vector can be omitted.

Example 4.4: We have a cluster of SMP nodes with 2 CPUs and a NIC each. We assume a 3-dimensional rectangular tile space J^S. Let us assign the tiles along dimension j_3^S to the same CPU, as indicated in Figure 4.12 by the grey arrows. The CPUs of the same SMP node will execute two neighboring rows of tiles, which belong to the same $j_1^S - j_3^S$ plane. In respect to the formula (4.5), we choose the grouping matrices to be:

$$P^G = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ -2 & -1 & 1 \end{bmatrix} \text{ and } H^G = (P^G)^{-1} = \begin{bmatrix} \frac{1}{2} & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}.$$

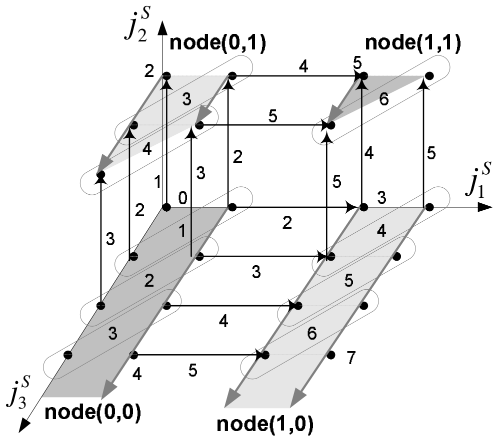


Figure 4.12: Example 4.4 - 2×1 CPUs per SMP node - Overlapping execution.
All tiles along the same grey arrow will be executed by the same CPU during consecutive time steps. The grey areas indicate which tiles will be executes by the CPUs of the same SMP node. The ovals indicate tiles that are grouped together and will be executed by different CPUs of the same node, during the same time step. The black arrows indicate dependences between tiles that will be executed in different SMP nodes and, thus, require a data transfer. The labels in the ovals-groups or besides black arrows-dependences indicate during which time step each group will be executed and each data transfer will take place, according to the overlapping execution policy.

In Figure 4.12 we show the grouping of tiles and when each computation step and each communication step will take place, according to the overlapping execution policy. In Table 4.3, we indicate the tiles of J^S that will be executed by each CPU of the 3 neighboring SMP nodes $(0,1)$, $(0,0)$, $(1,0)$ during each time step. It can be easily deduced that a group $(j_1^G, j_2^G, j_3^G) \in J^G$ will be executed in node (j_1^G, j_2^G) during the time step $t(\vec{j^G}) = j_1^G + j_2^G + j_3^G$. Therefore, as expected, the linear time scheduling vector for this example is $\Pi^G = (1,1,1)$.

Similarly, in Figure 4.13, we show the grouping of tiles and when each computation step and each communication step will be executed, according to the non-overlapping execution policy. In Table 4.4, we indicate the tiles of J^S that will be executed by each CPU of the 3 neighboring SMP nodes $(0,1)$, $(0,0)$, $(1,0)$ during each time step. It can be easily deduced that a group $(j_1^G, j_2^G, j_3^G) \in J^G$ will be executed in node (j_1^G, j_2^G) during the time step $t(\vec{j^G}) = j_3^G$. Therefore, as expected, the linear time scheduling vector for this example is $\Pi^G = (0,0,1)$.

Table 4.3: Example 4.4 - 2×1 CPUs per SMP node - Overlapping execution

Time Step	node (0,1) CPU 0	node (0,1) CPU 1	node (0,1) group	node (0,0) CPU 0	node (0,0) CPU 1	node (0,0) group	node (1,0) CPU 0	node (1,0) CPU 1	node (1,0) group
0				$\begin{pmatrix}0\\0\\0\end{pmatrix}$		$\begin{pmatrix}0\\0\\0\end{pmatrix}$			
1				$\begin{pmatrix}0\\0\\1\end{pmatrix}$	$\begin{pmatrix}1\\0\\0\end{pmatrix}$	$\begin{pmatrix}0\\0\\1\end{pmatrix}$			
2	$\begin{pmatrix}0\\1\\0\end{pmatrix}$		$\begin{pmatrix}0\\1\\1\end{pmatrix}$	$\begin{pmatrix}0\\0\\2\end{pmatrix}$	$\begin{pmatrix}1\\0\\1\end{pmatrix}$	$\begin{pmatrix}0\\0\\2\end{pmatrix}$			
3	$\begin{pmatrix}0\\1\\1\end{pmatrix}$	$\begin{pmatrix}1\\1\\0\end{pmatrix}$	$\begin{pmatrix}0\\1\\2\end{pmatrix}$	$\begin{pmatrix}0\\0\\3\end{pmatrix}$	$\begin{pmatrix}1\\0\\2\end{pmatrix}$	$\begin{pmatrix}0\\0\\3\end{pmatrix}$	$\begin{pmatrix}2\\0\\0\end{pmatrix}$		$\begin{pmatrix}1\\0\\2\end{pmatrix}$
4	$\begin{pmatrix}0\\1\\2\end{pmatrix}$	$\begin{pmatrix}1\\1\\1\end{pmatrix}$	$\begin{pmatrix}0\\1\\3\end{pmatrix}$	$\begin{pmatrix}0\\0\\4\end{pmatrix}$	$\begin{pmatrix}1\\0\\3\end{pmatrix}$	$\begin{pmatrix}0\\0\\4\end{pmatrix}$	$\begin{pmatrix}2\\0\\1\end{pmatrix}$	$\begin{pmatrix}3\\0\\0\end{pmatrix}$	$\begin{pmatrix}1\\0\\3\end{pmatrix}$
5	$\begin{pmatrix}0\\1\\3\end{pmatrix}$	$\begin{pmatrix}1\\1\\2\end{pmatrix}$	$\begin{pmatrix}0\\1\\4\end{pmatrix}$	$\begin{pmatrix}0\\0\\5\end{pmatrix}$	$\begin{pmatrix}1\\0\\4\end{pmatrix}$	$\begin{pmatrix}0\\0\\5\end{pmatrix}$	$\begin{pmatrix}2\\0\\2\end{pmatrix}$	$\begin{pmatrix}3\\0\\1\end{pmatrix}$	$\begin{pmatrix}1\\0\\4\end{pmatrix}$

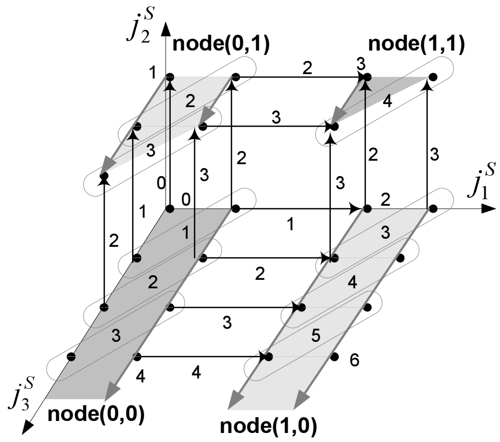


Figure 4.13: Example 4.4 - 2×1 CPUs per SMP node - Non-overlapping execution

Example 4.5: We have a cluster of SMP nodes with 4 CPUs and a NIC each. As previously, we assume a 3-dimensional rectangular tile space J^S. Let us assign the tiles along dimension j_3^S to the same CPU, as indicated in Figure 4.14 by the grey arrows. The CPUs of the same SMP node will undertake 4 neighboring lines of tiles which belong to the same $j_1^S - j_3^S$ plane.

According to formula (4.5), we choose the grouping matrices to be

$$P^G = \begin{bmatrix} 4 & 0 & 0 \\ 0 & 1 & 0 \\ -4 & -1 & 1 \end{bmatrix} \text{ and } H^G = (P^G)^{-1} = \begin{bmatrix} \frac{1}{4} & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

In Figure 4.14 we indicate the grouping of tiles and during which time step each computation

Table 4.4: Example 4.4 - 2×1 CPUs per SMP node - Non-overlapping execution

Time Step	node (0,1)			node (0,0)			node (1,0)		
	CPU 0	CPU 1	group	CPU 0	CPU 1	group	CPU 0	CPU 1	group
0				$\begin{pmatrix}0\\0\\0\end{pmatrix}$		$\begin{pmatrix}0\\0\\0\end{pmatrix}$			
1	$\begin{pmatrix}0\\1\\0\end{pmatrix}$		$\begin{pmatrix}0\\1\\1\end{pmatrix}$	$\begin{pmatrix}0\\0\\1\end{pmatrix}$	$\begin{pmatrix}1\\0\\0\end{pmatrix}$	$\begin{pmatrix}0\\0\\1\end{pmatrix}$			
2	$\begin{pmatrix}0\\1\\1\end{pmatrix}$	$\begin{pmatrix}1\\1\\0\end{pmatrix}$	$\begin{pmatrix}0\\1\\2\end{pmatrix}$	$\begin{pmatrix}0\\0\\2\end{pmatrix}$	$\begin{pmatrix}1\\0\\1\end{pmatrix}$	$\begin{pmatrix}0\\0\\2\end{pmatrix}$	$\begin{pmatrix}2\\0\\0\end{pmatrix}$		$\begin{pmatrix}1\\0\\2\end{pmatrix}$
3	$\begin{pmatrix}0\\1\\2\end{pmatrix}$	$\begin{pmatrix}1\\1\\1\end{pmatrix}$	$\begin{pmatrix}0\\1\\3\end{pmatrix}$	$\begin{pmatrix}0\\0\\3\end{pmatrix}$	$\begin{pmatrix}1\\0\\2\end{pmatrix}$	$\begin{pmatrix}0\\0\\3\end{pmatrix}$	$\begin{pmatrix}2\\0\\1\end{pmatrix}$	$\begin{pmatrix}3\\0\\0\end{pmatrix}$	$\begin{pmatrix}1\\0\\3\end{pmatrix}$
4	$\begin{pmatrix}0\\1\\3\end{pmatrix}$	$\begin{pmatrix}1\\1\\2\end{pmatrix}$	$\begin{pmatrix}0\\1\\4\end{pmatrix}$	$\begin{pmatrix}0\\0\\4\end{pmatrix}$	$\begin{pmatrix}1\\0\\3\end{pmatrix}$	$\begin{pmatrix}0\\0\\4\end{pmatrix}$	$\begin{pmatrix}2\\0\\2\end{pmatrix}$	$\begin{pmatrix}3\\0\\1\end{pmatrix}$	$\begin{pmatrix}1\\0\\4\end{pmatrix}$

step and each communication step will take place, following the overlapping execution policy. In Table 4.5 we indicate which tiles of the tile space J^S will be executed by each CPU of the first 3 SMP nodes of our cluster during a time step. In addition, we indicate which is the corresponding group of J^G. It can be easily deduced from Table 4.5 that a group $(j_1^G, j_2^G, j_3^G) \in J^G$ will be executed in the SMP node (j_1^G, j_2^G) during the time step $t(\vec{j^G}) = j_1^G + j_2^G + j_3^G$. Therefore, the linear time scheduling vector for this example is $\Pi^G = (1,1,1)$.

Similarly, in Figure 4.15 we indicate the grouping of tiles and during which time step each computation step and each communication step will take place, following the non-overlapping execution policy. In Table 4.6 we indicate which tiles of the tile space J^S will be executed by each CPU of the first 3 SMP nodes of our cluster during a time step. In addition, we indicate which is the corresponding group of J^G. It can be easily deduced from Table 4.6 that a group $(j_1^G, j_2^G, j_3^G) \in J^G$ will be executed in the SMP node (j_1^G, j_2^G) during the time step $t(\vec{j^G}) = j_3^G = j_1^S + j_2^S + j_3^S$. Therefore, the linear time scheduling vector for this example is $\Pi^G = (0,0,1)$.

Example 4.6: We have a cluster of SMP nodes with 4 CPUs and a NIC each. As previously, we assume a 3-dimensional rectangular tile space J^S. The CPUs of the same SMP node undertake 4 neighboring lines of tiles whose projection on the $j_1^S - j_2^S$ plane forms a square. Thus,

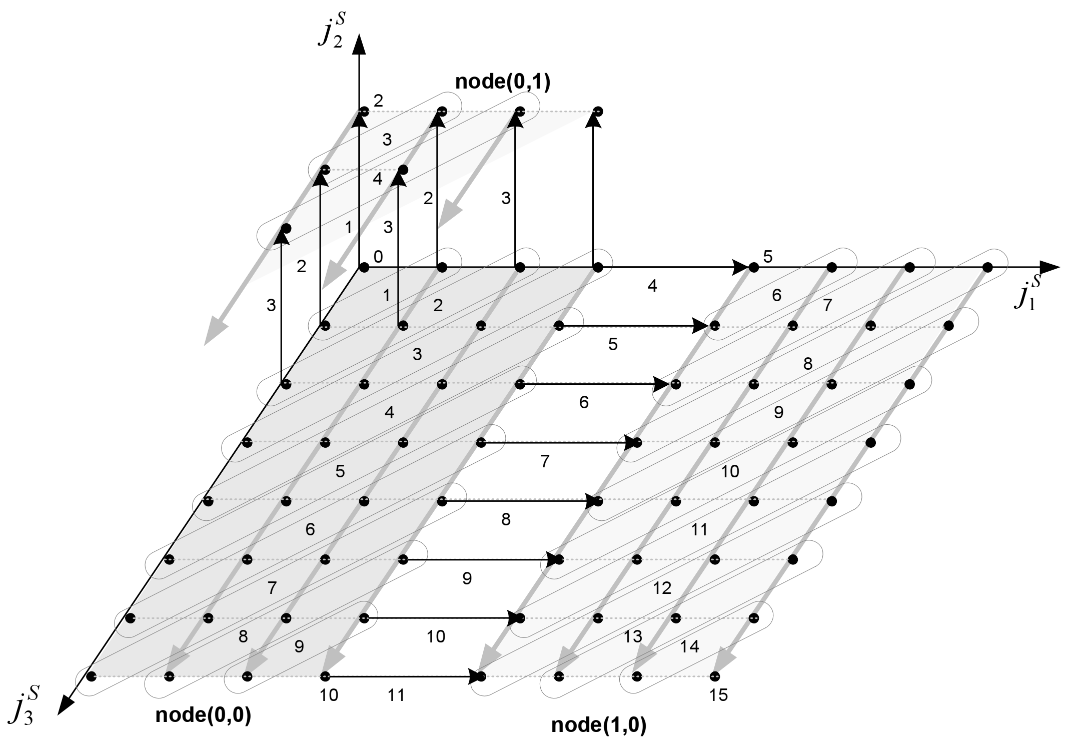


Figure 4.14: Example 4.5 - 4×1 CPUs per SMP node - Overlapping execution.
As in Figure 4.12, the labels in the ovals-groups or besides black arrows-dependences indicate during which time step each group will be executed and each data transfer will take place, according to the overlapping execution policy.

according to formula (4.5), the grouping matrices are

$$P^G = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ -2 & -2 & 1 \end{bmatrix} \text{ and } H^G = (P^G)^{-1} = \begin{bmatrix} \frac{1}{2} & 0 & 0 \\ 0 & \frac{1}{2} & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

In Figure 4.16, we indicate which tiles of J^S will be undertaken by each SMP node. In Figure 4.17, we have zoomed to the part of J^S assigned to an SMP node and we indicate which tiles of this part will be executed simultaneously by different CPUs. These tiles belong to the same grey plane. In Table 4.7 we indicate which tiles of the tile space J^S will be executed by each CPU of the first 3 SMP nodes of our cluster during a time step, following the overlapping execution policy. In addition, we indicate which is the corresponding group of J^G. As in Examples 4.4 and 4.5, it can be deduced that a group $(j_1^G, j_2^G, j_3^G) \in J^G$ will be executed in SMP node (j_1^G, j_2^G) during the time step $t(\vec{j^G}) = j_1^G + j_2^G + j_3^G$. Therefore, the linear time scheduling vector for this example is $\Pi^G = (1, 1, 1)$.

Similarly, in Table 4.8 we indicate which tiles of the tile space J^S will be executed by each CPU of the first 3 SMP nodes of our cluster during a time step, following the non-overlapping

Table 4.5: Example 4.5 - 4×1 CPUs per SMP node - Overlapping execution.
Since CPUs inside an SMP node form a 4×1 mesh, we have omitted the second dimension when labelling CPUs. It would be always equal to 0, as explained in page 102.

Time Step	node (0,0) CPU 0	CPU 1	CPU 2	CPU 3	group
0	$\begin{pmatrix}0\\0\\0\end{pmatrix}$				$\begin{pmatrix}0\\0\\0\end{pmatrix}$
1	$\begin{pmatrix}0\\0\\1\end{pmatrix}$	$\begin{pmatrix}1\\0\\0\end{pmatrix}$			$\begin{pmatrix}0\\0\\1\end{pmatrix}$
2	$\begin{pmatrix}0\\0\\2\end{pmatrix}$	$\begin{pmatrix}1\\0\\1\end{pmatrix}$	$\begin{pmatrix}2\\0\\0\end{pmatrix}$		$\begin{pmatrix}0\\0\\2\end{pmatrix}$
3	$\begin{pmatrix}0\\0\\3\end{pmatrix}$	$\begin{pmatrix}1\\0\\2\end{pmatrix}$	$\begin{pmatrix}2\\0\\1\end{pmatrix}$	$\begin{pmatrix}3\\0\\0\end{pmatrix}$	$\begin{pmatrix}0\\0\\3\end{pmatrix}$
4	$\begin{pmatrix}0\\0\\4\end{pmatrix}$	$\begin{pmatrix}1\\0\\3\end{pmatrix}$	$\begin{pmatrix}2\\0\\2\end{pmatrix}$	$\begin{pmatrix}3\\0\\1\end{pmatrix}$	$\begin{pmatrix}0\\0\\4\end{pmatrix}$
5	$\begin{pmatrix}0\\0\\5\end{pmatrix}$	$\begin{pmatrix}1\\0\\4\end{pmatrix}$	$\begin{pmatrix}2\\0\\3\end{pmatrix}$	$\begin{pmatrix}3\\0\\2\end{pmatrix}$	$\begin{pmatrix}0\\0\\5\end{pmatrix}$
Time Step	**node (0,1) CPU 0**	**CPU 1**	**CPU 2**	**CPU 3**	**group**
2	$\begin{pmatrix}0\\1\\0\end{pmatrix}$				$\begin{pmatrix}0\\1\\1\end{pmatrix}$
3	$\begin{pmatrix}0\\1\\1\end{pmatrix}$	$\begin{pmatrix}1\\1\\0\end{pmatrix}$			$\begin{pmatrix}0\\1\\2\end{pmatrix}$
4	$\begin{pmatrix}0\\1\\2\end{pmatrix}$	$\begin{pmatrix}1\\1\\1\end{pmatrix}$	$\begin{pmatrix}2\\1\\0\end{pmatrix}$		$\begin{pmatrix}0\\1\\3\end{pmatrix}$
5	$\begin{pmatrix}0\\1\\3\end{pmatrix}$	$\begin{pmatrix}1\\1\\2\end{pmatrix}$	$\begin{pmatrix}2\\1\\1\end{pmatrix}$	$\begin{pmatrix}3\\1\\0\end{pmatrix}$	$\begin{pmatrix}0\\1\\4\end{pmatrix}$
Time Step	**node (1,0) CPU 0**	**CPU 1**	**CPU 2**	**CPU 3**	**group**
5	$\begin{pmatrix}4\\0\\0\end{pmatrix}$				$\begin{pmatrix}1\\0\\4\end{pmatrix}$

execution policy. Once again, it can be deduced that a group $(j_1^G, j_2^G, j_3^G) \in J^G$ will be executed in SMP node (j_1^G, j_2^G) during the time step $t(\vec{j^G}) = j_3^G = j_1^S + j_2^S + j_3^S$. Therefore, the linear time scheduling vector for this example is $\Pi^G = (0, 0, 1)$.

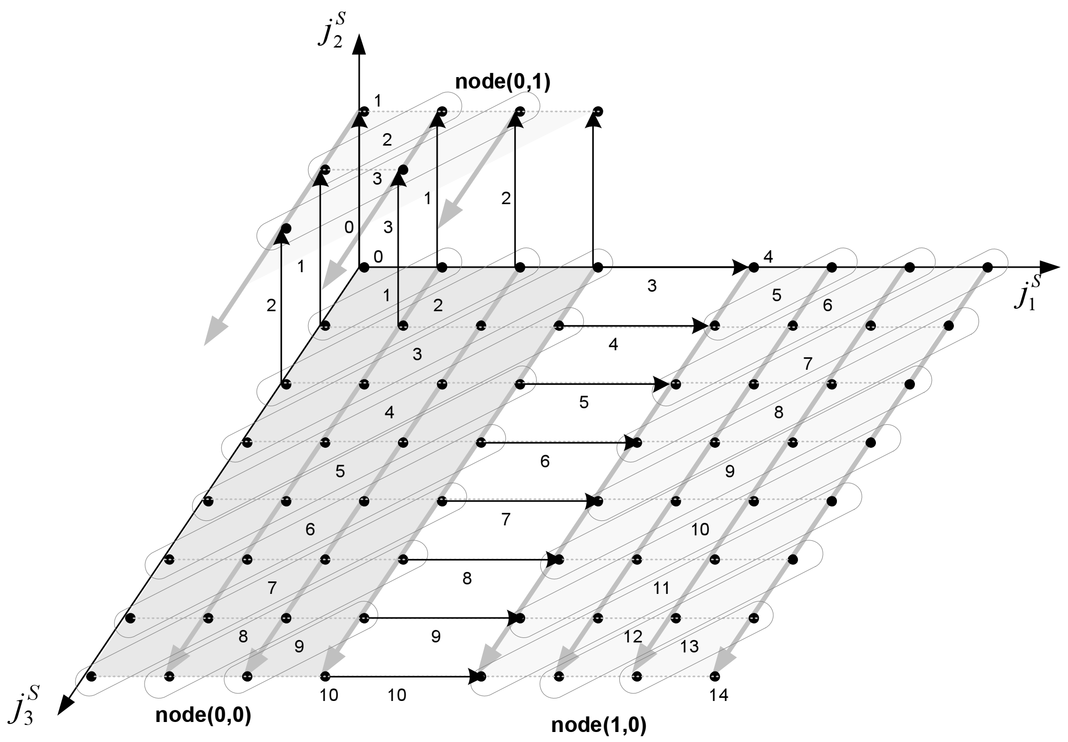


Figure 4.15: Example 4.5 - 4×1 CPUs per SMP node - Non-overlapping execution

4.4.4 Optimal selection of m_ks

Considering the minimization of the makespan

Let us consider (as in Example 4.3) a rectangular tile space J^S: $\forall j^S \in J^S$ it holds $0 \leq j_i^S \leq u_i^S$, $0 \leq i \leq n$. We apply grouping transformation, according to formula (4.5). Similarly to formula (4.3), it can be proven that the makespan of the execution will be

$$\wp_{overlap} = \sum_{k=1}^{n} w_k^S + \sum_{k \neq i} \lceil \frac{w_k^S}{m_k} \rceil - 2n + 2 \tag{4.6}$$

where $w_i^s = u_i^s + 1$, $i = 1, \ldots, n$ is the width of the tile space along dimension i.

In order to minimize the total completion time, we should apparently choose the i-th dimension, along which we allocate the tiles to the same CPU, so that it holds $w_i^S \geq w_k^S, \forall k = 1, \ldots, n$, as w_i^S is the only dimension of J^S which is involved in (4.6) only once.

After the selection of the i-th dimension, the ceiling functions involved in the expression (4.6) can be eliminated as follows:

$$\sum_{k=1}^{n} w_k^S + \sum_{k \neq i} \frac{w_k^S}{m_k} - 2n + 2 \leq \wp_{overlap} < \sum_{k=1}^{n} w_k^S + \sum_{k \neq i} \frac{w_k^S}{m_k} - n + 1$$

Thus, we can assert that the completion time of the algorithm is approximately minimum when

Table 4.6: Example 4.5 - 4×1 CPUs per SMP node - Non-overlapping execution

Time Step	node (0,0) CPU 0	CPU 1	CPU 2	CPU 3	group
0	$\begin{pmatrix}0\\0\\0\end{pmatrix}$				$\begin{pmatrix}0\\0\\0\end{pmatrix}$
1	$\begin{pmatrix}0\\0\\1\end{pmatrix}$	$\begin{pmatrix}1\\0\\0\end{pmatrix}$			$\begin{pmatrix}0\\0\\1\end{pmatrix}$
2	$\begin{pmatrix}0\\0\\2\end{pmatrix}$	$\begin{pmatrix}1\\0\\1\end{pmatrix}$	$\begin{pmatrix}2\\0\\0\end{pmatrix}$		$\begin{pmatrix}0\\0\\2\end{pmatrix}$
3	$\begin{pmatrix}0\\0\\3\end{pmatrix}$	$\begin{pmatrix}1\\0\\2\end{pmatrix}$	$\begin{pmatrix}2\\0\\1\end{pmatrix}$	$\begin{pmatrix}3\\0\\0\end{pmatrix}$	$\begin{pmatrix}0\\0\\3\end{pmatrix}$
4	$\begin{pmatrix}0\\0\\4\end{pmatrix}$	$\begin{pmatrix}1\\0\\3\end{pmatrix}$	$\begin{pmatrix}2\\0\\2\end{pmatrix}$	$\begin{pmatrix}3\\0\\1\end{pmatrix}$	$\begin{pmatrix}0\\0\\4\end{pmatrix}$
5	$\begin{pmatrix}0\\0\\5\end{pmatrix}$	$\begin{pmatrix}1\\0\\4\end{pmatrix}$	$\begin{pmatrix}2\\0\\3\end{pmatrix}$	$\begin{pmatrix}3\\0\\2\end{pmatrix}$	$\begin{pmatrix}0\\0\\5\end{pmatrix}$

Time Step	node (0,1) CPU 0	CPU 1	CPU 2	CPU 3	group
1	$\begin{pmatrix}0\\1\\0\end{pmatrix}$				$\begin{pmatrix}0\\1\\1\end{pmatrix}$
2	$\begin{pmatrix}0\\1\\1\end{pmatrix}$	$\begin{pmatrix}1\\1\\0\end{pmatrix}$			$\begin{pmatrix}0\\1\\2\end{pmatrix}$
3	$\begin{pmatrix}0\\1\\2\end{pmatrix}$	$\begin{pmatrix}1\\1\\1\end{pmatrix}$	$\begin{pmatrix}2\\1\\0\end{pmatrix}$		$\begin{pmatrix}0\\1\\3\end{pmatrix}$
4	$\begin{pmatrix}0\\1\\3\end{pmatrix}$	$\begin{pmatrix}1\\1\\2\end{pmatrix}$	$\begin{pmatrix}2\\1\\1\end{pmatrix}$	$\begin{pmatrix}3\\1\\0\end{pmatrix}$	$\begin{pmatrix}0\\1\\4\end{pmatrix}$

Time Step	node (1,0) CPU 0	CPU 1	CPU 2	CPU 3	group
4	$\begin{pmatrix}4\\0\\0\end{pmatrix}$				$\begin{pmatrix}1\\0\\4\end{pmatrix}$

the expression $\sum\limits_{k \neq i} \frac{w_k^S}{m_k}$ is minimized. According to Lemma C.3, this condition is valid when

$$m_k = w_k^S \left(\frac{m}{w_1^S \ldots w_{i-1}^S w_{i+1}^S \ldots w_n^S} \right)^{\frac{1}{n-1}}, k = 1, \ldots, n, k \neq i \tag{4.7}$$

Of course, it is not always feasible because the numbers m_i should be natural. But it always applies an approximate criterion for the selection of parameters m_k. Intuitively, it means that parameters m_k should be chosen so that ratios $\frac{w_k^S}{m_k}$ are as close to each other as possible.

Example 4.7: Let us consider a cluster of SMP nodes with $m = 4$ CPUs each and a 3-dimensional space J^S with size $20 \times 100 \times 20$. It means that $w_1^S = 20$, $w_2^S = 100$, $w_3^S = 20$.

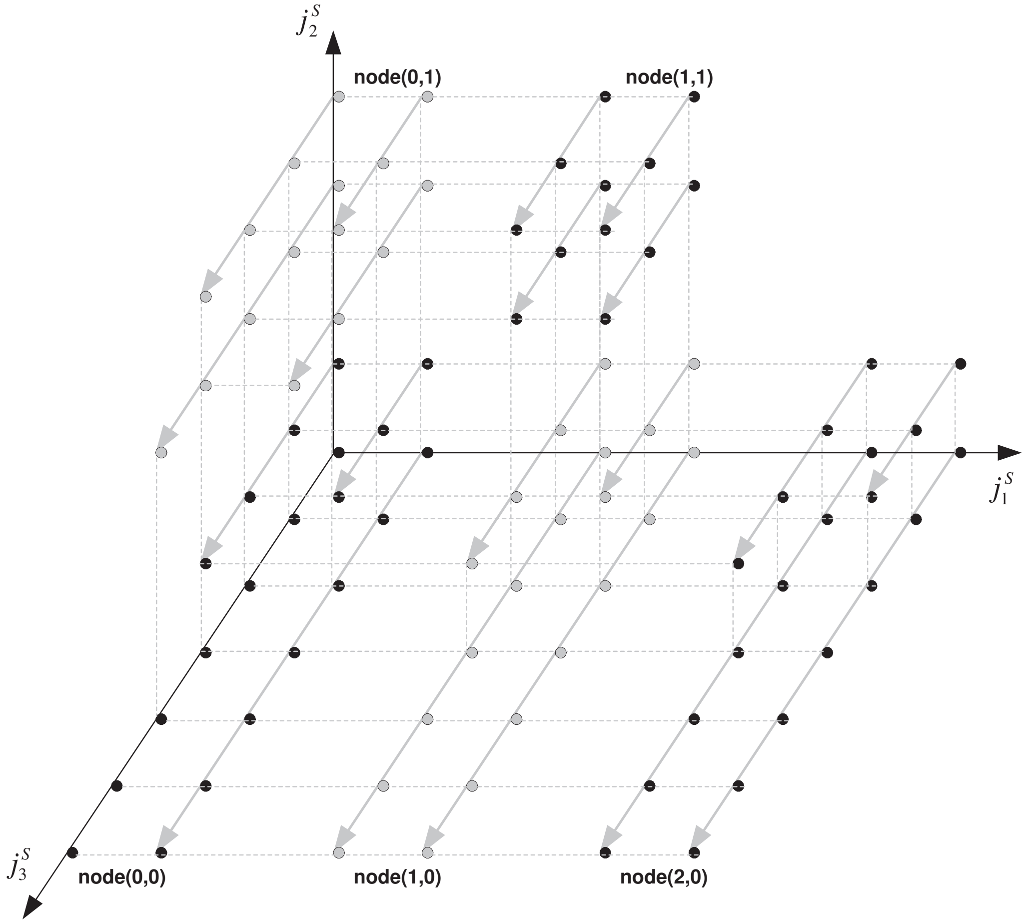


Figure 4.16: Example 4.6 - 2×2 CPUs per SMP node.
Neighboring tiles depicted using dots of the same color are assigned to the same SMP node.

Then, according to our previous analysis, the best choice will be: $i = 2$, $m_1 = 20\left(\frac{4}{20\times20}\right)^{\frac{1}{2}} = 2$, $m_3 = \frac{m}{m_1} = 2$. If we apply these values in expression (4.6), we get that the number of steps required for the completion of the execution will be $\wp_{overlap} = 156$. In contrast, if we chose $m_1 = 4$, $m_3 = 1$, then the expression (4.6) would get the value $\wp_{overlap} = 161 > 156$.

If the size of J^S is $20 \times 120 \times 150$ ($w_1^S = 20$, $w_2^S = 120$, $w_3^S = 150$), then, according to our previous analysis, the best choice will be: $i = 3$, $m_1 = 20\left(\frac{4}{20\times120}\right)^{\frac{1}{2}} = 0.816$. The closest natural number which divides $m = 4$ is $m_1 = 1$. Thus $m_2 = \frac{m}{m_1} = 4$. If we apply these values in the expression (4.6), we get that the number of steps required for the completion of the execution will be $\wp_{overlap} = 336$. In contrast, if we chose $m_1 = m_2 = 2$, then the expression (4.6) would get the value $\wp_{overlap} = 356 > 336$.

When the non-overlapping execution policy is followed, as deduced from formula (4.4), the

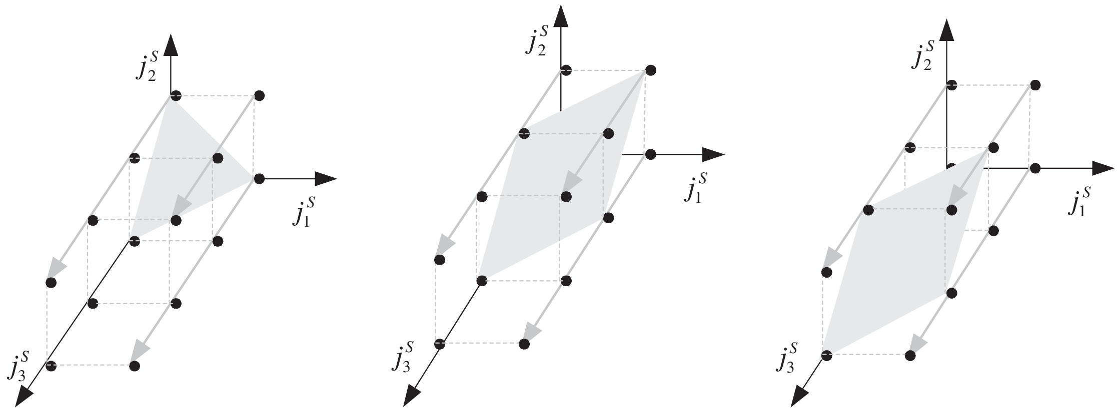


Figure 4.17: Example 4.6 - 2×2 CPUs per SMP node.
Each sub-figure depicts the tiles assigned to an SMP node. Tiles across a grey plane, are executed simultaneously by different CPUs of the SMP node.

selection of parameters m_k does no matter for the computation of the makespan.

Considering the minimization of the communication overhead

As one can easily observe in Example 4.7, when the overlapping execution policy is followed, the significance of the selection of parameters m_k, as it has just been described, is less when the maximum dimension w_i^S is much longer than dimensions $w_1^S, \ldots, w_{i-1}^S, w_{i+1}^S, \ldots, w_n^S$. So, it may be preferable to choose the values of parameters m_k taking into consideration the minimization of the communication requirements among the SMP nodes. This need is apparent when communication is not overlapped with computations. In that case, the less the communication load is, the faster the execution is completed.

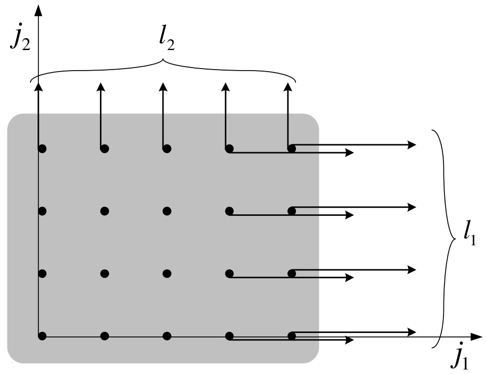


Figure 4.18: Communication load of a tile.
Communication load along dimension x is defined to be the number of dependence vectors, which cross the respective tile boundary line (or, generally, for n dimensions, hyperplane).

Table 4.7: Example 4.6 - 2×2 CPUs per SMP node - Overlapping execution
Unlike Examples 4.4 and 4.5, in this example we should label CPUs of an SMP node using both dimensions of the 2×2 virtual mesh.

Time Step	node (0,0) CPU (0,0)	CPU (0,1)	CPU (1,0)	CPU (1,1)	group
0	$\begin{pmatrix}0\\0\\0\end{pmatrix}$				$\begin{pmatrix}0\\0\\0\end{pmatrix}$
1	$\begin{pmatrix}0\\0\\1\end{pmatrix}$	$\begin{pmatrix}0\\1\\0\end{pmatrix}$	$\begin{pmatrix}1\\0\\0\end{pmatrix}$		$\begin{pmatrix}0\\0\\1\end{pmatrix}$
2	$\begin{pmatrix}0\\0\\2\end{pmatrix}$	$\begin{pmatrix}0\\1\\1\end{pmatrix}$	$\begin{pmatrix}1\\0\\1\end{pmatrix}$	$\begin{pmatrix}1\\1\\0\end{pmatrix}$	$\begin{pmatrix}0\\0\\2\end{pmatrix}$
3	$\begin{pmatrix}0\\0\\3\end{pmatrix}$	$\begin{pmatrix}0\\1\\2\end{pmatrix}$	$\begin{pmatrix}1\\0\\2\end{pmatrix}$	$\begin{pmatrix}1\\1\\1\end{pmatrix}$	$\begin{pmatrix}0\\0\\3\end{pmatrix}$
4	$\begin{pmatrix}0\\0\\4\end{pmatrix}$	$\begin{pmatrix}0\\1\\3\end{pmatrix}$	$\begin{pmatrix}1\\0\\3\end{pmatrix}$	$\begin{pmatrix}1\\1\\2\end{pmatrix}$	$\begin{pmatrix}0\\0\\4\end{pmatrix}$
5	$\begin{pmatrix}0\\0\\5\end{pmatrix}$	$\begin{pmatrix}0\\1\\4\end{pmatrix}$	$\begin{pmatrix}1\\0\\4\end{pmatrix}$	$\begin{pmatrix}1\\1\\3\end{pmatrix}$	$\begin{pmatrix}0\\0\\5\end{pmatrix}$

Time Step	node (0,1) CPU (0,0)	CPU (0,1)	CPU (1,0)	CPU (1,1)	group
3	$\begin{pmatrix}0\\2\\0\end{pmatrix}$				$\begin{pmatrix}0\\1\\2\end{pmatrix}$
4	$\begin{pmatrix}0\\2\\1\end{pmatrix}$	$\begin{pmatrix}0\\3\\0\end{pmatrix}$	$\begin{pmatrix}1\\2\\0\end{pmatrix}$		$\begin{pmatrix}0\\1\\3\end{pmatrix}$
5	$\begin{pmatrix}0\\2\\2\end{pmatrix}$	$\begin{pmatrix}0\\3\\1\end{pmatrix}$	$\begin{pmatrix}1\\2\\1\end{pmatrix}$	$\begin{pmatrix}1\\3\\0\end{pmatrix}$	$\begin{pmatrix}0\\1\\4\end{pmatrix}$

Time Step	node (1,0) CPU (0,0)	CPU (0,1)	CPU (1,0)	CPU (1,1)	group
3	$\begin{pmatrix}2\\0\\0\end{pmatrix}$				$\begin{pmatrix}1\\0\\2\end{pmatrix}$
4	$\begin{pmatrix}2\\0\\1\end{pmatrix}$	$\begin{pmatrix}2\\1\\0\end{pmatrix}$	$\begin{pmatrix}3\\0\\0\end{pmatrix}$		$\begin{pmatrix}1\\0\\3\end{pmatrix}$
5	$\begin{pmatrix}2\\0\\2\end{pmatrix}$	$\begin{pmatrix}2\\1\\1\end{pmatrix}$	$\begin{pmatrix}3\\0\\1\end{pmatrix}$	$\begin{pmatrix}3\\1\\0\end{pmatrix}$	$\begin{pmatrix}1\\0\\4\end{pmatrix}$

Let us represent with l_k the communication load of a tile along the k-th dimension, as indicated in Figure 4.18. If we group together $m_1 m_2$ tiles, then the communication loads among the SMP nodes will be $l_1 m_2 = \frac{m}{m_1} l_1$ and $l_2 m_1 = \frac{m}{m_2} l_2$, as indicated in Figure 4.19. Similarly, if we group together $m_1 \cdots m_{i-1} m_{i+1} \cdots m_n$ tiles, then the communication loads among the nodes of the cluster will be $\frac{m}{m_k} l_k$. Thus the total communication load of a group will be $l_{total} = m\left(\frac{l_1}{m_1} + \cdots + \frac{l_{i-1}}{m_{i-1}} + \frac{l_{i+1}}{m_{i+1}} + \cdots + \frac{l_n}{m_n}\right)$. According to Lemma C.3, it is minimized when $m_k = l_k \left(\frac{m}{l_1 \cdots l_{i-1} l_{i+1} \cdots l_n}\right)^{\frac{1}{n-1}}$, $k = 1, \ldots, n$, $k \neq i$. Of course, as numbers m_k should be natural, this criterion is also approximative.

Table 4.8: Example 4.6 - 2×2 CPUs per SMP node - Non-overlapping execution

Time Step	CPU (0,0)	CPU (0,1)	CPU (1,0)	CPU (1,1)	group
	node (0,0)				
0	$\begin{pmatrix}0\\0\\0\end{pmatrix}$				$\begin{pmatrix}0\\0\\0\end{pmatrix}$
1	$\begin{pmatrix}0\\0\\1\end{pmatrix}$	$\begin{pmatrix}0\\1\\0\end{pmatrix}$	$\begin{pmatrix}1\\0\\0\end{pmatrix}$		$\begin{pmatrix}0\\0\\1\end{pmatrix}$
2	$\begin{pmatrix}0\\0\\2\end{pmatrix}$	$\begin{pmatrix}0\\1\\1\end{pmatrix}$	$\begin{pmatrix}1\\0\\1\end{pmatrix}$	$\begin{pmatrix}1\\1\\0\end{pmatrix}$	$\begin{pmatrix}0\\0\\2\end{pmatrix}$
3	$\begin{pmatrix}0\\0\\3\end{pmatrix}$	$\begin{pmatrix}0\\1\\2\end{pmatrix}$	$\begin{pmatrix}1\\0\\2\end{pmatrix}$	$\begin{pmatrix}1\\1\\1\end{pmatrix}$	$\begin{pmatrix}0\\0\\3\end{pmatrix}$
4	$\begin{pmatrix}0\\0\\4\end{pmatrix}$	$\begin{pmatrix}0\\1\\3\end{pmatrix}$	$\begin{pmatrix}1\\0\\3\end{pmatrix}$	$\begin{pmatrix}1\\1\\2\end{pmatrix}$	$\begin{pmatrix}0\\0\\4\end{pmatrix}$
5	$\begin{pmatrix}0\\0\\5\end{pmatrix}$	$\begin{pmatrix}0\\1\\4\end{pmatrix}$	$\begin{pmatrix}1\\0\\4\end{pmatrix}$	$\begin{pmatrix}1\\1\\3\end{pmatrix}$	$\begin{pmatrix}0\\0\\5\end{pmatrix}$
	node (0,1)				
Time Step	**CPU (0,0)**	**CPU (0,1)**	**CPU (1,0)**	**CPU (1,1)**	**group**
2	$\begin{pmatrix}0\\2\\0\end{pmatrix}$				$\begin{pmatrix}0\\1\\2\end{pmatrix}$
3	$\begin{pmatrix}0\\2\\1\end{pmatrix}$	$\begin{pmatrix}0\\3\\0\end{pmatrix}$	$\begin{pmatrix}1\\2\\0\end{pmatrix}$		$\begin{pmatrix}0\\1\\3\end{pmatrix}$
4	$\begin{pmatrix}0\\2\\2\end{pmatrix}$	$\begin{pmatrix}0\\3\\1\end{pmatrix}$	$\begin{pmatrix}1\\2\\1\end{pmatrix}$	$\begin{pmatrix}1\\3\\0\end{pmatrix}$	$\begin{pmatrix}0\\1\\4\end{pmatrix}$
	node (1,0)				
Time Step	**CPU (0,0)**	**CPU (0,1)**	**CPU (1,0)**	**CPU (1,1)**	**group**
2	$\begin{pmatrix}2\\0\\0\end{pmatrix}$				$\begin{pmatrix}1\\0\\2\end{pmatrix}$
3	$\begin{pmatrix}2\\0\\1\end{pmatrix}$	$\begin{pmatrix}2\\1\\0\end{pmatrix}$	$\begin{pmatrix}3\\0\\0\end{pmatrix}$		$\begin{pmatrix}1\\0\\3\end{pmatrix}$
4	$\begin{pmatrix}2\\0\\2\end{pmatrix}$	$\begin{pmatrix}2\\1\\1\end{pmatrix}$	$\begin{pmatrix}3\\0\\1\end{pmatrix}$	$\begin{pmatrix}3\\1\\0\end{pmatrix}$	$\begin{pmatrix}1\\0\\4\end{pmatrix}$

In the rest of this chapter, we shall theoretically and experimentally compare the proposed methods with each other. Although our above theoretical results can be applied to any convex tile space, as explained in §2.2, we shall go on using only rectangular tile spaces, as in our previous examples. We consider that this simplification is convenient for clearly expressing some ideas and it does not constrain any of the advantages or disadvantages of the proposed methods.

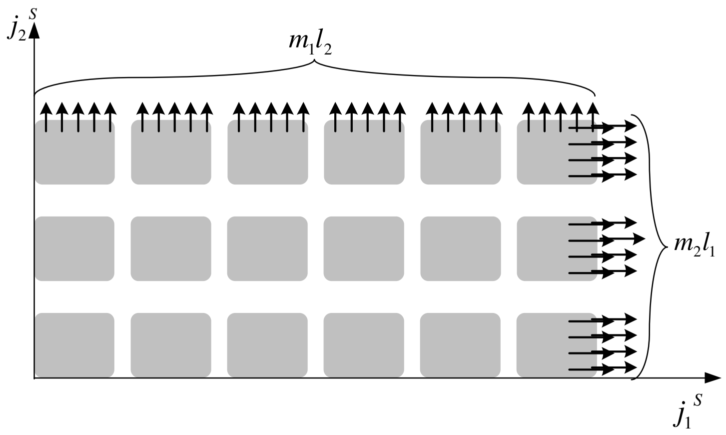


Figure 4.19: Communication load of a group.

Communication load along dimension x is defined to be the product of the communication load of a tile along dimension x, and the number of tiles, which touch the respective group boundary line (or, generally, for n dimensions, hyperplane).

4.5 Theoretical Comparison

In this section we shall compare vertical grouping, which is indicated in Figure 4.3, with the proposed scheme of hyperplane grouping, which is shown in Figures 4.4 and 4.8, in the case of a 2-dimensional algorithm and a cluster of SMPs with 2 CPUs each.

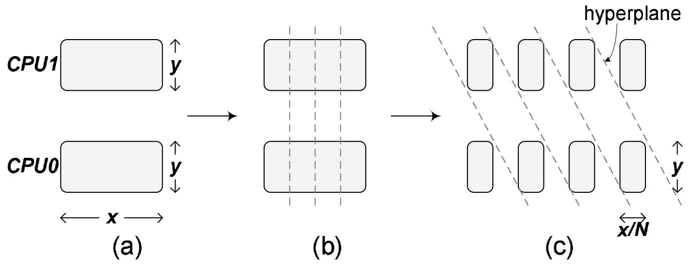


Figure 4.20: In order to execute at the same time tiles grouped together by a vertical grouping scheme, we should further divide them into sub-tiles and execute some of them in parallel, according to an intra-tile hyperplane scheduling.

As we have already mentioned, vertical grouping cannot exploit the computational power of both CPUs of our SMPs unless we split each tile into smaller subtiles and compute some of them in parallel, as shown in Figure 4.20. Let us assume that a CPU needs time α for the computation of a tile with dimensions x, y (Figure 4.20a). Consequently, it will need time $\frac{\alpha}{N}$ for the computation of a respective subtile with dimensions $\frac{x}{N}$, y (Figure 4.20c). The subtiles which are created can be computed by 2 CPUs in $N + 1$ computational steps, interleaved with N synchronization steps, following an optimal linear time schedule $(1, 1)$ as in Figure 4.20c. If the average time consumed for the synchronization of 2 CPUs of an SMP node is t_{synch_in}, then the total time required for the computation of a pair of initial tiles is

$$\beta = \alpha \frac{N+1}{N} + N t_{synch\_in}. \tag{4.8}$$

β is minimized when

$$N = \sqrt{\frac{\alpha}{t_{synch\_in}}}. \tag{4.9}$$

Therefore, the minimum value of β is $\beta_{min} = \alpha + 2\sqrt{\alpha t_{synch\_in}} > \alpha$.

If we consider an iteration space of size $X \times Y$, tiled with rectangular tiles of size xy, (for example in Figures 4.3, 4.4 we have $\frac{X}{x} = 10$, $\frac{Y}{y} = 6$), then we have the following options:

1. Following the **non-overlapping** scheme (which can be implemented using blocking calls) in combination with **vertical grouping**, the number of time steps required for the completion of the execution is $\wp = \frac{X}{x} + \frac{Y}{2y} - 1$. The minimum duration of a time step (according to formula (2.10)) is $\beta_{min} + t_{comm}$, where t_{comm} is the time required for the communication between two SMP nodes. Thus, the total time required is

$$T_{blocking,vertical} = \wp(\beta_{min} + t_{comm}) \simeq (\frac{X}{x} + \frac{Y}{2y})(\beta_{min} + t_{comm})$$

2. Following the **overlapping** scheme (which can be implemented using non-blocking calls) in combination with **vertical grouping**, the number of time steps required for the completion of the execution is $\wp = \frac{X}{x} + \frac{Y}{y} - 2$. According to formula (2.11), if we set $t_{comp} = \beta_{min}$, the minimum duration of a time step is $t_{start\_dma} + max(\beta_{min}, t_{comm\_dma}) + t_{synchro}$. Thus, the total time required is

$$T_{non-blocking,vertical} = \wp(t_{start\_dma} + max(\beta_{min}, t_{comm\_dma}) + t_{synchro}) \simeq$$
$$\simeq (\tfrac{X}{x} + \tfrac{Y}{y})(t_{start\_dma} + max(\beta_{min}, t_{comm\_dma}) + t_{synchro})$$

If $\beta_{min} \geq t_{comm\_dma}$, then

$$T_{non-blocking,vertical} \simeq (\frac{X}{x} + \frac{Y}{y})(t_{start\_dma} + \beta_{min} + t_{synchro})$$

3. Following the **overlapping** scheme in combination with **hyperplane grouping**, the number of time steps required for the completion of the execution is $\wp = \frac{X}{x} + \frac{3Y}{2y} - 2$. According to formula (2.11), if we set $t_{comp} = \alpha$, the minimum duration of a time step is

$t_{start\_dma} + max(\alpha, t_{comm\_dma}) + t_{synchro}$. Thus, the total time required is

$$T_{non-blocking,hyperplane} = \wp(t_{start\_dma} + max(\alpha, t_{comm\_dma}) + t_{synchro}) \simeq \\ \simeq (\tfrac{X}{x} + \tfrac{3Y}{2y})(t_{start\_dma} + max(\alpha, t_{comm\_dma}) + t_{synchro})$$

If $\alpha \geq t_{comm\_dma}$, then

$$T_{non-blocking,hyperplane} \simeq (\frac{X}{x} + \frac{3Y}{2y})(t_{start\_dma} + \alpha + t_{synchro})$$

In most real problems it holds that $\frac{Y/y}{X/x} = \lambda \ll 1$. Therefore, in case that $\beta_{min} \geq t_{comm}$, the overlapping scheme in combination with vertical grouping is more efficient than the non-overlapping scheme, when $t_{comm\_dma} > (t_{start\_dma} + \beta_{min} + t_{synchro})\frac{\frac{Y}{2y}}{\frac{X}{x}+\frac{Y}{2y}} \Leftrightarrow t_{comm} > \frac{\lambda}{2}(t_{start\_dma} + \beta_{min} + t_{synchro})$. In addition, the overlapping scheme, in combination with hyperplane grouping, is more efficient than the overlapping scheme, in combination with vertical grouping, when $(\frac{X}{x}+\frac{3Y}{2y})(t_{start\_dma}+\alpha+t_{synchro}) < (\frac{X}{x}+\frac{Y}{y})(t_{start\_dma}+\alpha+2\sqrt{\alpha t_{synch\_in}}+t_{synchro})$. If we consider $t_{start\_dma} + t_{synchro} \ll \alpha$, then, we get $2\sqrt{\frac{t_{synch\_in}}{\alpha}} > \frac{\lambda/2}{1+\lambda} \simeq \frac{\lambda}{2} \Rightarrow t_{synch\_in} > \alpha\left(\frac{\lambda}{4}\right)^2$. This is due to the fact that, using vertical grouping, the pipeline filling is faster, while, using hyperplane grouping, the pipeline throughput is faster. So, hyperplane grouping is preferable when the mapping dimension of the tile space is long enough in comparison to the rest dimensions. However, in any case, the hyperplane grouping has the advantage that it needs no extra tiling inside each tile in order to exploit the computational force of the CPUs.

Consequently, which communication and grouping policy is optimal, depends on the hardware characteristics. One should estimate the time parameters involved in the model (computation, transfer initialization overhead, actual transfer overhead) and determine which scheme is going to give the peak performance. In general, the purpose of the overlapping scheme, in combination with hyperplane grouping, is exploiting all modern architectural characteristics of NICs, such as DMA, RDMA, Zero Copy, or even NICs with embedded processors. Thus, this scheme will be optimal when these characteristics are actually available.

4.6 Experimental Verification

4.6.1 Experimental platform and algorithm

In [STK02], the pipelined schedule proposed in [GSK01] was applied, using a cluster of single CPU nodes with PCI-SCI NICs. In this thesis, as in [AST$^+$05], [ASTK02a], [ASTK02b], in order to evaluate the proposed methods, we ran our experiments on a Linux SMP cluster with 8 identical nodes. Each node had 128MB of RAM and 2 Pentium III 800 MHz CPUs. The cluster nodes were interconnected with an SCI ring, using SCI Dolphin's PCI-SCI D330 cards. SCI

NICs support shared memory programming, either through PIO (Programmed-IO) messaging, or through DMA. We are using their kernel-level DMA support for messaging. Invoking kernel system calls, causes extra CPU cycles overhead. However, we can avoid extra copying from user space to kernel space (physical memory) when using DMA. We allocate user level pages, which correspond to physically contiguous pre-reserved memory regions, for DMA communications.

Our test application was the following code:

```
for(i=1; i<=X; i++)
     for(j=1; j<=Y; j++)
       for(k=1; k<=Z; k++)
         A[i][j][k]=func(A[i-1][j][k],A[i][j-1][k],A[i][j][k-1]);
```

where A is an array of $X \times Y \times Z$ floats and $X = Y << Z$. Without lack of generality, we select as a tile a rectangle with ij, ik and jk sides. The dimension k is the largest one, so all tiles along k-axis are mapped onto the same processor, as proposed in §4.4.4. Each tile has i, j dimensions equal to x and the tile's "height" along k-axis equal to z. There are $\frac{X}{x}$ tiles along dimensions i and j and $\frac{Z}{z}$ tiles along dimension k. Tile's volume is equal to $g = x^2z$, and since the number of available processors is initially known, the only unknown parameter is z.

We applied both vertical and hyperplane grouping, using both blocking and non-blocking communication primitives. Since both vertical and hyperplane grouping can be combined with both overlapping and non-overlapping communication, we experimented with all four combinations. For each exemplary iteration space and each possible tile height, we calculated the total execution time for the above schemes. In order to implement these schemes, we used Linux POSIX threads with semaphores for the synchronization among the processors of an SMP node and the SISCI driver and libraries for the communication among the SMP nodes.

4.6.2 Tuning Parameters

First of all, as far as the implementation of vertical grouping is concerned, we experimentally verified formula (4.9), in order to calculate the optimal execution time for a couple of tiles by an SMP node. We assigned the computation of two tiles to the two processors of an SMP node and measured their execution time in respect to the number of subtiles into which each tile was cut, in order not to violate the iteration dependences. The experimental results, along with the theoretically expected curve, are plotted in Figure 4.21. The theoretical plot was calculated using the formula (4.8) with $\alpha \simeq 69msec$ and $t_{synch\_in} \simeq 11\mu sec$. These values were experimentally measured by running a simple code fragment thousands of times and calculating the average execution time. If we find the $N_{best,theoretical}$, that is the point N where the theoretical minimum is achieved and for this N we find the corresponding experimental overall time, then the difference between this value and the experimental minimum is less than $0,15\%$. This is clearly shown in Figure 4.22, which has zoomed in the minimum of the diagram of Figure 4.21. So we can safely use $N_{best,theoretical}$ as N_{best}.

This can be simply justified as follows: If we consider a shift δN of N, then the shift of β will be $\delta\beta = -\alpha\frac{\delta N}{N(N+\delta N)} + t_{synch\_in}\delta N$. If, in this formula, we set $N = N_{best,theoretical}$ we get that: $\frac{\delta\beta}{\beta_{min}} = \frac{(\frac{\delta N}{N_{best,theoretical}})^2}{1+\frac{\delta N}{N_{best,theoretical}}}\frac{1}{2+\sqrt{\frac{\alpha}{t_{synch\_in}}}}$. Therefore, the less the parameter t_{synch_in} is in comparison to α, the less important the exact selection of N is. Intuitively, in the extreme case, where t_{synch_in} is 0, we could always achieve the same results, no matter how fine grained the parallelism is (i.e. for very large N's). However, t_{synch_in} is always considerable and cannot be ignored for real life SMP architectures.

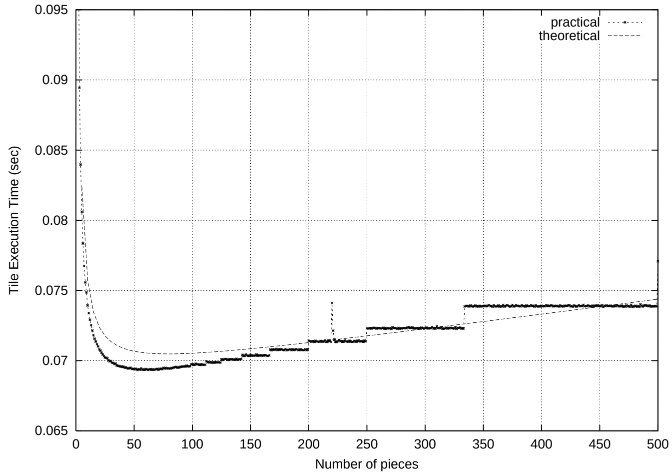


Figure 4.21: Vertical grouping - Tile execution time in respect to the number of slices a tile is cut

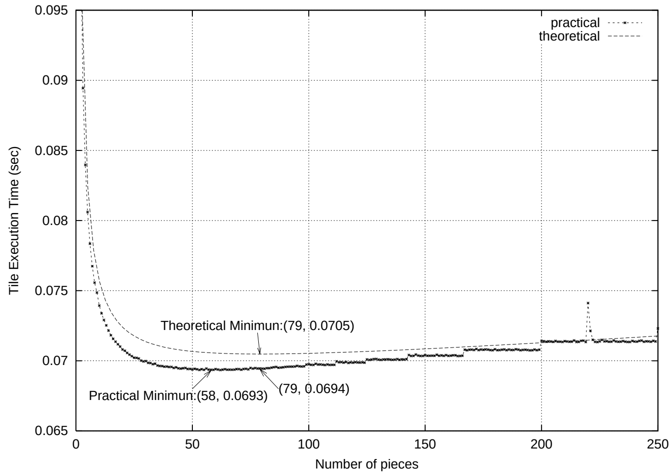


Figure 4.22: Vertical grouping - Zoom in the minimum area of the plot of Figure 4.21

4.6.3 Experimental Results

Once vertical grouping had been implemented and approximated with a theoretical formula, we implemented both blocking and non-blocking communication schemes. As far as the blocking

Table 4.9: Implementation of the non-overlapping scheme

Thread 0:	*Thread 1:*

foreach group assigned to node(i,j) do{
  receive from node(i-1,j)
  receive from node(i,j-1)
	foreach group assigned to node(i,j) do{
	
	  receive from node(i,j-1)

compute_tile(i,j,k,CPU0)	compute_tile(i,j,k,CPU1)
 send to node(i,j+1)	send to node(i+1,j) send to node(i,j+1)
semaphore_post(sem_s1) semaphore_wait(sem_s2) }	semaphore_post(sem_s2) semaphore_wait(sem_s1) }

Table 4.10: Implementation of the overlapping scheme

Thread 0:	*Thread 1:*	*Explanation*

foreach group assigned to node(i,j) do{
  trigger_interrupt to node(i-1,j)
  trigger_interrupt to node(i,j-1)
 	
  wait_interrupt from node(i,j+1)
 	
  send_dma(node(i,j+1),data)
	foreach group assigned to node(i,j) do{
	
	  trigger_interrupt to node(i,j-1)
	  wait_interrupt from node(i+1,j)
	  wait_interrupt from node(i,j+1)
	  send_dma(node(i+1,j),data)
	  send_dma(node(i,j+1),data)
		
		Inform "previous" nodes:
		"I am ready to accept data"
		Wait until "next" nodes
		are ready to accept data
		Initialization of DMA transfer
		to neighboring nodes

compute_tile(i,j,k,CPU0)	compute_tile(i,j,k,CPU1)	
 wait_dma() trigger_interrupt to node(i,j+1) wait_interrupt from node(i-1,j) wait_interrupt from node(i,j-1)	wait_dma() wait_dma() trigger_interrupt to node(i+1,j) trigger_interrupt to node(i,j+1) wait_interrupt from node(i,j-1)	Wait for DMA to complete Inform "next" nodes: "Your data has arrived" Wait until "previous" nodes have finished sending data
semaphore_post(sem_s1) semaphore_wait(sem_s2) }	semaphore_post(sem_s2) semaphore_wait(sem_s1) }	Implementation of a barrier

Table 4.11: Implementation of the vertical vs. hyperplane grouping

Vertical grouping	
compute_tile(i,j,k,CPU0):	*compute_tile(i,j,k,CPU1):*

foreach subtile of this tile do{
  compute each iteration of this subtile
  semaphore_post(sem1)
  semaphore_wait(sem2)
 	
}
	foreach subtile of this tile do{
	
	  semaphore_post(sem2)
	  semaphore_wait(sem1)
	  compute each iteration of this subtile
	}

Hyperplane grouping	
compute_tile(i,j,k,CPU0):	*compute_tile(i,j,k,CPU1):*
compute each iteration of this tile	compute each iteration of this tile

communication scheme is concerned, it was implemented using the pseudo-code of Table 4.9. On the other hand, the non-blocking scheme was implemented using the pseudo-code of Table 4.10. Notice that during each time step every SMP node in the ij plane with coordinates (i, j) receives from neighboring nodes $(i-1, j)$ and $(i, j-1)$, computes and sends to nodes $(i+1, j)$,$(i, j+1)$ (Figure 4.23). Since the `send_dma()` call is not blocking, the computation of the tiles will be performed concurrently with the transferring of data among the SMP nodes. After the execution of `wait_dma()`, it is assured that both computation and communication are already completed.

The implementation of vertical and hyperplane grouping was achieved by a proper procedure `compute_tile(i, j, k, CPUx)`. In order to implement vertical grouping, we used the pseudocode of Table 4.11. The number of subtiles inside a tile was selected according to formula (4.9). Notice that, the implementation of hyperplane grouping was much simpler, as shown in

Table 4.11.

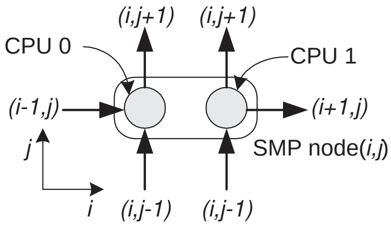


Figure 4.23: Directions and source/destination nodes of message exchanges for an SMP node with 2 CPUs

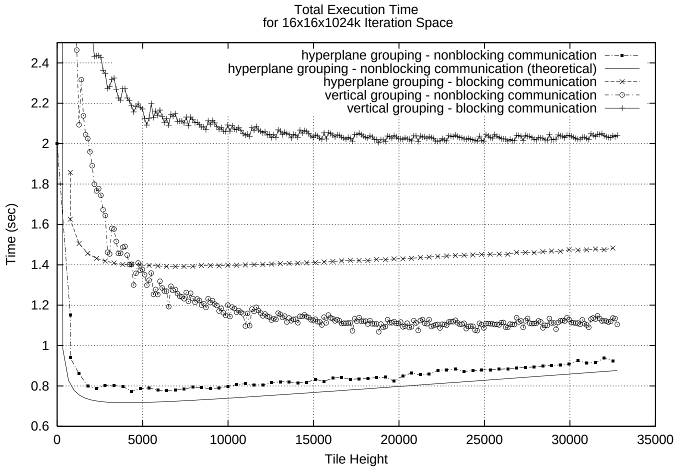


Figure 4.24: Experimental Results: $16 \times 16 \times 1024k$ iteration space

The problem was solved using various values of $X = Y$ and Z. For each schedule, we are interested in the overall minimum execution time achieved at an optimally selected tile height (see [GSK01], [STK02], [HS98]). The experimental results, shown in Figures 4.24-4.28, illustrate that, in every case, non-blocking communication is preferable to blocking communication and hyperplane grouping is preferable to vertical grouping. The lowest minimum is clearly achieved when using hyperplane grouping, in combination with non-blocking communication, in all cases. As far as hyperplane grouping, in combination with non-blocking communication, is concerned, according to our scheduling theory (formula (4.6)), the number of time steps required for the completion of an experiment is $\wp(x,y,z) = \frac{3X}{2x} + \frac{2Y}{y} + \frac{Z}{z} - 4$. The minimum duration of a time step, as mentioned in §4.5, is $(t_{start\_dma} + t_{comp} + t_{synchro})$. Thus, $T_{non-blocking,hyperplane} = (\frac{3X}{2x} + \frac{2Y}{y} + \frac{Z}{z} - 4)(t_{start\_dma} + t_{comp} + t_{synchro})$. This formula was used to produce the theoretical curves of Figures 4.24-4.26 with values $t_{start\_dma} + t_{synchro} = 100\mu sec$ and $t_{comp} = x^2 z t_{comp1}$, where t_{comp1} is the execution time of a single iteration and it was measured equal to $39,6nsec$.

One can easily verify from Figures 4.24-4.28 that the graphs of the theoretical model are very close to the corresponding experimental graphs, not only at the desired minimum, but along the whole graph. Thus, the theoretical model of scheduling is strongly verified by the experimental

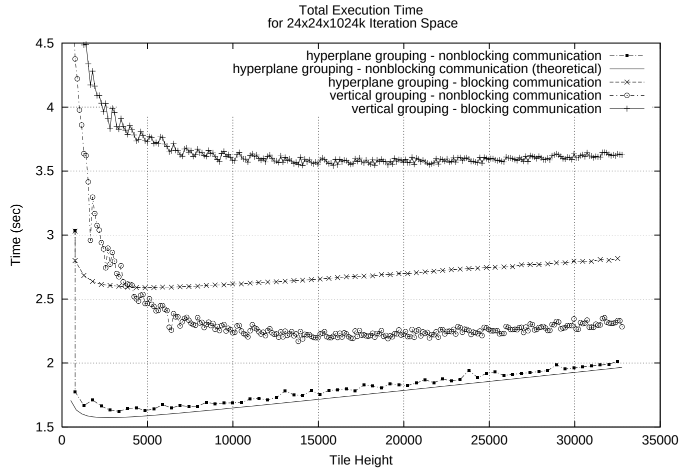


Figure 4.25: Experimental Results: $24 \times 24 \times 1024k$ iteration space

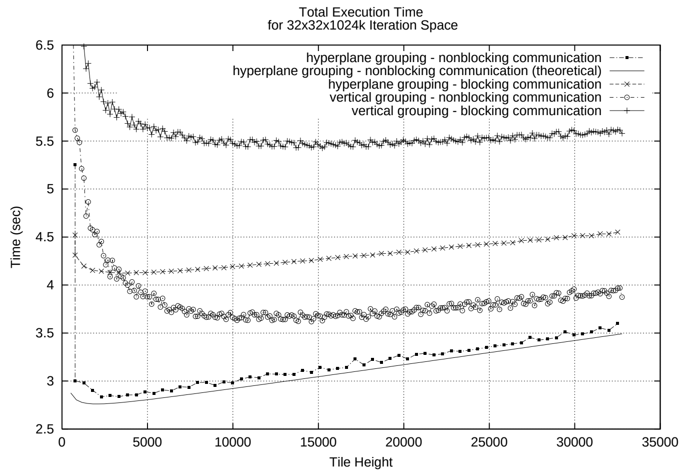


Figure 4.26: Experimental Results: $32 \times 32 \times 1024k$ iteration space

results.

4.6.4 Scalability Issues

The theoretical model presented in this chapter is general enough, so as not to be differentiated when scaling up the underlying hardware architecture. However, in this section, we shall examine some practical problems, which may rise.

For example, if we add more SMP nodes, the initial iteration space may be cut into smaller tiles. Thus, the computation to communication ratio of each tile $\frac{t_{comp}}{t_{comm\_dma}}$ may reduce because of two reasons:

1. Less computations are assigned to each SMP node, while the amount of data transfer required is not proportionally reduced.

2. If the network is saturated (by more SMP nodes trying to send more data in more messages

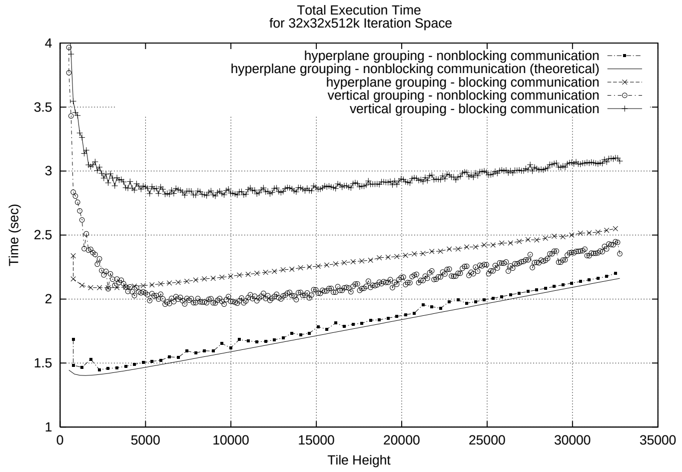


Figure 4.27: Experimental Results: $32 \times 32 \times 512$ iteration space

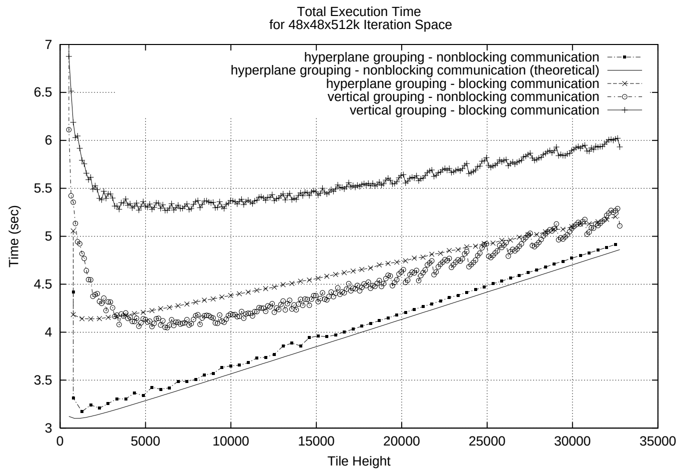


Figure 4.28: Experimental Results: $48 \times 48 \times 512$ iteration space

to each other), the increase in t_{comm_dma} will be more than relative to the increase in the volume of data transmitted.

However, considering an application with uniform dependences, as described in the algorithmic model in §2.2, and a torus interconnection topology, such as the one used for our experiments, the network will be never saturated due to the increase of SMP nodes. This is because each node need to communicate only with its neighbors, thus there are no shared resources among different communication channels. Thus, only the first reason mentioned above can potentially cause some trouble when adding more SMP nodes. But, if it still holds $t_{comp} \geq t_{comm\_dma}$, nothing will change in the implementation of our model. In the opposite case ($t_{comp} < t_{comm\_dma}$), the use of even more nodes will not be efficient. This problem will not concern our scheduling, but it will mean that the communication architecture is too slow to exploit all the computation power of the computing system. Then, it would be better not to use all the nodes available, as implied in [HS98]. However, regarding the speed and efficiency of modern interconnection networks, like the SCI based interconnect, or the Myrinet interconnect used for the experimentation of this thesis, it is not possible to encounter such a situation, especially when computing large iteration spaces of real problems.

If we add more CPUs inside each SMP node, we may again cut the initial iteration space into smaller tiles. The computation to communication ratio of each tile $\frac{t_{comp}}{t_{comm\_dma}}$ will be decreased again, but only for one reason: Less computations are assigned to each CPU. In particular, $\frac{t_{comp}}{t_{comm\_dma}}$ will be conversely proportional to the number of CPUs inside each SMP node. In this case, no more data need to be sent through the interconnection network, since the additional CPUs communicate with each other and with the preexisting CPUs through the shared memory of the SMP node. However, $t_{synchro}$ and t_{start_dma} will slightly increase, because, first, more CPUs need to initialize their DMA sends and receives and, second, these operations can not be executed at the same time by different threads of the same node (no thread-safe environment – see the implementation code of Table 4.10). This problem can be solved by assigning all communication overhead to one thread only and at the same time reducing the computation overhead of this thread. Following that technique, CPUs do not remain idle waiting to synchronize with each other, since the amount of computations assigned to the communicating thread may be properly calculated, so as the total communication+computation overhead to be evenly distributed among the CPUs of each node. The exact solution of this problem concerns the research conducted by Nikolaos Drosinos in Computing Systems Laboratory.

Another aspect of scalability (concerning the scheduling algorithm, not the hardware) is having so large iteration spaces that we cannot cut them into so few tiles. That is, applying a tile selection technique, such as the ones presented in [BDRR94], [Xue97a], [Xue00], [RR04], [KRC99], [LRW91], [WL91a], [PHP03], [MHCF98], we may get more rows of tiles than the CPUs available. Then we should apply a more complicated technique for assigning tiles to SMP nodes and CPUs as described in [AKK04] and in Chapter 5 of this thesis.

5

Scheduling onto a fixed number of homogeneous SMP nodes

In this chapter, we assume that the number of SMP nodes of the available cluster may be less than the number of SMP nodes needed for the application of a time scheduling produced by the techniques proposed in Chapter 4. Thus, we need to allocate more than one of the tasks produced to each CPU. Which of them will be assigned to the same CPU? This chapter answers the above question by proposing five alternative schedules. Each one seems to be preferable for a specific form of tile spaces or for a set of architectural characteristics.

5.1 Introduction

The schedule proposed in Chapter 4 assumes the availability of an unlimited number of SMP nodes or that the tile size has been selected so as the SMP nodes required do not exceed the available SMP nodes. However, it cannot be always true, since the tile size is often selected so as to minimize communication load [BDRR94], [Xue97a], [Xue00], [RR04], or to achieve locality in memory data references [KRC99], [LRW91], [WL91a], [PHP03], [MHCF98]. In [AKPT00] Andronikos et al. have proposed an assignment scheme onto a fixed number of nodes. It might be generalized, for assigning tiles onto a fixed number of nodes, however the complexity of evaluating which tiles should be assigned to which node is too high. Such an allocation scheme may be optimal, but it will be impractical if we want to incorporate it into an automatic code generation tool [GDAK02a]. On the other hand, automatic code generation without taking care of processor allocation and scheduling has certain drawbacks:

1. A lot of precesses are generated, which are not actually needed, since they may outnumber the processors available. As a result, the processes generation time may unnecessarily be comparable to the processes execution time, as we found out during our experimentation in [GDAK02a].

2. In addition, we are obliged to have confidence in the operating system to schedule processes. For example, MPI automatically allocates processes to processors cyclically, which may be far from optimal.

3. Finally, in case more than one processes are allocated to a CPU, optimizing tile size and shape according to cache locality criteria [KRC99], [LRW91], [WL91a], [PHP03], [MHCF98], will not have the desired results, as context-switching frequently between them might not allow them to build sufficient context in the cache.

For this purpose, a regular, periodic allocation scheme is needed, even if it is suboptimal. In [BDRV99], [CDR97] Boulet et al. and Calland et al. have theoretically proven the optimality of a cyclic assignment of 2-dimensional tiles onto a fixed number of single CPU nodes. On the other hand, Manjikian and Abdelrahman have presented in [MA01] an alternative method for scheduling tiled iteration spaces onto a fixed number of SMP nodes, without taking into account that there is no need for communication among CPUs of the same SMP node, since the data required are located in the node's shared memory.

In this chapter, we propose some methods for scheduling tiled iteration spaces onto an existing cluster with a fixed number of SMP nodes. All following formulas, which refer to the allocation of tiles or groups to the nodes of the cluster or to the corresponding execution steps are valid for any convex tile space, as defined in §2.2. However, when calculating the number of time steps required for the completion of the execution (makespan), we consider a rectangular tile space, as in formulas (4.3), (4.4), (4.6). We use this simplification in order to point out the basic concepts concerning each one of the proposed methods, without too complicated mathematical formulas. Anyway, it does not constrain any of the advantages or disadvantages of the methods proposed, apart from those concerning load balancing. In order to further simplify the mathematical formulas, we assume that the longest dimension of the tile space is the first one. Thus, according to §3.3.2, §4.4.4, tiles along the first dimension will be assigned to the same processor. This assumption can be easily cancelled by simply interchanging the first dimension with anyone else.

5.2 Cyclic assignment to SMPs

In [BDRV99], [CDR97] the optimality of the cyclic assignment of 2-dimensional tiles onto a fixed number of processors was theoretically proven. However, the calculations in [BDRV99], [CDR97] did not take into account the communication overhead involved. Generalizing this approach for n-dimensional tiles and for clusters of SMP nodes, we consider that the available SMP nodes form a virtual $(n-1)$-dimensional mesh of $p_2 \times \cdots \times p_n = p$ SMP nodes. We cyclically assign the groups to the SMP nodes. That is, we assign group $\vec{j^G}$ to the SMP node $(j_2^G \% p_2, \ldots, j_n^G \% p_n)$, as indicated in Figure 5.1.

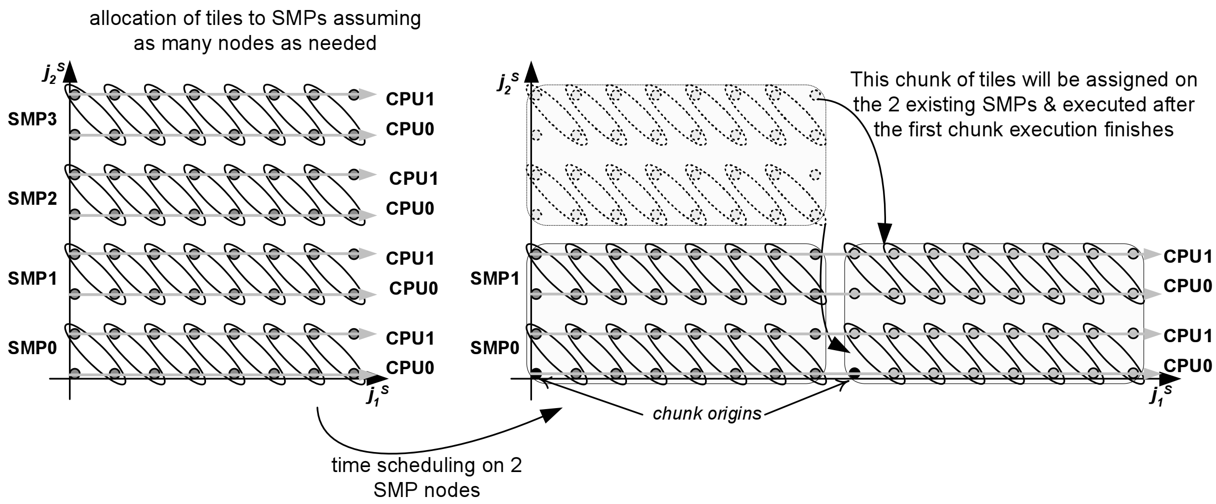


Figure 5.1: Cyclic assignment to SMP nodes.

Groups are cyclically assigned to SMP nodes. Equivalently, tiles are cyclically assigned to CPUs. Tile space areas, which can fit the existing architecture, are named as “chunks”. Chunks of tiles are executed one after the other, in lexicographic order.

Theorem 5.1 *The makespan of cyclically assigning a rectangular tile space to SMP nodes,* *assuming overlapping communication with computation is:*

$$\begin{aligned}\wp_{cyclic-overlap} &= \sum_{i=2}^{n}\left[(w_i^S-1)\%m_ip_i+(\lceil\frac{w_i^S}{m_i}\rceil-1)\%p_i\right]+w_1^S\prod_{i=2}^{n}\lceil\frac{w_i^S}{m_ip_i}\rceil\leq\\ &\leq\sum_{i=2}^{n}[(m_i+1)p_i]-2n+2+w_1^S\prod_{i=2}^{n}\lceil\frac{w_i^S}{m_ip_i}\rceil\end{aligned} \tag{5.1}$$

Proof: *Each SMP node will execute* $\lceil\frac{w_2^S}{m_2p_2}\rceil\times\cdots\times\lceil\frac{w_n^S}{m_np_n}\rceil$ *rows of groups. If the rows of groups assigned to an SMP node, are executed in lexicographic order, row* $(\bullet, j_2^G, \ldots, j_n^G)$ *will be executed in SMP node* $(j_2^G\%p_2,\ldots,j_n^G\%p_n)$ *after* $\sum_{i=2}^{n}\left[\lfloor\frac{j_i^G}{p_i}\rfloor\prod_{k=i+1}^{n}\lceil\frac{w_k^S}{m_kp_k}\rceil\right]$ *rows, imposing a latency of* w_1^S *time steps each. Thus, there is a total latency of* $w_1^S\sum_{i=2}^{n}\left[\lfloor\frac{j_i^G}{p_i}\rfloor\prod_{k=i+1}^{n}\lceil\frac{w_k^S}{m_kp_k}\rceil\right]$ *time steps. In addition, as deduced from Figure 5.1, the location of a group, relatively to the corresponding chunk origin, is* $(j_1^{G\prime}, j_2^G\%p_2,\ldots,j_n^G\%p_n)$*, where* $j_1^{G\prime}=j_1^S+\sum_{i=2}^{n}j_i^S\%m_ip_i$.

Therefore, if the underlying architecture allows for concurrent execution of computations and communication, following the overlapping execution scheme, group $\vec{j^G}$ *will be computed during the time step*

$$t(\vec{j^G})=j_1^{G\prime}+\sum_{i=2}^{n}j_i^G\%p_i+w_1^S\sum_{i=2}^{n}\left[\lfloor\frac{j_i^G}{p_i}\rfloor\prod_{k=i+1}^{n}\lceil\frac{w_k^S}{m_kp_k}\rceil\right]. \tag{5.2}$$

Thus, the number of time steps required for the completion of the execution will be

$$\begin{aligned}&\wp_{cyclic-overlap}=\max t(\vec{j^G})-\min t(\vec{j^G})+1=\\ &\overset{(C.3)}{=}u_1^S+\sum_{i=2}^{n}\left[u_i^S\%m_ip_i+\lfloor\frac{u_i^S}{m_i}\rfloor\%p_i\right]+w_1^S\sum_{i=2}^{n}\left[\lfloor\frac{u_i^S}{m_ip_i}\rfloor\prod_{k=i+1}^{n}\lceil\frac{w_k^S}{m_kp_k}\rceil\right]+1=\\ &\overset{(C.4)}{=}\sum_{i=2}^{n}\left[(w_i^S-1)\%m_ip_i+(\lceil\frac{w_i^S}{m_i}\rceil-1)\%p_i\right]+w_1^S+w_1^S\sum_{i=2}^{n}\left[(\lceil\frac{w_i^S}{m_ip_i}\rceil-1)\prod_{k=i+1}^{n}\lceil\frac{w_k^S}{m_kp_k}\rceil\right]=\\ &\overset{(C.7)}{=}\sum_{i=2}^{n}\left[(w_i^S-1)\%m_ip_i+(\lceil\frac{w_i^S}{m_i}\rceil-1)\%p_i\right]+w_1^S\prod_{i=2}^{n}\lceil\frac{w_i^S}{m_ip_i}\rceil\end{aligned}$$

⊣

The first term of the right-hand part in formula (5.1) represents the time required for filling the pipeline (that is, the initial idle time needed for the last processor to start computing), while the second term corresponds to the time each processor is busy executing calculations.

Lemma 5.1 *This schedule is valid iff*

$$w_1^S\prod_{k=l+1}^{n}\lceil\frac{w_k^S}{m_kp_k}\rceil\geq(m_l+1)p_l,$$

$\forall l=2,\ldots,n$ *such that* $w_l^S>m_lp_l$.

Proof: *In order to prove the validity of this schedule, it suffices to prove that the data needed for the computation of a tile are available during the desired time step. If the necessary data are available for the computation of the chunk origins, they will be also available for every inner tile. We assume that tiles are big enough to include all dependence vectors. Thus, each tile depends only on neighboring tiles. A chunk origin has coordinates of the form:* $\vec{j^S}_{origin} = (0, x_2 m_2 p_2, \ldots, x_n m_n p_n)$, *where* $x_i \in N$ $(i = 2, \ldots, n)$. *Thus, it will be executed in the SMP node* $(0, \ldots, 0)$ *during the time step* $t_{origin} = w_1^S \sum_{i=2}^{n} \left[x_i \prod_{k=i+1}^{n} \lceil \frac{w_k^S}{m_k p_k} \rceil \right]$ *(see formula (5.2)). If* $x_l \geq 1$ *(which presupposes* $w_l^S > m_l p_l$*), this chunk origin will be dependent from tile* $\vec{j^S}_{dependence} = (0, x_2 m_2 p_2, \ldots, x_{l-1} m_{l-1} p_{l-1}, x_l m_l p_l - 1, x_{l+1} m_{l+1} p_{l+1}, \ldots, x_n m_n p_n)$, *which will be executed in the SMP node* $(0, \ldots, 0, p_l - 1, 0, \ldots, 0)$ *during the time step* $t_{dependence} = (m_l + 1)p_l - 2 + w_1^S [\sum_{i=2}^{n} [x_i \prod_{k=i+1}^{n} \lceil \frac{w_k^S}{m_k p_k} \rceil] - \prod_{k=l+1}^{n} \lceil \frac{w_k^S}{m_k p_k} \rceil]$. *Since these two tiles will be executed in different SMP nodes, for the necessary data to be available, it must hold* $t_{origin} - t_{dependence} \geq 2 \Leftrightarrow w_1^S \prod_{k=l+1}^{n} \lceil \frac{w_k^S}{m_k p_k} \rceil \geq (m_l + 1)p_l$. *This inequality should be valid* $\forall l = 2, \ldots, n$ *such that* $w_l^S > m_l p_l$. ⊣

If the condition, defined by Lemma 5.1, is not valid, then there is not an actual shortage of processors along dimension l. Thus, we can schedule along this dimension as if there were as many processors as needed. For example, see the difference between Figures 5.2 and 5.3.

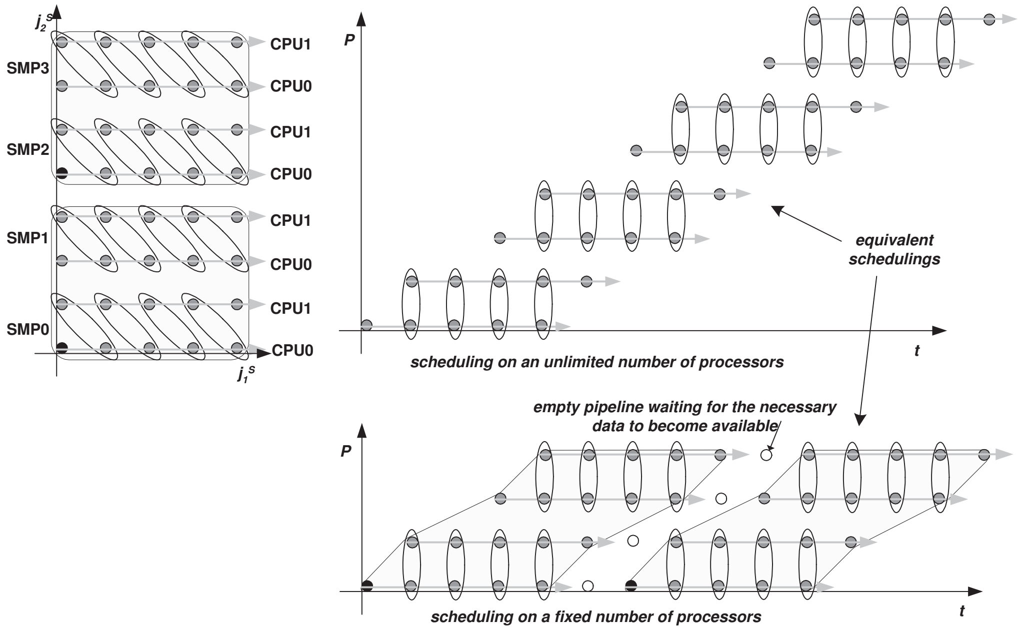


Figure 5.2: Cyclic scheduling when there is not actual lack of processors.
When there are only 2 SMP nodes available, the time steps, when each tile will be computed, do not change at all.

If we should do with a conventional communication architecture as node interconnect (i.e. without NIC support for relieving the CPU from the communication burden):

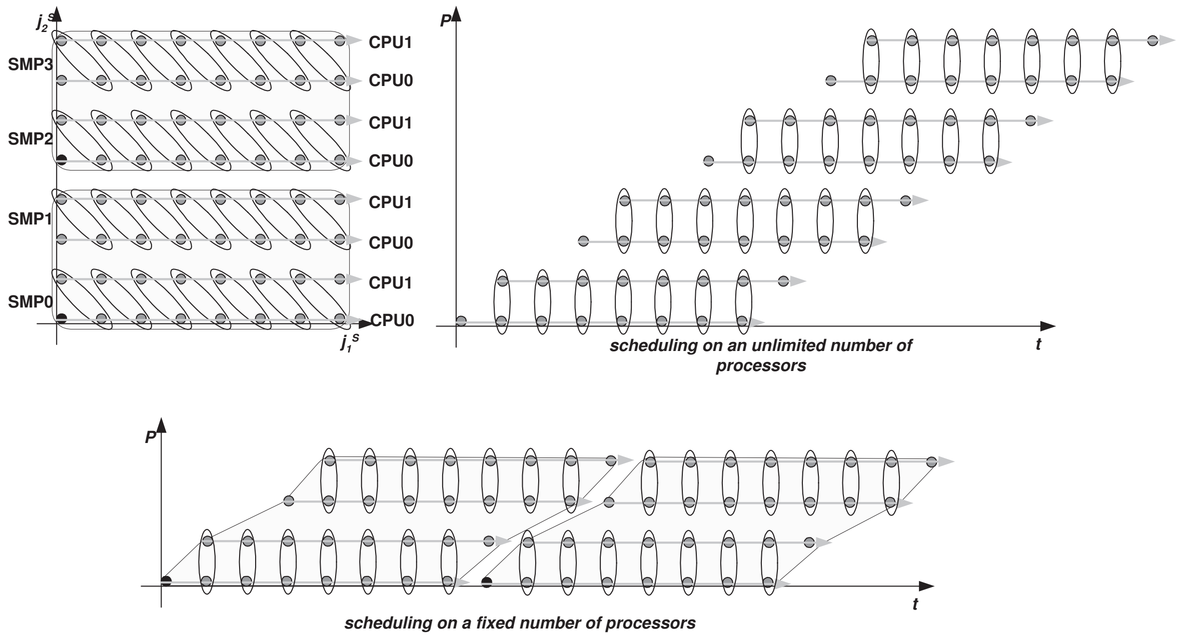


Figure 5.3: Cyclic scheduling when there is lack of processors.
The computation of the second chunk of tiles starts at time step $t = 8$, instead of $t = 6$, according to formula (5.2).

Theorem 5.2 *The makespan of cyclically assigning a rectangular tile space to SMP nodes, following the non-overlapping execution scheme, is:*

$$\begin{aligned} \wp_{cyclic-nonoverlap} &= \sum_{i=2}^{n} \left[(w_i^S - 1)\% m_i p_i\right] + w_1^S \prod_{i=2}^{n} \lceil \frac{w_i^S}{m_i p_i} \rceil \leq \\ &\leq \sum_{i=2}^{n} m_i p_i - n + 1 + w_1^S \prod_{i=2}^{n} \lceil \frac{w_i^S}{m_i p_i} \rceil \end{aligned} \tag{5.3}$$

Proof: *As in the proof of theorem 5.1, the latency before the computation of a group consists of the latency imposed by lexicographically previous rows assigned to the same processor, plus the latency imposed by previous groups of the same row. Consequently, group $\vec{j^G}$ will be computed during the time step*

$$t(\vec{j^G}) = j_1^{G'} + w_1^S \sum_{i=2}^{n} \left[\lfloor \frac{j_i^G}{p_i} \rfloor \prod_{k=i+1}^{n} \lceil \frac{w_k^S}{m_k p_k} \rceil \right] \tag{5.4}$$

Thus, the makespan of the execution will be

$$\begin{aligned}
\wp_{cyclic-nonoverlap} &= \max t(\vec{j^G}) - \min t(\vec{j^G}) + 1 = \\
&\overset{(C.3)}{=} u_1^S + \sum_{i=2}^{n}\left[u_i^S \% m_i p_i\right] + w_1^S \sum_{i=2}^{n}\left[\lfloor \frac{u_i^S}{m_i p_i} \rfloor \prod_{k=i+1}^{n} \lceil \frac{w_k^S}{m_k p_k} \rceil\right] + 1 = \\
&\overset{(C.4)}{=} \sum_{i=2}^{n}\left[(w_i^S - 1)\% m_i p_i\right] + w_1^S + w_1^S \sum_{i=2}^{n}\left[(\lceil \frac{w_i^S}{m_i p_i} \rceil - 1) \prod_{k=i+1}^{n} \lceil \frac{w_k^S}{m_k p_k} \rceil\right] = \\
&\overset{(C.7)}{=} \sum_{i=2}^{n}\left[(w_i^S - 1)\% m_i p_i\right] + w_1^S \prod_{i=2}^{n} \lceil \frac{w_i^S}{m_i p_i} \rceil
\end{aligned}$$

$\dashv$

Lemma 5.2 *The schedule of Theorem 5.2 is always valid, assuming $w_1^S \geq w_i^S$, $i = 2, \ldots, n$.*

Proof: *As in the proof of Lemma 5.1, in order for this schedule to be valid, the data needed for the computation of a tile should be available during the corresponding time step. A chunk origin $\vec{j^S}_{origin} = (0, x_2 m_2 p_2, \ldots, x_n m_n p_n)$ ($x_i \in N$ $(i = 2, \ldots, n)$), will be executed during the time step $t_{origin} = w_1^S \sum_{i=2}^{n}\left[x_i \prod_{k=i+1}^{n} \lceil \frac{w_k^S}{m_k p_k} \rceil\right]$ (see formula (5.4)). If $x_l \geq 1$ (which presupposes $w_l^S > m_l p_l$), this chunk origin will be dependent from tile $\vec{j^S}_{dependence} = (0, x_2 m_2 p_2, \ldots, x_{l-1} m_{l-1} p_{l-1}, x_l m_l p_l - 1, x_{l+1} m_{l+1} p_{l+1}, \ldots, x_n m_n p_n)$, which will be executed during the time step $t_{dependence} = m_l p_l - 1 + w_1^S [\sum_{i=2}^{n} [x_i \prod_{k=i+1}^{n} \lceil \frac{w_k^S}{m_k p_k} \rceil] - \prod_{k=l+1}^{n} \lceil \frac{w_k^S}{m_k p_k} \rceil]$. Since, in the non-overlapping execution scheme, the data are transferred among SMPs during the time step of their computation, for the necessary data to be available, it must hold $t_{origin} - t_{dependence} \geq 1 \Leftrightarrow w_1^S \prod_{k=l+1}^{n} \lceil \frac{w_k^S}{m_k p_k} \rceil \geq m_l p_l$. This inequality is valid $\forall l = 2, \ldots, n$ such that $w_l^S > m_l p_l$, because: $w_1^S \prod_{k=l+1}^{n} \lceil \frac{w_k^S}{m_k p_k} \rceil \geq w_1^S \geq w_l^S > m_l p_l$.* $\dashv$

5.3 Mirror assignment to SMPs

Let us consider another schedule, if we assign the tiles to SMP nodes as indicated in Figure 5.4. That is, we assign group $\vec{j^G}$ to the SMP node

$$\left(\begin{array}{c} j_2^G \% p_2 \texttt{ if } even(j_2^G / p_2) \\ (p_2 - 1) - j_2^G \% p_2 \texttt{ if } odd(j_2^G / p_2) \end{array}, \ldots, \begin{array}{c} j_n^G \% p_n \texttt{ if } even(j_n^G / p_n) \\ (p_n - 1) - j_n^G \% p_n \texttt{ if } odd(j_n^G / p_n) \end{array} \right).$$

This schedule has the advantage that there is no need for data transfer along the boundaries of chunks of tiles, thus less time is wasted for communication.

Theorem 5.3 *When following the mirror assignment schedule, in combination with the over-*

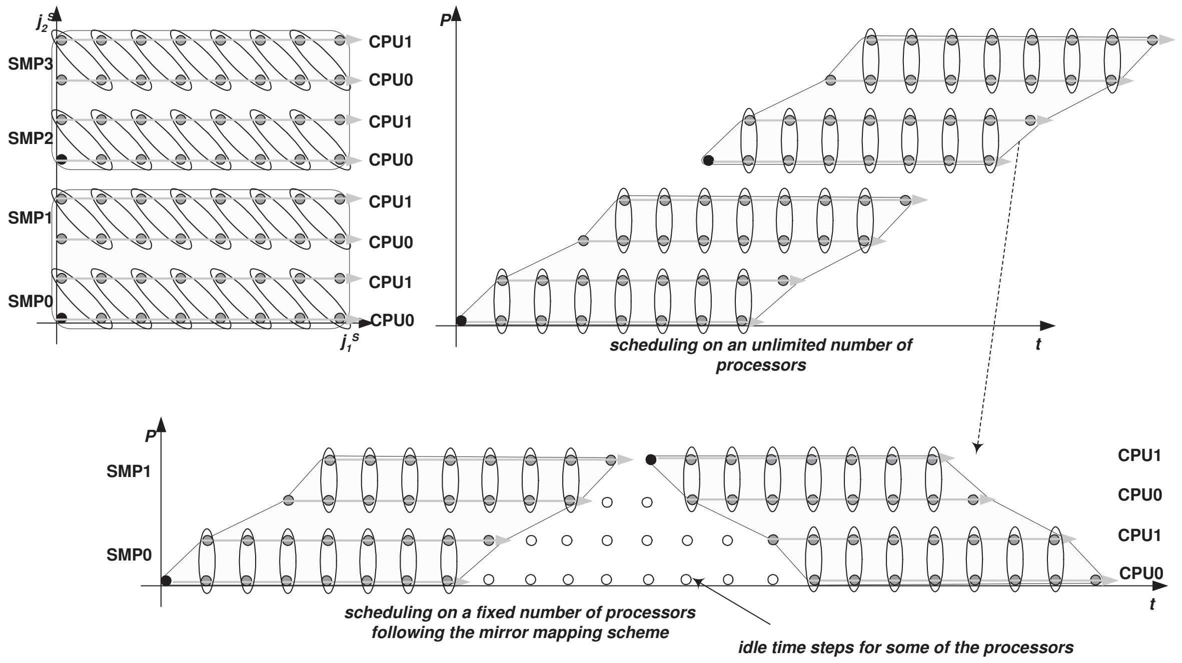


Figure 5.4: Mirror assignment to SMP nodes.

As in the cyclic assignment scheme, the tile space is divided into chunks, which fit the existing processing architecture. The difference is that tiles along the same chunk boundary are assigned to the same SMP node. Thus, there is no need for communication across chunk boundaries.

lapping execution scheme, the makespan is:

$$\begin{aligned}\wp_{mirror-overlap} = & \sum_{i=2}^{n}\left[(w_i^S-1)\%m_ip_i+(\lceil\tfrac{w_i^S}{m_i}\rceil-1)\%p_i\right]-\sum_{i=2}^{n}[(m_i+1)p_i]+2n-2+\\ & +\left[w_1^S+\sum_{i=2}^{n}[(m_i+1)p_i]-2n+2\right]\prod_{i=2}^{n}\lceil\tfrac{w_i^S}{m_ip_i}\rceil\leq\\ & \leq\left[w_1^S+\sum_{i=2}^{n}[(m_i+1)p_i]-2n+2\right]\prod_{i=2}^{n}\lceil\tfrac{w_i^S}{m_ip_i}\rceil\end{aligned}\tag{5.5}$$

Proof: *As in the cyclic assignment schedule, if the chunks of groups are executed in lexicographic order, the chunk containing row $(\bullet, j_2^G, \ldots, j_n^G)$ will be executed after*

$$\sum_{i=2}^{n}\left[\lfloor\frac{j_i^G}{p_i}\rfloor\prod_{k=i+1}^{n}\lceil\frac{w_k^s}{m_kp_k}\rceil\right]$$

chunks. The latency imposed by each of the previous chunks, is greater than the respective one when applying the cyclic assignment schedule. It equals to $w_1^S+\sum_{i=2}^{n}[(m_i+1)p_i]-2n+2$, since the computation of a whole chunk should be finished before the computation of the next chunk starts. In addition, as deduced from Figure 5.4, the position of a group, relatively to the corresponding chunk origin, is $(j_1^{G'}, j_2^G\%p_2, \ldots, j_n^G\%p_n)$, where $j_1^{G'}=j_1^S+\sum_{i=2}^{n}j_i^S\%m_ip_i$.

Therefore, group $\vec{j^G}$ will be computed during the time step

$$t(\vec{j^G}) = j_1^{G'} + \sum_{i=2}^{n} j_i^G \% p_i + \left[w_1^S + \sum_{i=2}^{n} [(m_i+1)p_i] - 2n + 2 \right] \sum_{i=2}^{n} \left[\lfloor \frac{j_i^G}{p_i} \rfloor \prod_{k=i+1}^{n} \lceil \frac{w_k^S}{m_k p_k} \rceil \right]$$

Thus, the makespan will be

$$\begin{aligned}
&\wp_{mirror-overlap} = \max t(\vec{j^G}) - \min t(\vec{j^G}) + 1 = \\
&\stackrel{(C.3)}{=} u_1^S + \sum_{i=2}^{n} \left[u_i^S \% m_i p_i + \lfloor \frac{u_i^S}{m_i} \rfloor \% p_i \right] + \\
&+ \left[w_1^S + \sum_{i=2}^{n} [(m_i+1)p_i] - 2n + 2 \right] \sum_{i=2}^{n} \left[\lfloor \frac{u_i^S}{m_i p_i} \rfloor \prod_{k=i+1}^{n} \lceil \frac{w_k^S}{m_k p_k} \rceil \right] + 1 = \\
&\stackrel{(C.4),(C.7)}{=} \sum_{i=2}^{n} \left[(w_i^S - 1)\% m_i p_i + (\lceil \frac{w_i^S}{m_i} \rceil - 1)\% p_i \right] - \sum_{i=2}^{n} [(m_i+1)p_i] + 2n - 2 + \\
&+ \left[w_1^S + \sum_{i=2}^{n} [(m_i+1)p_i] - 2n + 2 \right] \prod_{i=2}^{n} \lceil \frac{w_i^S}{m_i p_i} \rceil .
\end{aligned}$$

$\dashv$

Following this schedule, there is no need to prove that the data required will be available during the computation of a tile, since,

1. the tiles of a chunk are dependent only on tiles of the same or of a lexicographically previous chunk and,

2. there is no possibility to overlap the computations of different chunks.

If there is no shortage of processors ($w_i^S \leq m_i p_i$, $\forall i = 2, \ldots, n$), the proposed schedules are equivalent. Otherwise, it can be easily deduced from formulas (5.1), (5.5) that $\wp_{cyclic-overlap} < \wp_{mirror-overlap}$. Their difference is due to the fact that, following the mirror assignment schedule, every time the computation of a chunk finishes and the computation of the next one starts, there are some idle time steps for some of the processors, as indicated in Figure 5.4 by white dots. Thus, when a time step for the cyclic schedule is equal to a time step for the mirror one, the cyclic schedule is preferable to the mirror one. In fact, this is the case for the overlapping execution scheme.

Theorem 5.4 *Following the mirror assignment schedule, in combination to the non-overlapping execution scheme, the makespan of the execution is:*

$$\begin{aligned}
&\wp_{mirror-nonoverlap} = \\
&= \sum_{i=2}^{n} [(w_i^S - 1)\% m_i p_i] - \sum_{i=2}^{n} m_i p_i + n - 1 + \left[w_1^S + \sum_{i=2}^{n} m_i p_i - n + 1 \right] \prod_{i=2}^{n} \lceil \frac{w_i^S}{m_i p_i} \rceil \leq \\
&\leq \left[w_1^S + \sum_{i=2}^{n} m_i p_i - n + 1 \right] \prod_{i=2}^{n} \lceil \frac{w_i^S}{m_i p_i} \rceil
\end{aligned} \tag{5.6}$$

Proof: *The latency imposed by each one of the previous chunks is* $w_1^S + \sum_{i=2}^{n} m_i p_i - n + 1$. *Consequently, group* $\vec{j^G}$ *will be computed during the time step* $t(\vec{j^G}) = j_1^{G'} + (w_1^S + \sum_{i=2}^{n} m_i p_i - n + 1) \sum_{i=2}^{n} \left[\lfloor \frac{j_i^G}{p_i} \rfloor \prod_{k=i+1}^{n} \lceil \frac{w_k^S}{m_k p_k} \rceil \right]$. *Thus, the makespan of the execution will be*

$$\wp_{mirror-nonoverlap} = \max t(\vec{j^G}) - \min t(\vec{j^G}) + 1 =$$
$$\stackrel{(C.3)}{=} u_1^S + \sum_{i=2}^{n} \left[u_i^S \% m_i p_i \right] + \left[w_1^S + \sum_{i=2}^{n} m_i p_i - n + 1 \right] \sum_{i=2}^{n} \left[\lfloor \frac{u_i^S}{m_i p_i} \rfloor \prod_{k=i+1}^{n} \lceil \frac{w_k^S}{m_k p_k} \rceil \right] + 1 =$$
$$\stackrel{(C.7)}{=} \sum_{i=2}^{n} \left[(w_i^S - 1) \% m_i p_i \right] - \sum_{i=2}^{n} m_i p_i + n - 1 + \left[w_1^S + \sum_{i=2}^{n} m_i p_i - n + 1 \right] \prod_{i=2}^{n} \lceil \frac{w_i^S}{m_i p_i} \rceil$$

$\dashv$

It can be deduced from formulas (5.3), (5.6) that $P_{cyclic-nonoverlap} \leq P_{mirror-nonoverlap}$. (They are equivalent only in case there is no lack of processors.) However, since the communication overhead is not hidden under the computation time, this schedule may sometimes result in a shorter total execution time, due to better exploitation of the available bandwidth. In particular, if there are only two SMP nodes along a dimension, no SMP node should both send and receive data along that dimension. Thus, the communication overhead will be halved.

5.4 Cluster assignment to SMPs

Alternatively, following the approach of [MA01], generalizing it for n-dimensional spaces and taking into account that there is no need for communication among processors of the same SMP node, we may assign neighboring rows of tiles to the same CPU, as indicated in Figure 5.5.

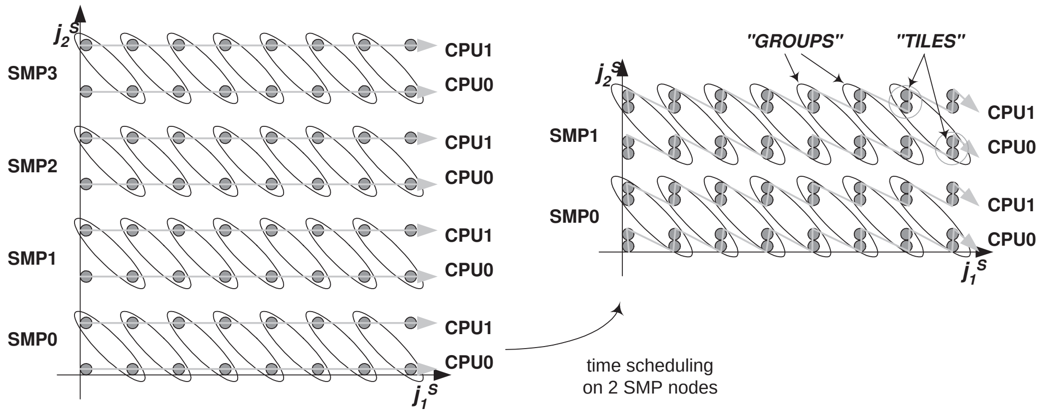


Figure 5.5: Cluster assignment to SMP nodes.
Neighboring tiles, clustered together to TILES, are assigned to the same CPU. Time scheduling does not any more concern tiles or groups, but TILES or GROUPS.

Theorem 5.5 *When following the cluster assignment schedule, in combination to the overlapping execution scheme, the makespan of the execution is:*

$$\wp_{cluster-overlap} = \prod_{i=2}^{n} \lceil \frac{w_i^S}{m_i p_i} \rceil \left(w_1^S - 2n + 2 + \sum_{i=2}^{n} \lceil \frac{w_i^S}{\lceil \frac{w_i^S}{m_i p_i} \rceil} \rceil + \sum_{i=2}^{n} \lceil \frac{w_i^S}{m_i \lceil \frac{w_i^S}{m_i p_i} \rceil} \rceil \right) \tag{5.7}$$

Proof: *In order to achieve this schedule, we cluster together neighboring tiles* $(j_1^S, j_2^S, \ldots, j_n^S)$, *mapping them to a "supertile", or TILE, labelled as* $(j_1^S, \lfloor \frac{j_2^S}{\lceil \frac{w_2^S}{m_2 p_2} \rceil} \rfloor, \ldots, \lfloor \frac{j_n^S}{\lceil \frac{w_n^S}{m_n p_n} \rceil} \rfloor)$. *Thus, the corresponding GROUP will be* $\vec{j^G} = (j_1^S + \sum_{i=2}^{n} \lfloor \frac{j_i^S}{\lceil \frac{w_i^S}{m_i p_i} \rceil} \rfloor, \lfloor \frac{j_2^S}{m_2 \lceil \frac{w_2^S}{m_2 p_2} \rceil} \rfloor, \ldots, \lfloor \frac{j_n^S}{m_n \lceil \frac{w_n^S}{m_n p_n} \rceil} \rfloor)$ *and it will be executed during the time STEP* $t(\vec{j^S}) = j_1^S + \sum_{i=2}^{n} \lfloor \frac{j_i^S}{\lceil \frac{w_i^S}{m_i p_i} \rceil} \rfloor + \sum_{i=2}^{n} \lfloor \frac{j_i^S}{m_i \lceil \frac{w_i^S}{m_i p_i} \rceil} \rfloor$. *Consequently, the MAKESPAN of the algorithm is*

$$\wp_{\text{CLUSTER-OVERLAP}} = \max t(\vec{j^S}) - \min t(\vec{j^S}) + 1 = \\ \stackrel{(C.4)}{=} w_1^S - 2n + 2 + \sum_{i=2}^{n} \lceil \frac{w_i^S}{\lceil \frac{w_i^S}{m_i p_i} \rceil} \rceil + \sum_{i=2}^{n} \lceil \frac{w_i^S}{m_i \lceil \frac{w_i^S}{m_i p_i} \rceil} \rceil$$

As a TILE consists of $\prod_{i=2}^{n} \lceil \frac{w_i^S}{m_i p_i} \rceil$ *tiles, assuming that the duration of a time step is mainly determined by the computation time* t_{comp}, *a STEP will be equivalent to* $\prod_{i=2}^{n} \lceil \frac{w_i^S}{m_i p_i} \rceil$ *time steps (excluding the DMA initialization and synchronization time). Thus, the total number of steps required for the completion of the execution will be*

$$\wp_{cluster-overlap} = \prod_{i=2}^{n} \lceil \frac{w_i^S}{m_i p_i} \rceil \wp_{\text{CLUSTER-OVERLAP}} = \\ = \prod_{i=2}^{n} \lceil \frac{w_i^S}{m_i p_i} \rceil \left(w_1^S - 2n + 2 + \sum_{i=2}^{n} \lceil \frac{w_i^S}{\lceil \frac{w_i^S}{m_i p_i} \rceil} \rceil + \sum_{i=2}^{n} \lceil \frac{w_i^S}{m_i \lceil \frac{w_i^S}{m_i p_i} \rceil} \rceil \right)$$

⊣

Lemma 5.3 *It holds that* $\wp_{cyclic-overlap} \leq \wp_{cluster-overlap}$.

Proof: *When there is no lack of processors* ($w_i^S \leq m_i p_i$, $\forall i = 2, \ldots, n$), *the proposed schemes are equivalent and it can be easily proven from (5.1), (5.7) that*

$$\wp_{cyclic-overlap} = \wp_{cluster-overlap}$$

Otherwise, (5.7) $\Rightarrow$

$$\wp_{cluster-overlap} > \sum_{i=2}^{n} \left[\lceil \frac{w_i^S}{\lceil \frac{w_i^S}{m_i p_i} \rceil} \rceil - 1 + \lceil \frac{w_i^S}{m_i \lceil \frac{w_i^S}{m_i p_i} \rceil} \rceil - 1 \right] + w_1^S \prod_{i=2}^{n} \lceil \frac{w_i^S}{m_i p_i} \rceil.$$

If we write $w_i^S = x_i m_i p_i - y_i$, *where* x_i, y_i *are integer numbers and* $x_i \geq 1$, $0 \leq y_i < m_i p_i - 1$, *then it holds that:*

$$\left.\begin{array}{l}\left.\begin{array}{r}(w_i^S-1)\%m_ip_i = m_ip_i - y_i - 1\\ \lceil\frac{w_i^S}{\lceil\frac{w_i^S}{m_ip_i}\rceil}\rceil - 1 \overset{(C.5)}{=} m_ip_i - \lfloor\frac{y_i}{x_i}\rfloor - 1\end{array}\right\} \Rightarrow (w_i^S-1)\%m_ip_i \leq \lceil\frac{w_i^S}{\lceil\frac{w_i^S}{m_ip_i}\rceil}\rceil - 1\\ \left.\begin{array}{r}(\lceil\frac{w_i^S}{m_i}\rceil - 1)\%p_i \overset{(C.5)}{=} p_i - \lfloor\frac{y_i}{m_i}\rfloor - 1\\ \lceil\frac{w_i^S}{m_i\lceil\frac{w_i^S}{m_ip_i}\rceil}\rceil - 1 \overset{(C.5)}{=} p_i - \lfloor\frac{y_i}{m_ix_i}\rfloor - 1\end{array}\right\} \Rightarrow (\lceil\frac{w_i^S}{m_i}\rceil - 1)\%p_i \leq \lceil\frac{w_i^S}{m_i\lceil\frac{w_i^S}{m_ip_i}\rceil}\rceil - 1\end{array}\right\} \Rightarrow$$

$$\Rightarrow \wp_{cyclic-overlap} < \wp_{cluster-overlap}.$$

⊣

Thus, this schedule results to a worse makespan than the cyclic one. Their difference is due to the fact that, in this schedule, the filling of the pipeline is slower (that is, the last processor starts executing computations later). In case $w_1^S >> w_i^S$ $(i = 2, \ldots, n)$, the time each processor is busy, outflanks the pipeline filing time and it holds that $P_{cyclic-overlap} \simeq P_{cluster-overlap}$. However, the previous mathematical lemma has not taken into consideration the time required for the initialization of messages and for synchronization. Since the cluster assignment schedule requires less messages to be sent and less synchronization, in some cases it may be practically proven more efficient.

Theorem 5.6 *Following the cluster assignment schedule, in combination to the non-overlapping execution scheme, the makespan of the execution is:*

$$\wp_{cluster-nonoverlap} = C\left(w_1^S - n + 1 + \sum_{i=2}^{n}\lceil\frac{w_i^S}{\lceil\frac{w_i^S}{m_ip_i}\rceil}\rceil\right) \leq C\left(w_1^S - n + 1 + \sum_{i=2}^{n} m_ip_i\right) \tag{5.8}$$

where $1 \leq C \leq \prod_{i=2}^{n}\lceil\frac{w_i^S}{m_ip_i}\rceil$

Proof: *Tile* $(j_1^S, j_2^S, \ldots, j_n^S)$*, corresponding to GROUP*

$$\vec{j^G} = (j_1^S + \sum_{i=2}^{n}\lfloor\frac{j_i^S}{\lceil\frac{w_i^S}{m_ip_i}\rceil}\rfloor, \lfloor\frac{j_2^S}{m_2\lceil\frac{w_2^S}{m_2p_2}\rceil}\rfloor, \ldots, \lfloor\frac{j_n^S}{m_n\lceil\frac{w_n^S}{m_np_n}\rceil}\rfloor)$$

is executed during the time STEP $t(\vec{j^S}) = j_1^S + \sum_{i=2}^{n}\lfloor\frac{j_i^S}{\lceil\frac{w_i^S}{m_ip_i}\rceil}\rfloor$*. Consequently, the MAKESPAN of the execution is*

$$\wp_{\text{CLUSTER-NONOVERLAP}} = \max t(\vec{j^S}) - \min t(\vec{j^S}) + 1 \overset{(C.4)}{=} w_1^S - n + 1 + \sum_{i=2}^{n}\lceil\frac{w_i^S}{\lceil\frac{w_i^S}{m_ip_i}\rceil}\rceil.$$

A computation subSTEP is equivalent to $\prod_{i=2}^{n}\lceil\frac{w_i^S}{m_ip_i}\rceil$ *computation substeps, but a communication subSTEP is equivalent to less than* $\prod_{i=2}^{n}\lceil\frac{w_i^S}{m_ip_i}\rceil$ *communication substeps. In particular,* *if the communication load is equal along all communication dimensions (as resulted by the method proposed in [Xue97a]), the amount of data to be transferred, as indicated in Figure 5.6, is* $\prod_{i=2}^{n}\lceil\frac{w_i^S}{m_i p_i}\rceil\sum_{i=2}^{n}\frac{1}{(n-1)\lceil\frac{w_i^S}{m_i p_i}\rceil} \leq \prod_{i=2}^{n}\lceil\frac{w_i^S}{m_i p_i}\rceil$ *times the communication load of a tile. Thus, the makespan of the algorithm will be*

$$\wp_{cluster-nonoverlap} = C\wp_{\text{CLUSTER-NONOVERLAP}} \text{ (where } 1 \leq C \leq \prod_{i=2}^{n}\lceil\frac{w_i^S}{m_i p_i}\rceil) \Rightarrow$$
$$\wp_{cluster-nonoverlap} = C\left(w_1^S - n + 1 + \sum_{i=2}^{n}\lceil\frac{w_i^S}{\lceil\frac{w_i}{m_i p_i}\rceil}\rceil\right)$$

⊣

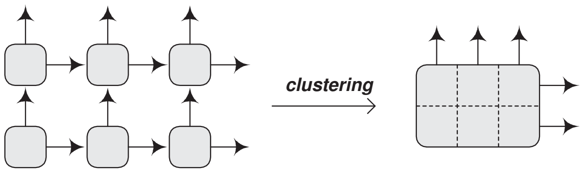


Figure 5.6: Clustering communication

In conclusion, comparing to the cyclic assignment schedule, this method has the drawback of slower pipeline filling. However, it results to less communication overhead, which significantly reduces the total execution time, especially when the non-overlapping execution scheme is applied.

5.5 Retiling

A more efficient schedule can be obtained, if we adapt the size of tiles to the available number of SMPs (Figure 5.7). That is, we retile the initial iteration space, so as to get $w_i^{S'} = m_i p_i$, $(i = 2, \ldots, n)$ and $w_1^{S'} = w_1^S \prod_{i=2}^{n}\frac{w_i^S}{m_i p_i}$. Then, the size of a "new" tile will be equal to the size of an "old" tile and, consequently, a "new" computation step will be equivalent to an "old" computation step. Following the overlapping execution scheme, the number of time steps required for the completion of the execution, according to formula (4.3), will be $\wp_{retile-overlap} = \sum_{i=1}^{n} w_i^{S'} + \sum_{i=2}^{n}\lceil\frac{w_i^{S'}}{m_i}\rceil - 2n + 2 \Rightarrow$

$$\wp_{retile-overlap} = \sum_{i=2}^{n}\left[(m_i + 1)\, p_i\right] - 2n + 2 + w_1^S \prod_{i=2}^{n}\frac{w_i^S}{m_i p_i} \tag{5.9}$$

In case $w_i^S \% m_i p_i = 0$ $(i = 2, \ldots, n)$, it holds that $\wp_{retile-overlap} = \wp_{cyclic-overlap}$. Otherwise, $\wp_{retile-overlap} < \wp_{cyclic-overlap}$. Their difference is due to the fact that the cyclic schedule does not assign exactly the same number of tiles to each processor, resulting to a slight load imbalance.

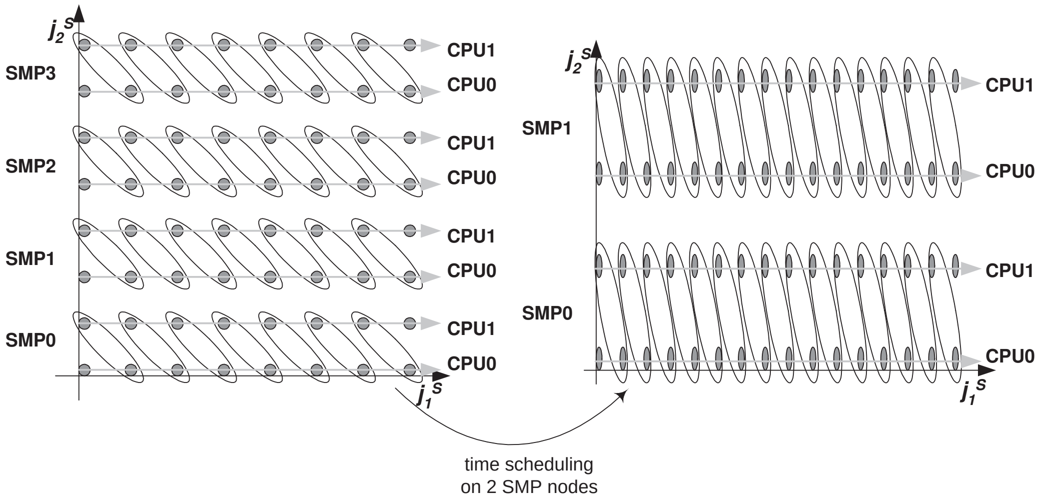


Figure 5.7: Retiling.
The tile space is re-constructed from scratch, so as to fit the existing processing architecture.

Using the non-overlapping execution scheme, the number of time steps required for the completion of the execution, according to formula (4.4), will be $\wp_{retile-nonoverlap} = \sum_{i=1}^{n} w_i^{S'} - n + 1 \Rightarrow$

$$\wp_{retile-nonoverlap} = \sum_{i=2}^{n} m_i p_i - n + 1 + w_1^S \prod_{i=2}^{n} \frac{w_i^S}{m_i p_i} \tag{5.10}$$

From (5.3), (5.10), we can deduce that $\wp_{retile-nonoverlap} \leq \wp_{cyclic-nonoverlap}$. In addition, a "new" computation substep is equivalent to an "old" computation substep, but a "new" communication substep is equivalent to less than an "old" communication substep. In particular, as in Theorem 5.6, if the communication load is equal along all communication dimensions, the amount of data to be transferred is $\sum_{i=2}^{n} \frac{1}{(n-1)\frac{w_i^S}{m_i p_i}} \leq 1$ times the communication load of an "old" tile.

In conclusion, when the tile space is rectangular, this schedule is preferable to previously proposed ones, assuming that there are no factors constraining the tile shape, such as false sharing, or cache locality [KRC99], [LRW91], [WL91a], [MHCF98], [PHP03]. It can fully exploit the computational power of all the SMP nodes and it achieves a perfect load balance, without imposing any additional complexity to the initial schedule, at least when a rectangular tile space is concerned. But if, apart from parallel scheduling, there are other factors constraining the tile size and shape, this schedule may prove to be inefficient, since it totally reorganizes the execution order of iterations.

5.6 Experimental Results

5.6.1 Experimental Platform

In order to evaluate the proposed methods, we use a Linux SMP cluster with 2 identical nodes. Each node has 1GB of RAM and 2 Pentium III @ 1266 MHz CPUs. The cluster nodes communicate through a Myrinet high performance interconnect, using the GM low level message passing system.

In order to utilize the available processors in each SMP node as efficiently as possible, our implementation uses one multi-threaded process per SMP, with the number of threads equal to the number of CPUs. Multithreading support is based on the LinuxThreads library. Threads executing on the same SMP communicate using shared memory, eliminating the need for message passing. For the data exchange between processes executing on different SMPs, Myricom's GM version 1.6.3 is used [Myr02]. GM is a low-level message passing library for Myrinet. It comprises a library used by userspace programs, an OS driver (in our case, a Linux kernel module) and a Myrinet Control Program (MCP), which is executed on the LANai, the embedded RISC microprocessor on the Myrinet NIC. The GM driver is used during the execution of a userspace process to open and close *ports* and to allocate and free memory suitable for DMA transfers. A port is a communication endpoint, used as the interface between a userspace process and the NIC. Having opened a port, a process can communicate directly with the NIC, without the need for system calls, bypassing the operating system. Thus, all data exchange is performed directly to and from userspace buffers.

To provide flow control between the host and the NIC, sending and receiving messages is regulated by tokens. Initially, a process possesses a finite number of send and receive tokens. To be able to receive a message, the process must provide GM with a buffer in DMAable memory, relinquishing a receive token. When a message is received, the DMA engine on the Myrinet NIC places it directly into the userspace buffer. The process polls for new messages and retrieves the receive token when a message arrives. The same applies to sending messages: The process relinquishes a send token by requesting the transmission of a message from a userspace buffer, then retrieves it when the send operation completes and an appropriate send completion callback function is executed by GM. As the data exchange between the host memory and the NIC is undertaken by the DMA engine on the NIC, without involving the CPU, overlapping of communication with computation is possible.

5.6.2 Experimental Data: Rectangular Tile Spaces

We performed several series of experiments in order to evaluate and compare the practical speedups obtained using each one of the four alternative schedules, combined with both the alternative execution schemes. Our test application code was the following:

```
for(i=1; i<=X; i++)
  for(j=1; j<=Y; j++)
    for(k=1; k<=Z; k++)
      A[i][j][k]=func(A[i-1][j][k],A[i][j-1][k],A[i][j][k-1]);
```

where A is an array of $X \times Y \times Z$ floats and $X = Y << Z$. Without lack of generality, we consider, as a tile, a rectangle with ij, ik and jk sides. The dimension k is the largest one, so all tiles along the k-axis are mapped onto the same processor, as proposed in [AKPT99], [GSK01]. Each tile has i, j, k dimensions equal to x. Thus, there are $\frac{X}{x}$ tiles along dimensions i, j and $\frac{Z}{x}$ tiles along dimension k. Tile's volume is equal to $g = x^3$. As described in [HS98], g has been selected, so that $t_{comp} = t_{comm}$, after experimentally measuring the computation time per iteration, the time required per data item to be transferred and the communication initialization and finalization overhead.

After implementing all four schedules in combination with both execution schemes, as described by the pseudo-code of Tables 5.1, 5.2, we measured the performance of all schedules and compared it with their theoretically expected performance. For various tile sizes, we have conducted a series of experiments for each schedule+execution scheme combination, varying the iteration space size. In Figures 5.8-5.10 we have plotted our experimental results along with the respective theoretical curves. As a measure of performance, we have used the ratio of the speedup obtained to the best possible speedup. That is, we have depicted the ratio of the speedup obtained to the number of processors used. Thus, the closer a plot is to 1, the more efficient a schedule is. As can be seen in Figures 5.8-5.10, the practical completion times of our experiments differ to our theoretical predictions by at most 3%. For the overlapping communication schedules, this can be attributed to both the DMA engine on the Myrinet NIC and the CPU trying to access data in memory.

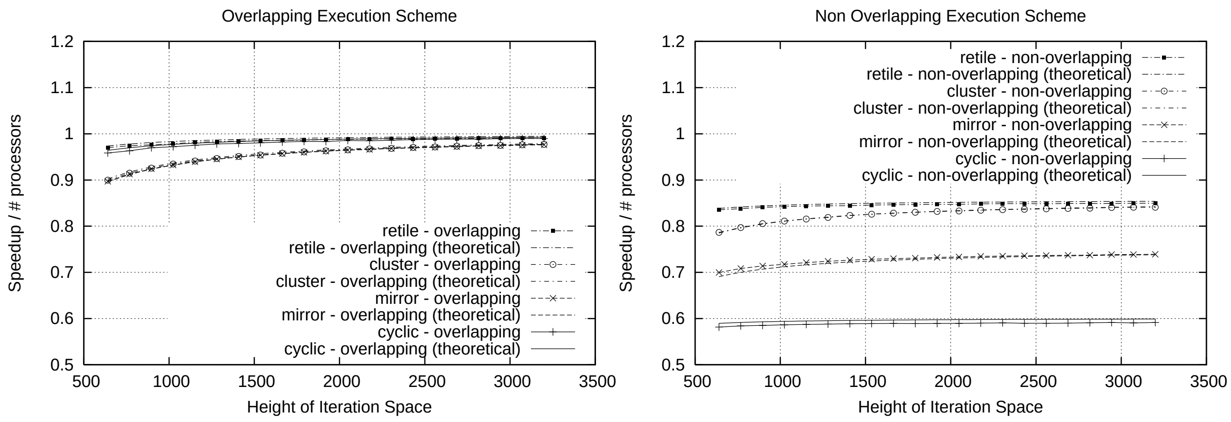


Figure 5.8: Experimental Data: Tile Size $32 \times 32 \times 32$

One can easily deduce that in almost all cases, the retiling schedule achieves the best performance, both theoretically and experimentally. This result was expected, since the retiling schedule absolutely adjusts tiles to the existing configuration of a cluster. However, in our ex-

Table 5.1: Implementation of schedules (cyclic assignment, mirror assignment, cluster assignment to SMP nodes) when the tile space is rectangular

Cyclic Assignment - Rectangular Tile Space

FOREACH CPU with coordinates $(cpu\_id_2, \ldots, cpu\_id_n)$
    in SMP node with coordinates $(smp\_id_2, \ldots, smp\_id_n)$ DO
  FOR $(t_2 = smp\_id_2 * m_2 + cpu\_id_2;\ t_2 < w_2^S;\ t_2 += m_2 * p_2)$
    FOR $(t_3 = smp\_id_3 * m_3 + cpu\_id_3;\ t_3 < w_3^S;\ t_3 += m_3 * p_3)$
      FOR $(t_1 = 0;\ t_1 < w_1^S;\ t_1{+}{+})\{$
        Execute pre-computation part of Communication
        Execute Computation of tile (t_1, t_2, t_3)
        Execute post-computation part of Communication
      }

Mirror Assignment - Rectangular Tile Space

FOREACH CPU with coordinates $(cpu\_id_2, \ldots, cpu\_id_n)$
    in SMP node with coordinates $(smp\_id_2, \ldots, smp\_id_n)$ DO
  FOR $(x_2 = 0;\ x_2 < \lceil \frac{w_2^S}{m_2 * p_2} \rceil;\ x_2{+}{+})\{$
    $t_2 = x_2 * m_2 * p_2 + (1 - x_2\%2) * (smp\_id_2 * m_2 + cpu\_id_2) + (x_2\%2) * (m_2 * p_2 - 1 - smp\_id_2 * m_2 - cpu\_id_2);$
    IF $(t_2 < w_2^S)$
      FOR $(x_3 = 0;\ x_3 < \lceil \frac{w_3^S}{m_3 * p_3} \rceil;\ x_3{+}{+})\{$
        $t_3 = x_3 * m_3 * p_3 + (1 - x_3\%2) * (smp\_id_3 * m_3 + cpu\_id_3) + (x_3\%2) * (m_3 * p_3 - 1 - smp\_id_3 * m_3 - cpu\_id_3);$
        IF $(t_3 < w_3^S)\{$
          Execute pre-computation part of Communication
          Execute Computation of tile (t_1, t_2, t_3)
          Execute post-computation part of Communication
        }
      }
  }

Cluster Assignment - Rectangular Tile Space

FOREACH CPU with coordinates $(cpu\_id_2, \ldots, cpu\_id_n)$
    in SMP node with coordinates $(smp\_id_2, \ldots, smp\_id_n)$ DO
  FOR $(t_1 = 0;\ t_1 < w_1^S;\ t_1{+}{+})\{$
    Execute pre-computation part of Communication
    FOR $(t_2 = (smp\_id_2 * m_2 + cpu\_id_2) * \lceil \frac{w_2^S}{m_2 * p_2} \rceil;$
        $t_2 < min(w_2^S, (smp\_id_2 * m_2 + cpu\_id_2 + 1) * \lceil \frac{w_2^S}{m_2 * p_2} \rceil);\ t_2{+}{+})$
      FOR $(t_3 = (smp\_id_3 * m_3 + cpu\_id_3) * \lceil \frac{w_3^S}{m_3 * p_3} \rceil;$
          $t_3 < min(w_3^S, (smp\_id_3 * m_3 + cpu\_id_3 + 1) * \lceil \frac{w_3^S}{m_3 * p_3} \rceil);\ t_3{+}{+})\{$
        Execute Computation of tile (t_1, t_2, t_3)
      }
    Execute post-computation part of Communication
  }

Retiling - Rectangular Tile Space

$w_1^S * = \frac{w_2^S}{m_2 * p_2} * \frac{w_3^S}{m_3 * p_3}$
$w_2^S = m_2 * p_2$
$w_3^S = m_3 * p_3$
FOREACH CPU with coordinates $(cpu\_id_2, \ldots, cpu\_id_n)$
    in SMP node with coordinates $(smp\_id_2, \ldots, smp\_id_n)$ DO{
  $t_2 = smp\_id_2 * m_2 + cpu\_id_2;$
  $t_3 = smp\_id_3 * m_3 + cpu\_id_3;$
  FOR $(t_1 = 0;\ t_1 < w_1^S;\ t_1{+}{+})\{$
      Execute pre-computation part of Communication
      Execute Computation of tile (t_1, t_2, t_3)
      Execute post-computation part of Communication
  }
}

Table 5.2: Execution schemes implementation (overlapping vs. non-overlapping) using the GM low level message passing system

Non Overlapping Execution Scheme	***Overlapping Execution Scheme***
`Pre-computation Part of Communication`	

`gm_provide_receive_buffer()`
`do`
`  poll the GM event queue`
`  process the event`
`until data received`
	`If on first tile`
	`  Execute a non-overlapping receive`
	`gm_provide_receive_buffer() for tile` (t_1+1, t_2, t_3)
	`gm_send_with_callback() for tile` (t_1-1, t_2, t_3)

`Post-computation Part of Communication`	

`gm_send_with_callback()`
`do`
`  poll the GM event queue`
`  process the event`
`until data sent`
`Barrier for Threads in SMP`
	`do`
	`  poll the GM event queue`
	`  process the event`
	`until send & receive completed`
	`Barrier for Threads in SMP`
	`If on last tile`
	`  Execute a non-overlapping send`

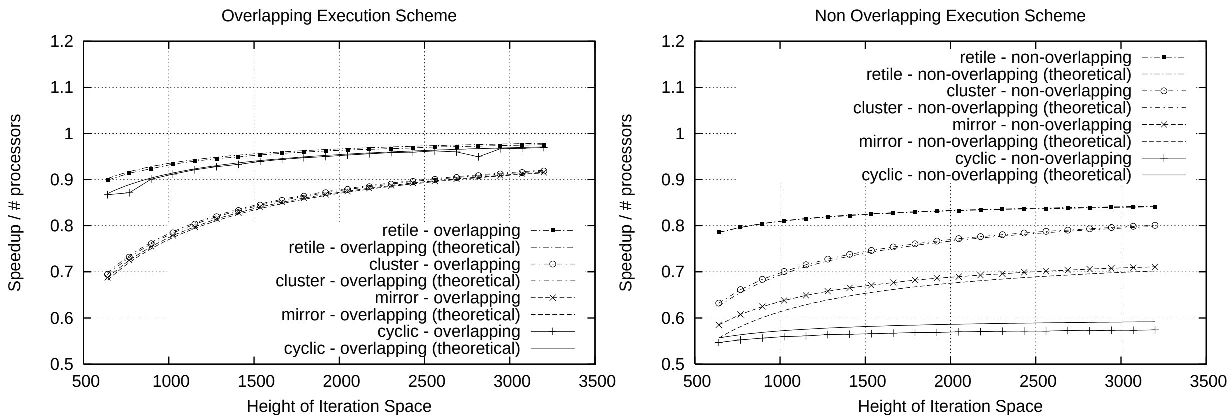


Figure 5.9: Experimental Data: Tile Size $128 \times 32 \times 32$

periments we have eliminated the effect of cache miss penalties by using small iteration space widths. If our iteration space dimensions, which are not assigned to the same processor, were too long, the retiling schedule could have destroyed the data locality achieved by optimally selected small tiles.

Note also that in the above examples the cluster assignment schedule, using tile size x, is equivalent to the retiling schedule, using tile size $4x$. This was expected, considering that by construction the iterations executed and the data sent in these two cases are the same. What differs is the execution order of iterations but here we have eliminated the cache misses overhead, in order to test the optimality of our schedules and not data locality.

When following the non-overlapping execution scheme, the difference among the performance of the four schedules is mainly due to the volume of the data to be transferred. As depicted in Figure 5.11, the mirror assignment schedule involves double the communication of retiling and cluster assignment schedule, while the cyclic assignment schedule involves 6 times the same communication volume.

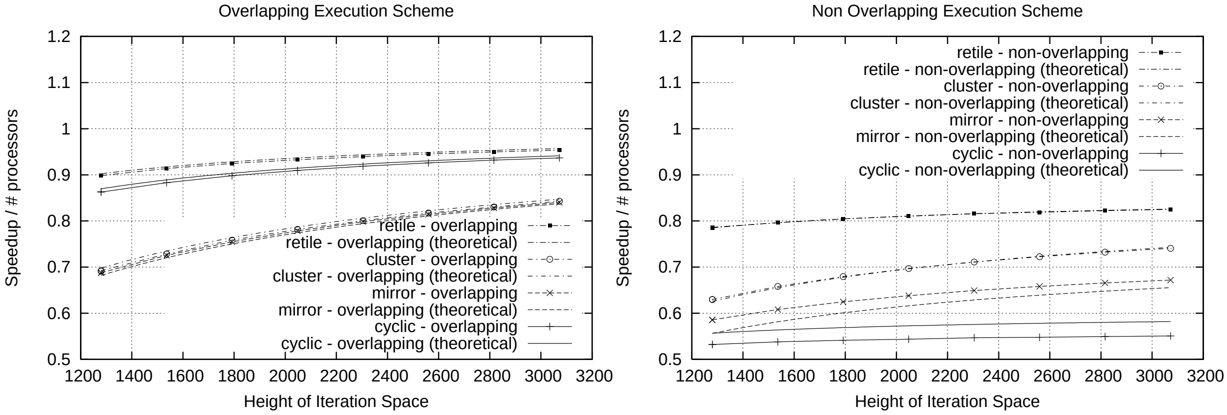


Figure 5.10: Experimental Data: Tile Size $256 \times 32 \times 32$

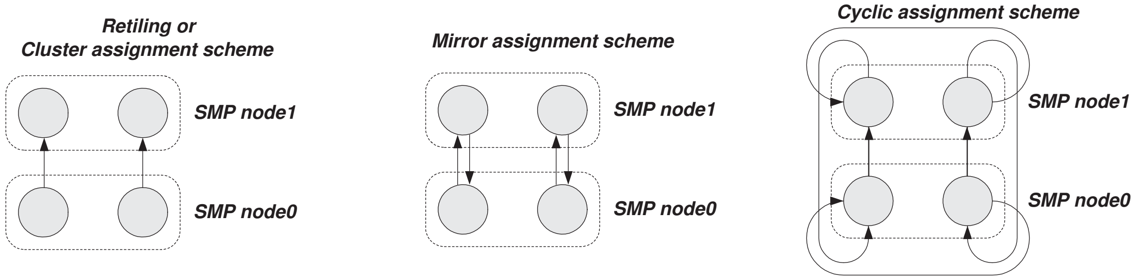


Figure 5.11: Communication among SMPs

When following the overlapping execution scheme, since the communication volume is hidden under computation, their difference is due to the time steps that each SMP has to stall waiting for the required data to arrive. The number of these time steps are equal regarding the retiling and the cyclic assignment schedules. However, using the cluster or the mirror assignment schedule, the number of idle time steps (see Figures 5.3, 5.4) is multiplied by the number of tiles clustered together, or, equivalently, the number of clunks of tiles, which fit the processing architecture.

In addition, note that all schedules achieve better performance for long iteration spaces. This is due to the fact that, when the mapping dimension of the iteration space is comparatively short, the time required for the last processor to start computing after the first data have arrived, is not minor in comparison to the total execution time.

5.6.3 Simulation Data

The previous experimental data have been obtained on a cluster of 2 SMP nodes with 2 CPUs each. Note in Figure 5.11 that in the retiling and the cluster assignment schedule there is no SMP node that should both send and receive data. Thus, we expect that the relative performance of the four schedules would change when scaling up our underlying architecture. In order to evaluate the merits of the proposed schedules, using bigger clusters than the one we had available, we performed a number of simulations, whose results are depicted in Figures 5.12-

5.14. The performance of all four schedules has been simulated assuming that the initialization of DMA and synchronization overhead is negligible, as deduced from microbenchmarking in our platform.

In particular, all measurements of time intervals have been based on the `rdtsc` (Read TimeStamp Counter) instruction, which is available on all Intel processors beyond Pentium. This instruction returns the value of a 64-bit register which is incremented every clock cycle. Since `rdtsc` can be called directly by a userspace process, we do not incur the overhead of the `gettimeofday` system call. Thus, we have measured: 400 cycles for the `send_with_callback` function, which is $0.316\mu sec$ on a PIII@1266MHz, 800 cycles for `gm_provide_receive_buffer`, which is $0.632\mu sec$ and 5598 cycles for a barrier, which is $4.421\mu sec$. Thus, the total non-overlappable communication latency imposed to each tile is less than $6\mu sec$ in the worst case. This overhead is negligible in comparison to a tile computation, which, in all cases, needed more than $24msec$.

Similar to Figures 5.8-5.10, the values plotted in Figures 5.12-5.14 express, for each proposed schedule, the speedup obtained, divided by the number of CPUs used: $\frac{Speedup}{Number\ of\ Processors\ Used}$. Therefore, the closest a plot is to 1, the more efficient the corresponding schedule will be.

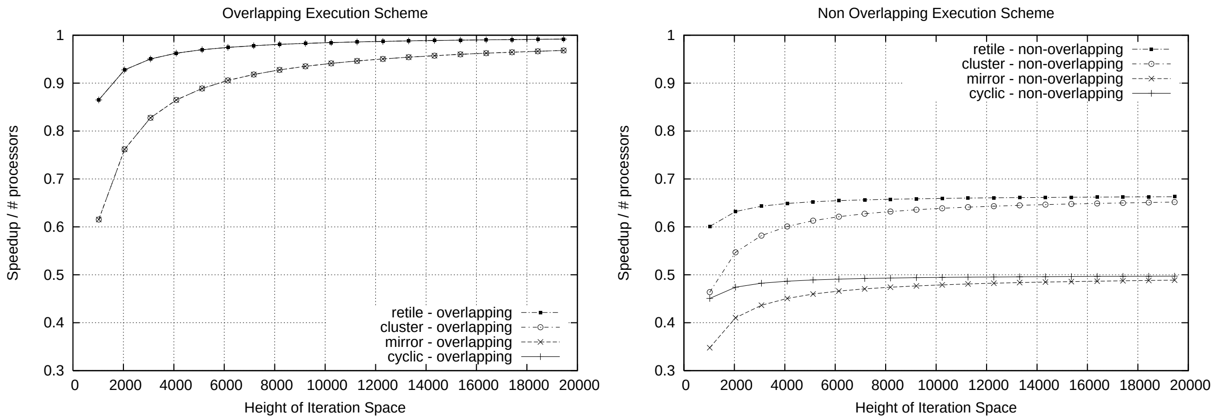


Figure 5.12: Simulation Data: Tile Space $\cdots \times 16 \times 16$ on a grid of 4×4 nodes with 2×2 CPUs each

It can be easily seen that when we are not interested in possible cache miss penalties imposed by reorganizing the tile space, the retiling schedule is again the most efficient one, due to the fact that it can fully exploit the computational power of all the SMP nodes and by definition it achieves a perfect load balance.

As far as the cluster assignment schedule is concerned, for small tile spaces, it is inefficient due to its slow pipeline filling. However, when the mapping dimension of the tile space is long enough, this schedule achieves high speedups, due to the fact that it minimizes the volume of data to be transferred. In fact, as explained in §5.6.2, the plot representing the cluster assignment schedule will fall onto the plot representing the retiling schedule if we shift it parallely to the x-axis (see Figures 5.12, 5.14). The cluster assignment schedule is less efficient than the retiling

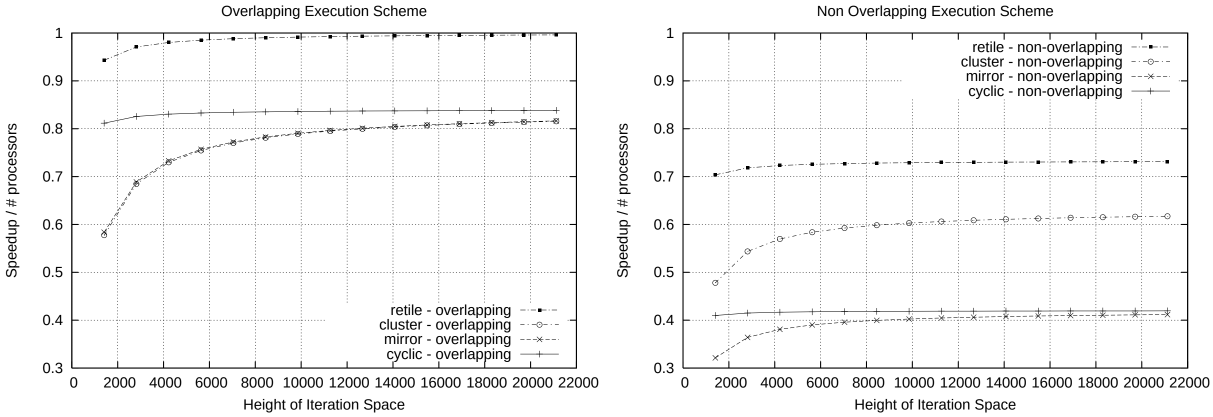


Figure 5.13: Simulation Data: Tile Space $\cdots \times 22 \times 22$ on a grid of 4×4 nodes with 2×2 CPUs each

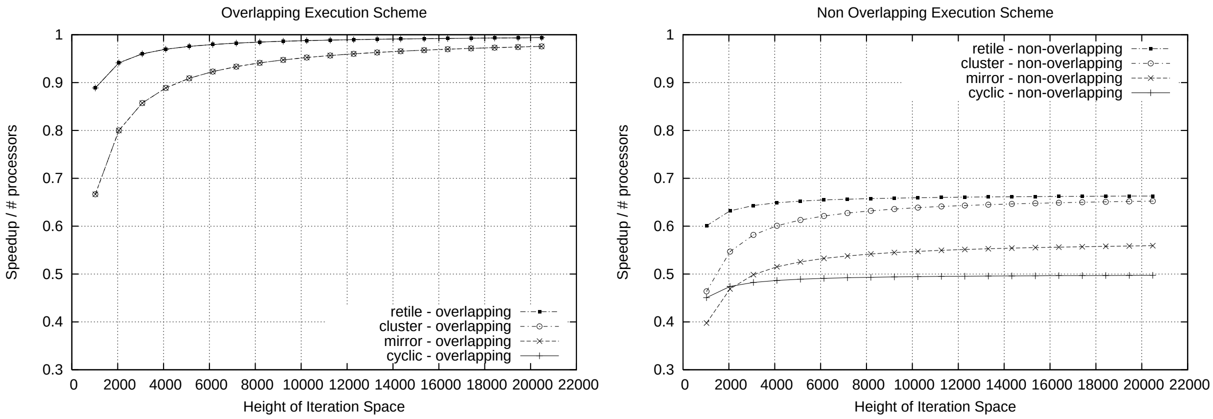


Figure 5.14: Simulation Data: Tile Space $\cdots \times 16 \times 16$ on a grid of 2×2 nodes with 4×4 CPUs each

schedule, only in case w_i^S is not a multiple of $m_i p_i$ (see Figure 5.13), due to load imbalance.

We also deduce that the cyclic assignment schedule is equivalent to the retiling schedule, when the number of tiles along each dimension i is a multiple of $m_i p_i$ and the overlapping execution scheme is used. Otherwise, if w_i^S is not a multiple of $m_i p_i$, their difference is due to the fact that the cyclic schedule does not achieve a perfect load balance. Using the non-overlapping execution scheme, the difference is due to the fact that, as analyzed in Figure 5.6 and §5.5, the cyclic schedule results to more communication load, which is not hidden under the computation load. In addition, it can be more efficient than the cluster assignment schedule, only in case we use the overlapping communication scheme. This is due to the fact that in this case the extra communication overhead of the cyclic schedule is hidden under the computation load.

The mirror assignment schedule is almost always the least efficient, apart from the case of using the non-overlapping execution scheme on a grid of 2×2 SMP nodes. Even then, it is not more efficient than the cluster assignment schedule. This is due to the fact that it combines the disadvantages of the cyclic schedule with the disadvantages of the cluster assignment schedule.

That is, there is at least one node, which has both to send and to receive data (unless there are at most two nodes along each dimension of the grid, as in Figures 5.8-5.10 and Figure 5.14), thus the duration of a time step is equal to the one of the cyclic schedule and the improvement in the exploitation of the existing bandwidth is minor. In addition, after all SMP nodes have started their execution, there are some idle time steps for some of them (see Figure 5.4), corresponding to the slower pipeline filling of the cluster assignment schedule.

5.7 Block-cyclic assignment to SMPs

Since, as shown in §5.6.2-§5.6.3, apart from retiling, the best performance is given by either the cyclic or the cluster assignment schedule, we also designed a combination of these schedules: block-cyclic assignment schedule. So, we hope to achieve the happy medium between them. Especially when dealing with non-rectangular tile spaces, block-cyclic schedule is supposed to achieve low communication overhead (as the cluster assignment schedule does), and at the same time relatively good load balance (as the cyclic assignment schedule does).

As shown in Figure 5.15, block-cyclic schedule is formed by clustering together some neighboring tiles, as we did in the cluster assignment schedule. For example, in Figure 5.15, we cluster together $b_2 = 2$ tiles. The difference, in comparison to the cluster schedule, lies in the fact that now we do not cluster together so many tiles, as to get a number of rows of TILES equal to the number of CPUs available. In the sequel, we cyclically schedule TILES, or GROUPS, similarly to scheduling tiles or groups according to the cyclic assignment schedule.

Theorem 5.7 *The makespan of block-cyclically assigning a rectangular tile space to SMP nodes, assuming overlapping communication with computation is:*

$$\wp_{block-cyclic-overlap} = \left[\sum_{i=2}^{n}\left[(\lceil\frac{w_i^S}{b_i}\rceil - 1)\%m_ip_i + (\lceil\frac{w_i^S}{b_im_i}\rceil - 1)\%p_i\right] + w_1^S\prod_{i=2}^{n}\lceil\frac{w_i^S}{b_im_ip_i}\rceil\right]\prod_{i=2}^{n} b_i \quad (5.11)$$

Proof: *In order to achieve this schedule, we cluster together* $b_2 \times \cdots \times b_n$ *neighboring tiles* $(j_1^S, j_2^S, \ldots, j_n^S)$*, mapping them to TILE labelled as* $(j_1^S, \lfloor\frac{j_2^S}{b_2}\rfloor, \ldots, \lfloor\frac{j_n^S}{b_n}\rfloor)$*. The boundaries of the consequent TILE Space are* $0..u_1^S = w_1^S - 1$ *for the first dimension and* $0..\lfloor\frac{u_i^S}{b_i}\rfloor = \lceil\frac{w_i^S}{b_i}\rceil - 1$ *for* $i = 2, \ldots, n$*.*

Thus, replacing w_i^S *with* $\lceil\frac{w_i^S}{b_i}\rceil$*,* $i = 2, \ldots, n$ *in formula (5.1) and taking into account formula (C.2), we get:*

$$\wp_{\text{BLOCK-CYCLIC-NONOVERLAP}} = \sum_{i=2}^{n}\left[(\lceil\frac{w_i^S}{b_i}\rceil - 1)\%m_ip_i + (\lceil\frac{w_i^S}{b_im_i}\rceil - 1)\%p_i\right] + w_1^S\prod_{i=2}^{n}\lceil\frac{w_i^S}{b_im_ip_i}\rceil$$

In addition, as a TILE consists of $\prod_{i=2}^{n} b_i$ *tiles, assuming that the duration of a time step*

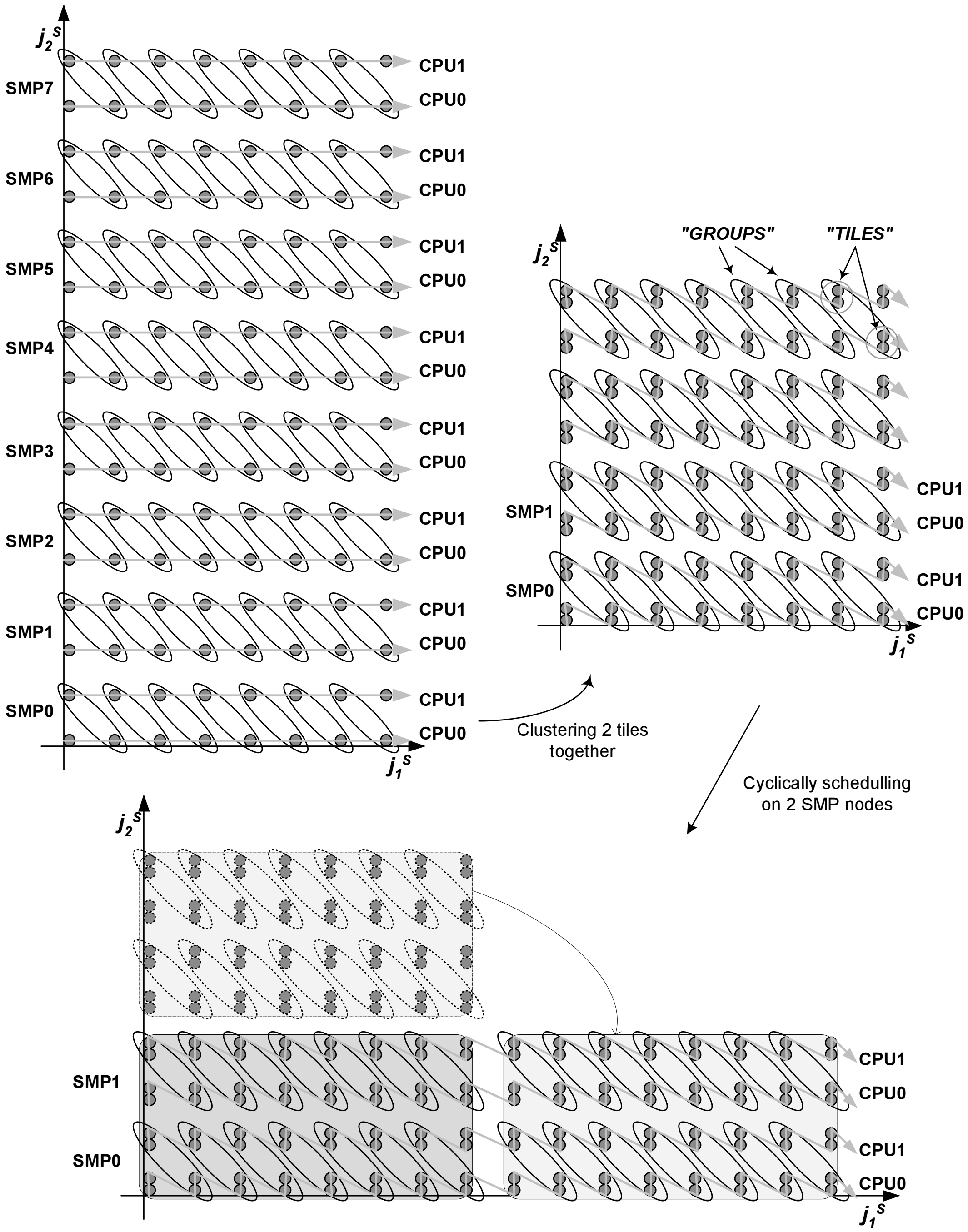


Figure 5.15: Block-cyclic assignment to SMP nodes.
Firstly, tiles are clustered together, so as to form TILES. Then, TILES are cyclically assigned to CPUs. Chunks of TILES are executed one after the other, in lexicographic order.

is mainly determined by the computation time t_{comp}, a STEP will be equivalent to $\prod_{i=2}^{n} b_i$ time steps (excluding the DMA initialization and synchronization time). Thus, the total number *of steps required for the completion of the execution will be*

$$\begin{aligned}\wp_{block-cyclic-overlap} &= \wp_{\text{BLOCK-CYCLIC-OVERLAP}} \prod_{i=2}^{n} b_i = \\ &= \left[\sum_{i=2}^{n}\left[(\lceil\frac{w_i^S}{b_i}\rceil - 1)\% m_i p_i + (\lceil\frac{w_i^S}{b_i m_i}\rceil - 1)\% p_i\right] + w_1^S \prod_{i=2}^{n}\lceil\frac{w_i^S}{b_i m_i p_i}\rceil\right] \prod_{i=2}^{n} b_i\end{aligned}$$

⊣

Theorem 5.8 *The makespan of block-cyclically assigning a rectangular tile space to SMP nodes, following the non-overlapping execution scheme, is:*

$$\wp_{block-cyclic-nonoverlap} = C\left(\sum_{i=2}^{n}\left[(\lceil\frac{w_i^S}{b_i}\rceil - 1)\% m_i p_i\right] + w_1^S \prod_{i=2}^{n}\lceil\frac{w_i^S}{b_i m_i p_i}\rceil\right) \tag{5.12}$$

where $1 \leq C \leq \prod_{i=2}^{n} b_i$.

Proof: *As in the proof of theorem 5.7, in formula (5.3) we replace* w_i^S *with* $\lceil\frac{w_i^S}{b_i}\rceil$, $i = 2, \ldots, n$. *Thus, we get:*

$$\wp_{\text{BLOCK-CYCLIC-NONOVERLAP}} = \sum_{i=2}^{n}\left[(\lceil\frac{w_i^S}{b_i}\rceil - 1)\% m_i p_i\right] + w_1^S \prod_{i=2}^{n}\lceil\frac{w_i^S}{b_i m_i p_i}\rceil$$

In addition, as in the proof of theorem 5.6, a computation subSTEP is equivalent to $\prod_{i=2}^{n} b_i$ *computation substeps, but a communication subSTEP is equivalent to less than* $\prod_{i=2}^{n} b_i$ *communication substeps. In particular, if the communication load is equal along all communication dimensions (as resulted by the method proposed in [Xue97a]), the amount of data to be transferred, as indicated in Figure 5.6, is* $\prod_{i=2}^{n} b_i \sum_{i=2}^{n} \frac{1}{(n-1)b_i} \leq \prod_{i=2}^{n} b_i$ *times the communication load of a tile. Thus, the makespan of the execution will be*

$$\begin{aligned}&\wp_{block-cyclic-nonoverlap} = C\wp_{\text{BLOCK-CYCLIC-NONOVERLAP}} \text{ (where } 1 \leq C \leq \prod_{i=2}^{n} b_i) \Rightarrow \\ &\wp_{block-cyclic-nonoverlap} = C\left(\sum_{i=2}^{n}\left[(\lceil\frac{w_i^S}{b_i}\rceil - 1)\% m_i p_i\right] + w_1^S \prod_{i=2}^{n}\lceil\frac{w_i^S}{b_i m_i p_i}\rceil\right)\end{aligned}$$

⊣

When the tile space is rectangular, the block cyclic assignment schedule can be implemented by the pseudocode of Table 5.3.

5.8 Implementation issues for non-rectangular tile spaces

As deduced from Tables 5.1, 5.3, the implementation of the proposed schedules onto a rectangular tile space space is quite simple and straightforward. However, concerning a non-rectangular tile

Table 5.3: Implementation of the block-cyclic assignment schedule when the tile space is rectangular

Block-Cyclic Assignment - Rectangular Tile Space
FOREACH CPU with coordinates $(cpu\_id_2, \ldots, cpu\_id_n)$ in SMP node with coordinates $(smp\_id_2, \ldots, smp\_id_n)$ DO   FOR $(tt_2 = smp\_id_2 * b_2 * m_2 + cpu\_id_2 * b_2;\ tt_2 < w_2^S;\ tt_2 += b_2 * m_2 * p_2)$ FOR $(tt_3 = smp\_id_3 * b_3 * m_3 + cpu\_id_3 * b_3;\ tt_3 < w_3^S;\ tt_3 += b_3 * m_3 * p_3)$ FOR $(t_1 = 0;\ t_1 < w_1^S;\ t_1 + +)\{$ Execute pre-computation part of Communication   FOR $(t_2 = tt_2;\ t_2 < min(w_2^S, tt_2 + b_2);\ t_2 + +)$ FOR $(t_3 = tt_3;\ t_3 < min(w_3^S, tt_3 + b_3);\ t_3 + +)\{$ Execute Computation of tile (t_1, t_2, t_3) $\}$ Execute post-computation part of Communication $\}$

space, an eventual implementation may be inefficient or crush, if some details are not taken into account.

5.8.1 Assigning as many neighboring tiles as possible to the same SMP node

According to the pseudocode of Table 5.1 for the cyclic assignment schedule, or of Table 5.3 for the block-cyclic one, we may assume that, when a non rectangular tile space is involved, formulas

$$t_2 = l_2^S + smp\_id_2 m_2 + cpu\_id_2 \text{ and } t_3 = l_3^S + smp\_id_3 m_3 + cpu\_id_3$$

or

$$tt_2 = l_2^S + smp\_id_2 b_2 m_2 + cpu\_id_2 b_2 \text{ and } tt_3 = l_3^S + smp\_id_3 b_2 m_3 + cpu\_id_3 b_3$$

respectively, should be employed for the calculation of the lower loop bounds. However, this allocation scheme would result to non-rectangular parts of the tile space being assigned to each SMP node. It would increase the communication load of the final parallel execution, as depicted in Figure 5.16(a).

In order to evict such an inefficient utilization of the bandwidth, we propose the use of function

$$adjust\_mod(l, \alpha, \beta, b) = \begin{cases} \lfloor \frac{l}{\alpha} \rfloor \alpha + \beta \text{ if } \lfloor \frac{l}{\alpha} \rfloor \alpha + \beta + b - 1 \geq l \\ \lceil \frac{l}{\alpha} \rceil \alpha + \beta \text{ else} \end{cases} \tag{5.13}$$

which results to the allocation scheme of Figure 5.16(b), if we replace the lower bounds of the respective loop indices by:

$$t_2 = adjust\_mod(l_2^S, m_2 p_2, smp\_id_2 m_2 + cpu\_id_2, 1)$$
$$t_3 = adjust\_mod(l_3^S, m_3 p_3, smp\_id_3 m_3 + cpu\_id_3, 1)$$

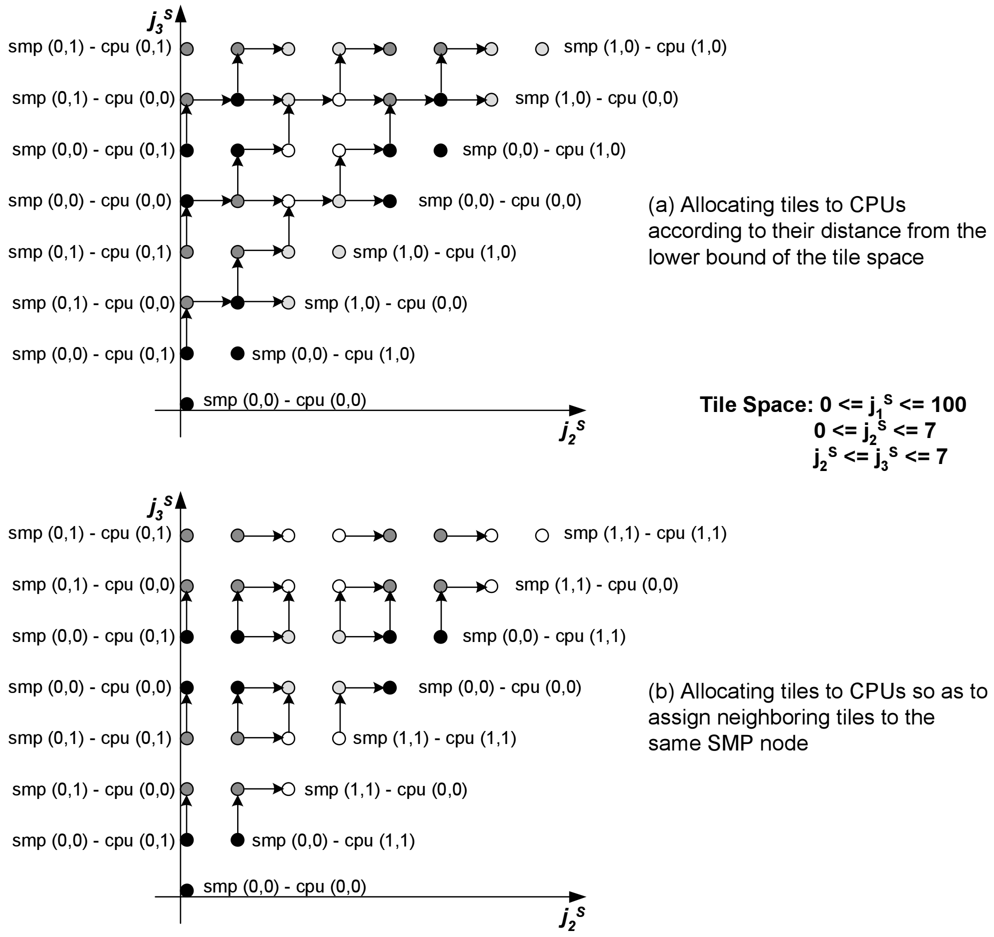


Figure 5.16: Allocating a non-rectangular tile space to processors.
In this figure we have represented the projection of the tile space onto axis plane $j_2^S - j_3^S$. We indicate which processors undertake the boundary tiles, if we have a cluster of 2×2 SMP nodes, containing 2×2 processors each. Tiles, which are assigned to the same SMP node have been depicted using the same grey tone. We have also indicated the subsequent communication among tiles assigned to different SMP nodes, using black arrows. In subfigure (a) more data transfers are implied. Some neighboring tiles, which should exchange data are unnecessarily assigned to different SMP nodes.

or

$$tt_2 = adjust\_mod(l_2^S, b_2m_2p_2, smp\_id_2b_2m_2 + cpu\_id_2b_2, b_2)$$
$$tt_3 = adjust\_mod(l_3^S, b_3m_3p_3, smp\_id_3b_3m_3 + cpu\_id_3b_3, b_3)$$

It can be incorporated in the pseudocode as indicated in Tables 5.4 and 5.7.

5.8.2 Evicting deadlocks

In this section, we shall analyze the problem of deadlocks in case the Myrinet platform is used for the implementation, as in §5.6.1. Similar considerations should be taken when parallelizing in most platforms. Some of them may not imply the use of tokens, however, they will not be able to support an unlimited number of messages to be pending among processors.

Table 5.4: Implementation of the cyclic assignment schedule when the tile space is not rectangular

Cyclic Assignment - Non Rectangular Tile Space

FOREACH CPU with coordinates $(cpu\_id_2, \ldots, cpu\_id_n)$
        in SMP node with coordinates $(smp\_id_2, \ldots, smp\_id_n)$ DO
  FOR $(t_2 = adjust\_mod(l_2^S, m_2 * p_2, smp\_id_2 * m_2 + cpu\_id_2, 1);\ t_2 \leq u_2^S;\ t_2 += m_2 * p_2)$
    FOR $(t_3 = adjust\_mod(l_3^S, m_3 * p_3, smp\_id_3 * m_3 + cpu\_id_3, 1);\ t_3 \leq u_3^S;\ t_3 += m_3 * p_3)$
      FOR $(t_1 = l_1^S;\ t_1 \leq u_1^S;\ t_1{+}{+})\{$
        Execute pre-computation part of Communication
        Execute Computation of tile (t_1, t_2, t_3)
        Execute post-computation part of Communication
      $\}$

where we have assumed that loop bounds l_2^S, u_2^S, l_3^S, u_3^S, l_1^S, u_1^S, have been recalculated, using Fourier Motzkin Elimination method [BW95], [Ban93], so as to be expressed in the order t_2, t_3, t_1

Table 5.5: Implementation of the cluster assignment schedule when the tile space is not rectangular

Cluster Assignment - Non Rectangular Tile Space

FOREACH CPU with coordinates $(cpu\_id_2, \ldots, cpu\_id_n)$
        in SMP node with coordinates $(smp\_id_2, \ldots, smp\_id_n)$ DO
  FOR $(t_1 = l_1^S;\ t_1 \leq u_1^S;\ t_1{+}{+})\{$
    Execute pre-computation part of Communication
    FOR $(t_2 = max(l_2^S, min\_l_2^S + (smp\_id_2 * m_2 + cpu\_id_2) * \lceil \frac{max\_u_2^S - min\_l_2^S + 1}{m_2 * p_2} \rceil);$
            $t_2 \leq min(u_2^S, min\_l_2^S + (smp\_id_2 * m_2 + cpu\_id_2 + 1) * \lceil \frac{max\_u_2^S - min\_l_2^S + 1}{m_2 * p_2} \rceil - 1);\ t_2{+}{+})$
      FOR $(t_3 = max(l_3^S, min\_l_3^S + (smp\_id_3 * m_3 + cpu\_id_3) * \lceil \frac{max\_u_3^S - min\_l_3^S + 1}{m_3 * p_3} \rceil;$
            $t_3 \leq min(u_3^S, min\_l_3^S + (smp\_id_3 * m_3 + cpu\_id_3 + 1) * \lceil \frac{max\_u_3^S - min\_l_3^S + 1}{m_3 * p_3} \rceil - 1);\ t_3{+}{+})\{$
        Execute Computation of tile (t_1, t_2, t_3)
      $\}$
    Execute post-computation part of Communication
  $\}$

where $min\_l_2 = min(l_2(t_1))$ and $max\_u_2 = max(u_2(t_1))$. Similarly, $min\_l_3 = min(l_3(t_1, t_2))$ and $max\_u_3 = max(u_3(t_1, t_2))$. These values can be calculated by applying Fourier Motzkin Elimination method [BW95], [Ban93] to the tile space boundaries, considering that outermost loop indices are t_2, t_3, respectively.

Table 5.6: Implementation of the mirror assignment schedule when the tile space is not rectangular

Mirror Assignment - Non Rectangular Tile Space

FOREACH CPU with coordinates $(cpu\_id_2, \ldots, cpu\_id_n)$
    in SMP node with coordinates $(smp\_id_2, \ldots, smp\_id_n)$ DO
  FOR $(x_2 = 0;\ x_2 \leq \lceil \frac{u_2^S - l_2^S + 1}{m_2 * p_2} \rceil - 1;\ x_2 ++)\{$
    $t_2 = l_2^S + x_2 * m_2 * p_2 + (1 - x_2\%2) * (smp\_id_2 * m_2 + cpu\_id_2) +$
      $+(x_2\%2) * (m_2 * p_2 - 1 - smp\_id_2 * m_2 - cpu\_id_2);$
    IF $(l_2^S \leq t_2 \leq u_2^S)$
      FOR $(x_3 = 0;\ x_3 \leq \lceil \frac{max\_u_3^S - min\_l_3 + 1}{m_3 * p_3} \rceil - 1;\ x_3 ++)\{$
        $t_3 = min\_l_3 + x_3 * m_3 * p_3 + (1 - x_3\%2) * (smp\_id_3 * m_3 + cpu\_id_3) +$
          $+(x_3\%2) * (m_3 * p_3 - 1 - smp\_id_3 * m_3 - cpu\_id_3);$
        IF $(l_3 \leq t_3 \leq u_3^S)$
          FOR $(t_1 = l_1^S;\ t_1 \leq u_1^S;\ t_1 ++)\{$
            Execute pre-computation part of Communication
            Execute Computation of tile (t_1, t_2, t_3)
            Execute post-computation part of Communication
          }
      }
  }

As in Table 5.4, we have assumed that loop bounds l_2^S, u_2^S, l_3^S, u_3^S, l_1^S, u_1^S, have been recalculated, so as to be expressed in the order t_2, t_3, t_1.

Table 5.7: Implementation of the block-cyclic assignment schedule when the tile space is not rectangular

Block-Cyclic Assignment - Non Rectangular Tile Space

FOREACH CPU with coordinates $(cpu\_id_2, \ldots, cpu\_id_n)$
    in SMP node with coordinates $(smp\_id_2, \ldots, smp\_id_n)$ DO
  FOR $(tt_2 = adjust\_mod(l_2^S, b_2 * m_2 * p_2, smp\_id_2 * b_2 * m_2 + cpu\_id_2 * b_2, b_2);$
      $tt_2 \leq u_2^S;\ tt_2 += b_2 * m_2 * p_2)$
    FOR $(tt_3 = adjust\_mod(ll_3^S, b_3 * m_3 * p_3, smp\_id_3 * b_3 * m_3 + cpu\_id_3 * b_3, b_3);$
        $tt_3 \leq uu_3^S;\ tt_3 += b_3 * m_3 * p_3)$
      FOR $(t_1 = ll_1^S;\ t_1 \leq uu_1^S;\ t_1 ++)\{$
        Execute pre-computation part of Communication
        FOR $(t_2 = max(l_2^S, tt_2);\ t_2 \leq min(u_2^S, tt_2 + b_2 - 1);\ t_2 ++)$
          FOR $(t_3 = max(l_3^S, tt_3);\ t_3 \leq min(u_3^S, tt_3 + b_3 - 1);\ t_3 ++)\{$
            if $l_1^S(t_2, t_3) \leq t_1 \leq u_1^S(t_2, t_3)$ Execute Computation of tile (t_1, t_2, t_3)
          }
        Execute post-computation part of Communication
      }

As in Table 5.4, we have assumed that loop bounds l_2^S, u_2^S, l_3^S, u_3^S, l_1^S, u_1^S, have been recalculated, so as to be expressed in the order t_2, t_3, t_1. In addition, bound $ll_3^S(tt_2)$ is calculated by formula giving $l_3^S(t_2)$, if we replace t_2 with tt_2, if its multiplying factor is positive, or with $tt_2 + b_2 - 1$, if its multiplying factor is negative. That is, we replace each at_2 with $max(a, 0)tt_2 + min(a, 0)(tt_2 + b_2 - 1)$. Similarly, $uu_3^S(tt_2)$ is calculated by the formula giving $u_3^S(t_2)$, if we replace each at_2 with $min(a, 0)tt_2 + max(a, 0)(tt_2 + b_2 - 1)$. Limits $ll_1^S(tt_2, tt_3)$ and $uu_1^S(tt_2, tt_3)$ are calculated in the same way.

When using Myrinet-GM [Myr02], the receive event queue provides 317 tokens per port, 254 for receive events and 63 for send events. However, when implementing a cyclic assignment schedule (or a block-cyclic one), as in Figure 5.17, it is strongly possible that more than 254 receive events have arrived before the first of them is necessary for the node to go on with computations. In the case of a rectangular tile space, this problem can be easily coped with as follows: Before the computation of a tile each CPU may check for pending events, whether it needs for data in order to go on, or not.

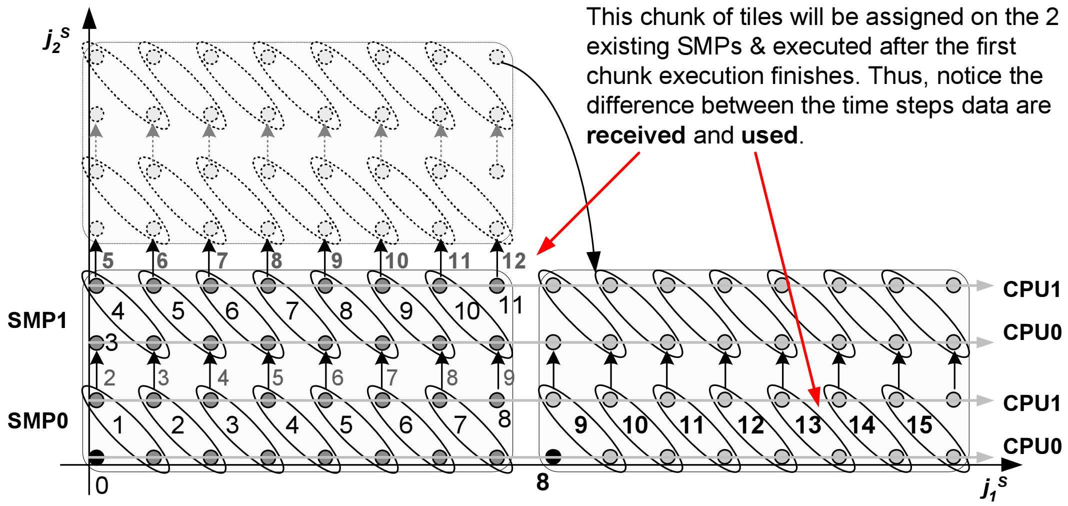


Figure 5.17: Time distance between the arrival of an event and the use of data it carries. Since the mapping direction of the tile space is too short in this example, only 3 events will remain pending until time step 8, when the execution of the second chunk of tiles starts in SMP node 0. The longer dimension j_1^S will be, the more events will be pending.

In the case of a non-rectangular tile space, the implementation is not so simple. As shown in Figure 5.18 and argued in the caption below, deadlocks in a non-rectangular tile space cannot be coped with by simply checking the event queue before the execution of a tile. In Figure 5.18, CPU 0 of node 1 is stalled.

A possible solution of this problem is as follows: When starting the execution of a row of tiles, each thread, which is possible to receive data, should create an assistant thread. It checks for pending events in the receive event queue and if it finds one, the event is processed and a new receive token is made available. If there are no receive events in the queue, the CPU is yielded to the main thread. So, if the assistant thread is useless, as in the case of a rectangular tile space, it will not considerably slow down the execution of the main thread.

5.8.3 Simulation Data

In order to study the behavior of the block-cyclic assignment scheme, we have constructed a simulation program. It really creates so many threads, as the processors of the cluster are

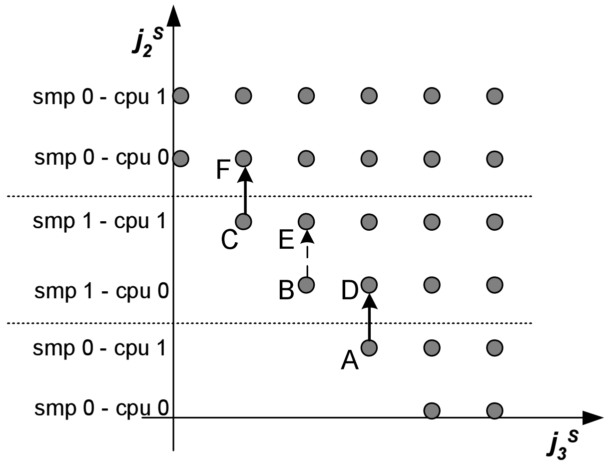


Figure 5.18: Deadlocks in the execution of non-rectangular tile space.
In this figure, the projection of the tile space onto axis plane $j_2^S - j_3^S$ is presented. While CPU 1 of SMP node 1 is computing the row of tiles labelled as C and filling in the receive buffers of node 0, CPU 0 of the node 1 is stalling on a barrier between rows B and E. At the same time, the data arriving from the neighboring node 0, due to the computation of row A, are likely to fill in the receive buffers and use up the receive tokens of node 1. However, if the computation of row A does not finish, the computation of row F will never start, so as to restore the receive tokens needed for row C.

supposed to be. It acts as if traversing the tile space, but instead of executing computations, it adds a time interval to the time previous computations have been computed and necessary data have arrived. Instead of exchanging data, threads exchange the time instances each tile and its subsequent communication are supposed to complete. Thus, we may experiment with all tile spaces and with underlying architectures that we do not have really available. We may set the communication characteristics to resemble any slow or fast network architecture.

Alternative Direction Implicit Integration (ADI)

First, we experimented with the Alternative Direction Implicit Integration (ADI) benchmark. The code segment which implies the main computational load and which deserves parallelization is given by the following nested `for`-loop:

```
for (t=0; t≤T-1; t++)
   for (i=0; i≤I-1; i++)
      for (j=0; j≤J-1; j++){
         X[t,i,j]=X[t-1,i,j]+X[t-1,i,j-1]*A[i,j]/B[t-1,i,j-1]-
                  X[t-1,i-1,j]*A[i,j]/B[t-1,i-1,j];
         B[t,i,j]=B[t-1,i,j]-A[i,j]*A[i,j]/B[t-1,i,j-1]
                  -A[i,j]*A[i,j]/B[t-1,i-1,j];
      }
```

The dependence matrix of this code segment is

$$D = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

One of the optimal tiling matrices, according to communication minimization criteria [Xue97a], can be proven to be

$$P = \begin{bmatrix} 10 & 10 & 10 \\ 0 & 10 & 0 \\ 0 & 0 & 10 \end{bmatrix}$$

After applying this tiling transformation, to the initial code segment with I=J=200 and T=1000, the tiled code segment can be rewritten as follows:

```
for (ii=0; ii≤19; ii++)
   for (jj=0; jj≤19; jj++)
      for (tt=-2-ii-jj; tt≤99-ii-jj; tt++){
         Work with tile (tt, ii, jj)
      }
```

We simulated the execution of this code segment on a cluster with a fixed number of SMP nodes and a fixed number o CPUs inside each node. We tested all possible values of parameters p_i, m_i, b_i, so as to locate those characteristics that give the best performance. In the following diagrams (Figures 5.19-5.22(b)) we have used the ratio $\frac{Speedup}{Number\ of\ Processors\ Used}$ as an index of the efficiency of a schedule. The maximum value of this fraction may theoretically equal to 1. The closer to 1 ratio $\frac{Speedup}{Number\ of\ Processors\ Used}$ is, the more efficient the respective schedule is considered.

In this benchmark the number of tiles of each row `(ii, jj)` is constant (equal to 102). Thus, the computation load of the algorithm is evenly distributed to processors iff the rows of tiles are evenly distributed. As an indicator of load balance along dimension i, we have used function

$$bal_i = \begin{cases} 0 \text{ if } p_i m_i = 1 \\ (w_i - \lfloor \frac{w_i}{p_i m_i b_i} \rfloor p_i m_i b_i) b_i \text{ else} \end{cases}$$

The outcome of this function is equal to 0 iff the rows of tiles are evenly distributed to processors. As a global indicator of load balance, we have used function

$$bal = \sum bal_i$$

As deduced from Figure 5.19, load balance is necessary and sufficient for achieving the optimal performance when we afford just one SMP node. Otherwise, as deduces from Figures 5.20(a),

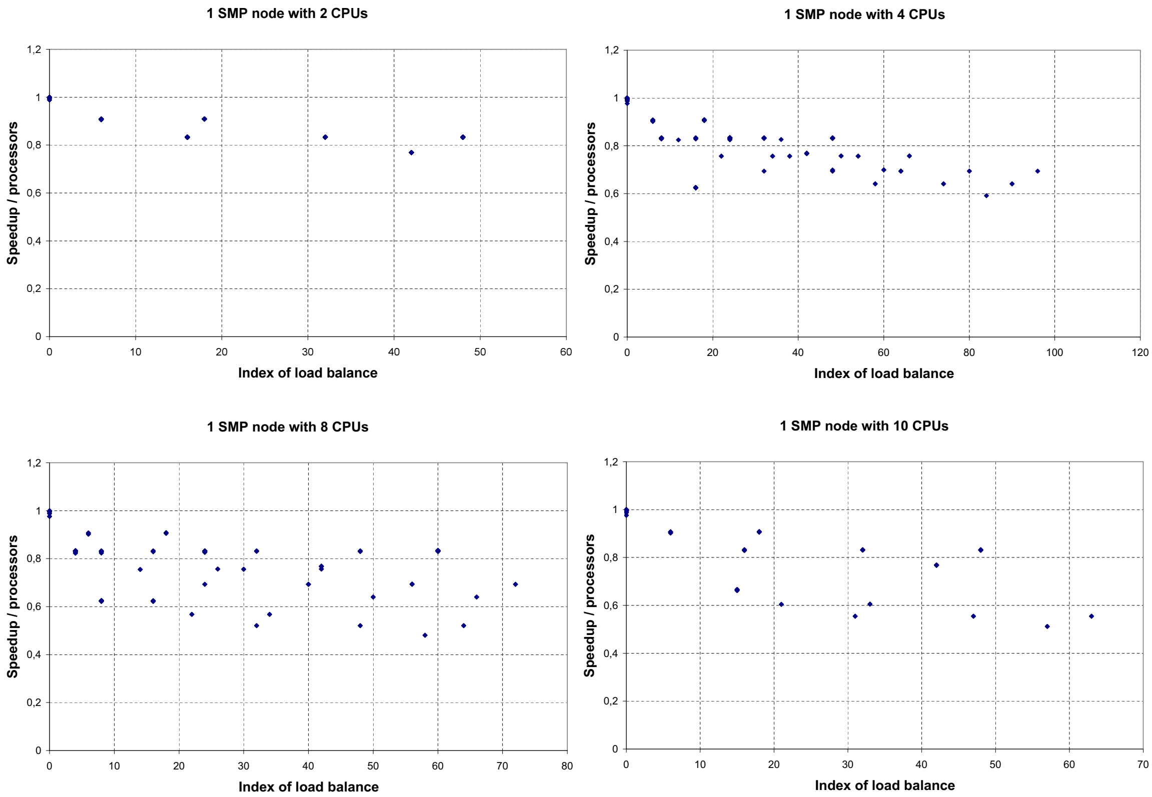


Figure 5.19: Simulation Data: Execution of ADI onto a shared memory multiprocessor. Ratio $\frac{Speedup}{Number\ of\ Processors\ Used}$ is plotted as a function of an index indicating load balance. The optimal performance is achieved when this index indicates a perfect load balance.

5.21(a), 5.22(a), 5.23(a), 5.24(a), 5.25(a), load balance is necessary, but not sufficient for achieving the optimal speedup.

In order to model the data transfer load along dimension i, we have used function

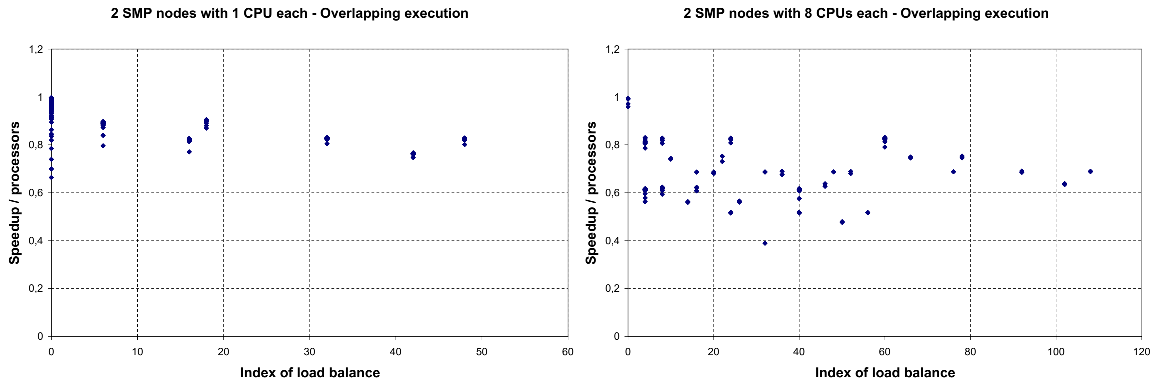
$$comm_i = -1 + \begin{cases} 1 \text{ if } p_i = 1 \\ \lceil \frac{w_i}{p_i m_i b_i} \rceil \text{ else} \end{cases}$$

The total communication load is modelled by function

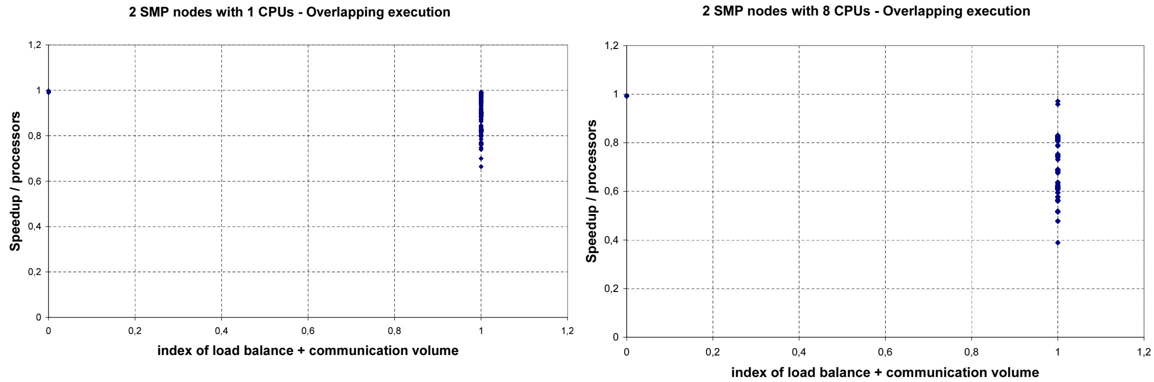
$$comm = \sum (comm_i \prod_{j \neq i} w_j)$$

It can be easily deduced from Figures 5.23(b), 5.24(b), 5.25(b) that, when the non-overlapping execution policy is followed, it is necessary to minimize the communication load, in order to achieve the optimal speedup. When the overlapping execution policy is followed, we did not notice such a relation between communication load and speedup.

In Figures 5.20(b), 5.21(b), 5.22(b), 5.23(c), 5.24(c), 5.25(c), we have used value 0 for the horizontal axis when both load balance and communication indices equal to 0 and value 1



(a) A perfect load balance is necessary, but not sufficient for achieving the optimal speedup.



(b) The optimal speedup is achieved when the computation load is evenly distributed among processors and the communication load is minimized.

Figure 5.20: Simulation Data: Execution of ADI onto a cluster of 2 SMP nodes, following the overlapping execution policy

otherwise. We conclude that almost always the speedup is optimal when both load balance and communication criteria are fulfilled. This holds even for the overlapping execution policy, although we did not find out a direct dependence between communication load and speedup.

In Tables 5.8-5.9, we have indicated the maximum values of ratio $\frac{Speedup}{Number\ of\ Processors\ Used}$ along with the virtual grid configuration and the blocking parameters used. Notice that, for a non negligible value of the time needed for synchronization and overlapped communication, the blocking parameters and grid configuration, that give the optimal performance are almost identical for both the overlapping and the non-overlapping execution policies. In such rectangular tile spaces, we should use the cluster assignment scheme, at least along dimensions with more than one SMP nodes. In comparison to the simulations conducted in §5.6.3, notice that now we have used a non negligible value for the times needed for synchronization and for the initialization of communication, so as to predict the performance of slower than Myrinet interconnection technologies.

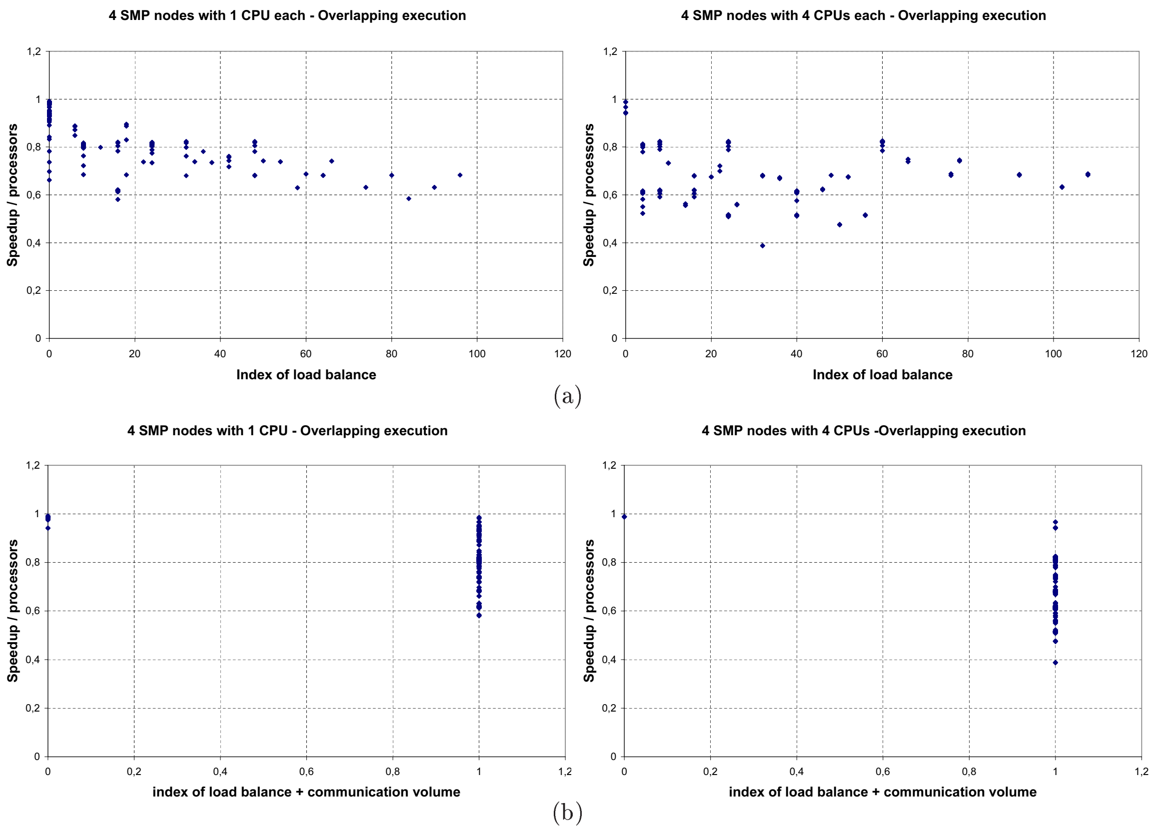


Figure 5.21: Simulation Data: Execution of ADI onto a cluster of 4 SMP nodes, following the overlapping execution policy

Table 5.8: ADI - Simulation Data

The maximum values of ratio $(Speedup)/(Number\ of\ Processors\ Used)$ are achieved when the cluster assignment scheme is followed.

	p_2	p_3	m_2	m_3	b_2	b_3	$Speedup/processors$
1 SMP × 2 CPUs	1	1	1	2	20	10	0.99996
1 SMP × 4 CPUs	1	1	2	2	10	10	0.99987
	1	1	1	4	20	5	0.99985
	1	1	4	1	5	20	0.99985
1 SMP × 8 CPUs	1	1	2	4	10	5	0.99960
	1	1	4	2	5	10	0.99960
	1	1	2	4	5	5	0.99910
	1	1	4	2	5	5	0.99910
1 SMP × 10 CPUs	1	1	1	10	20	2	0.99963
	1	1	10	1	2	20	0.99963
	1	1	2	5	10	4	0.99950
	1	1	5	2	4	10	0.99950

Gauss Successive Over-Relaxation (SOR)

In the sequel, we experimented with the Gauss Successive Over-Relaxation (SOR) benchmark. The code segment which implies the main computational load and which deserves parallelization

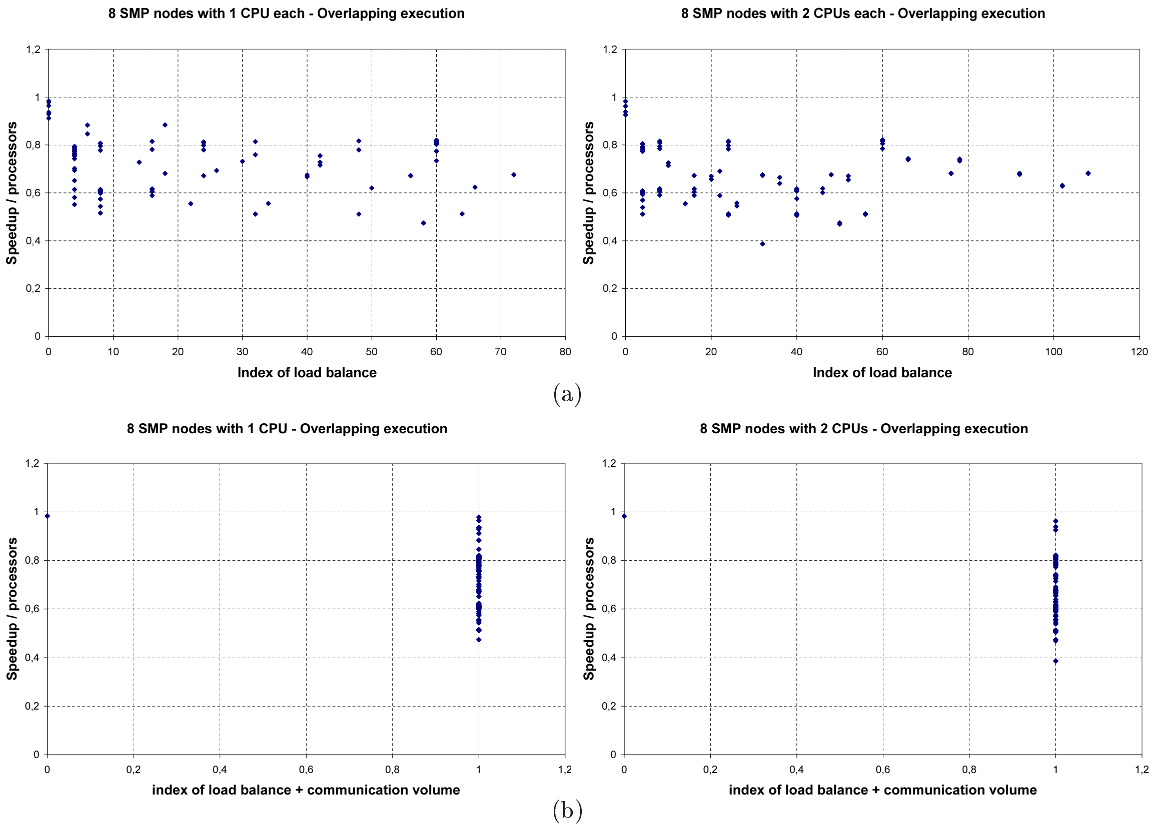


Figure 5.22: Simulation Data: Execution of ADI onto a cluster of 8 SMP nodes, following the overlapping execution policy

is given by the following nested for-loop:

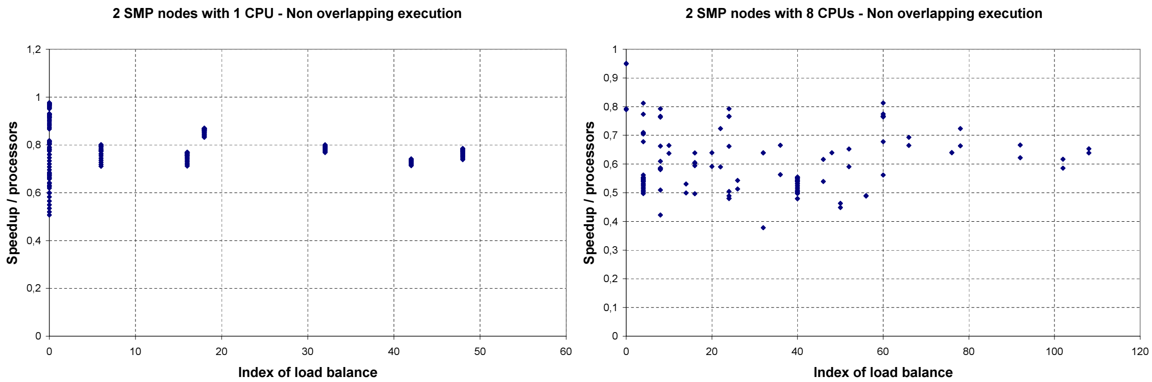
```
for (t=0; t≤T-1; t++)
   for (i=0; i≤I-1; i++)
      for (j=0; j≤J-1; j++){
         A[t,i,j]=(w/4)(A[t,i-1,j]+A[t,i,j-1]+A[t-1,i+1,j]+A[t-1,i,j+1])+
                   (1-w)A[t-1,i,j]
      }
```

The dependence matrix of this code segment is

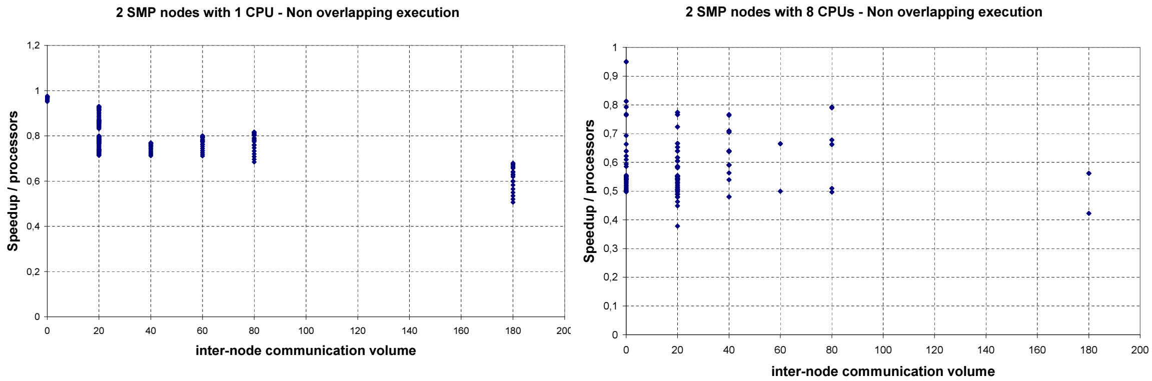
$$D = \begin{bmatrix} 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & -1 \end{bmatrix}$$

One of the optimal tiling matrices, according to communication minimization criteria [Xue97a], can be proven to be

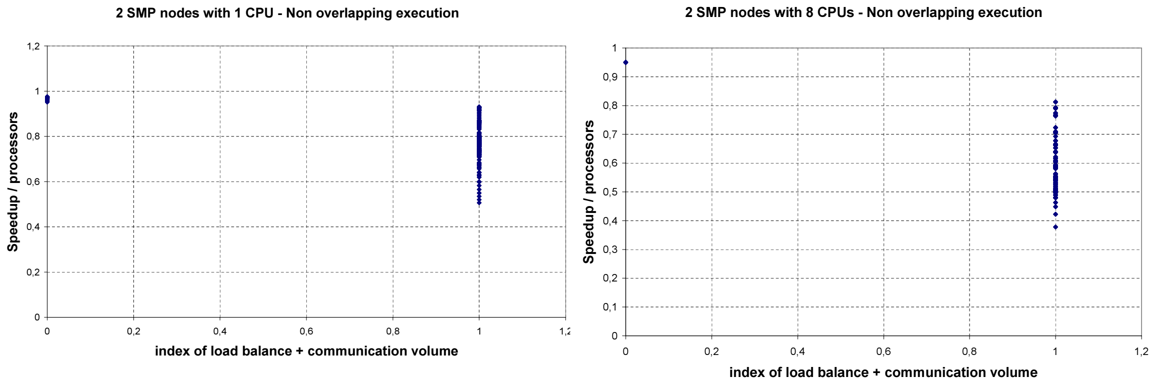
$$P = \begin{bmatrix} 10 & 10 & -10 \\ -10 & 0 & 10 \\ 0 & -10 & 10 \end{bmatrix}$$



(a) A perfect load balance is necessary, but not sufficient for achieving the optimal speedup.



(b) The minimization of the communication load is necessary, but not sufficient for achieving the optimal speedup.



(c) The optimal speedup is achieved when the computation load is evenly distributed among processors and the communication load is minimized.

Figure 5.23: Simulation Data: Execution of ADI onto a cluster of 2 SMP nodes, following the non-overlapping execution policy

After applying this tiling transformation, to the initial code segment with I=J=200 and T=1000, the tiled code segment can be rewritten as follows:

```
for (ii=0; ii≤119; ii++)
   for (jj=ii; jj≤ii+20; jj++)
      for (tt=max(0, jj-20, -ii+jj-1); tt≤min(119, jj, -ii+jj+100); tt++){
         Work with tile (tt, ii, jj)
      }
```

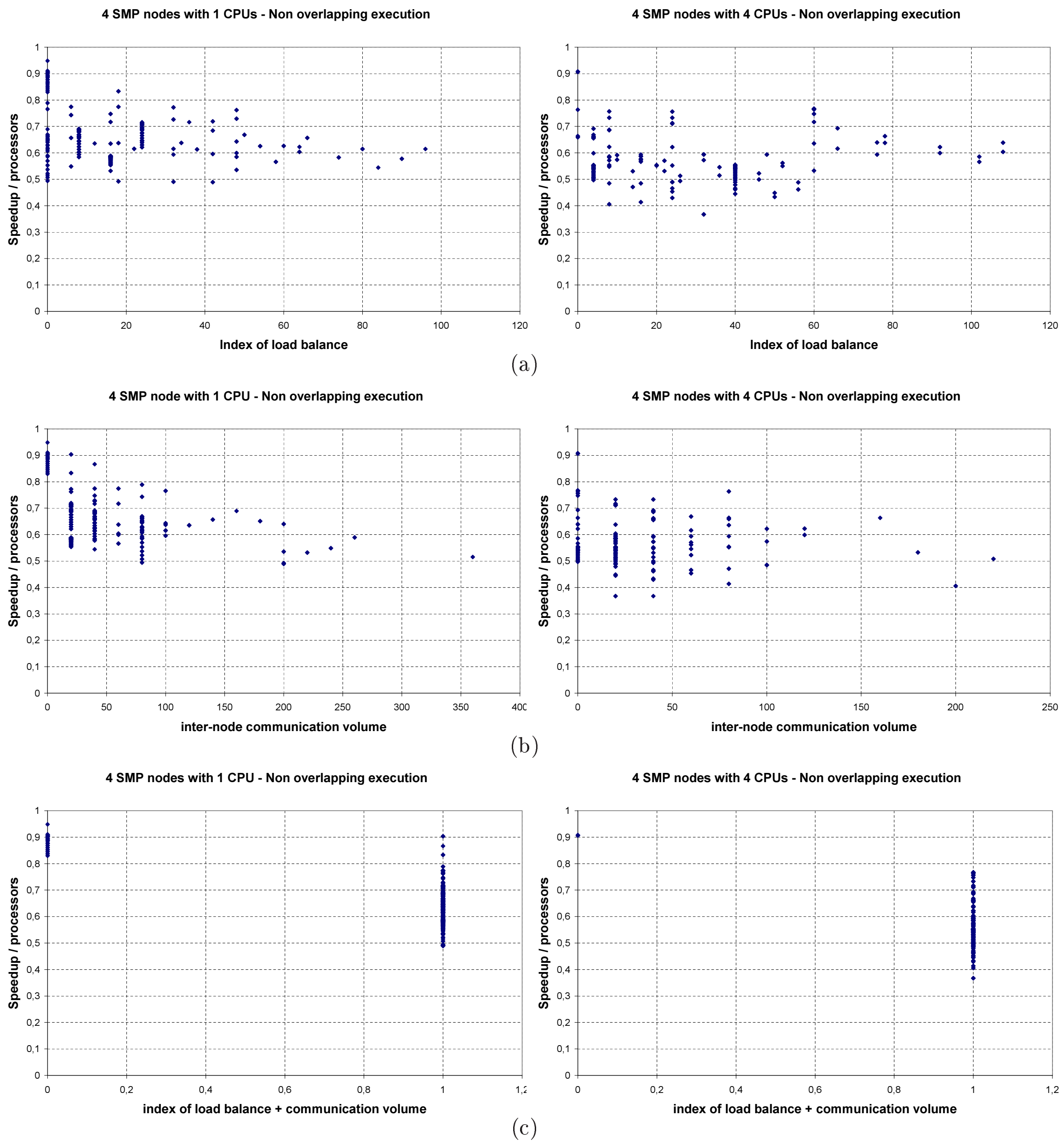
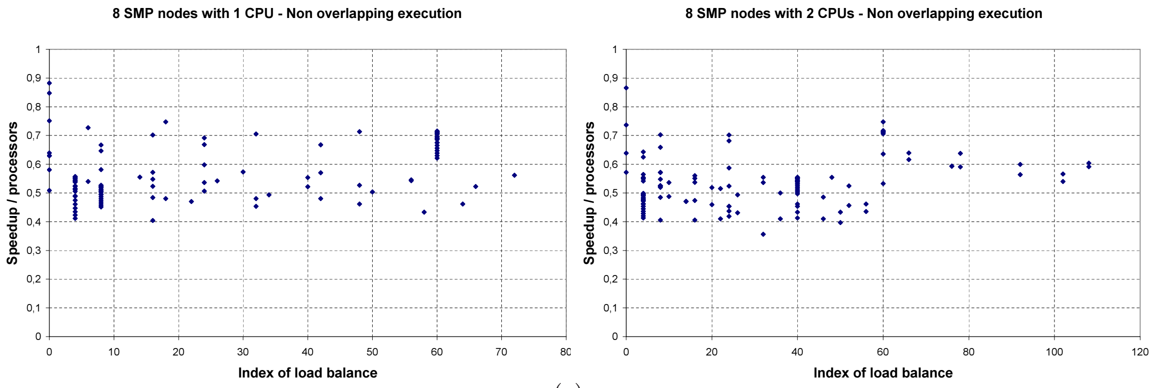


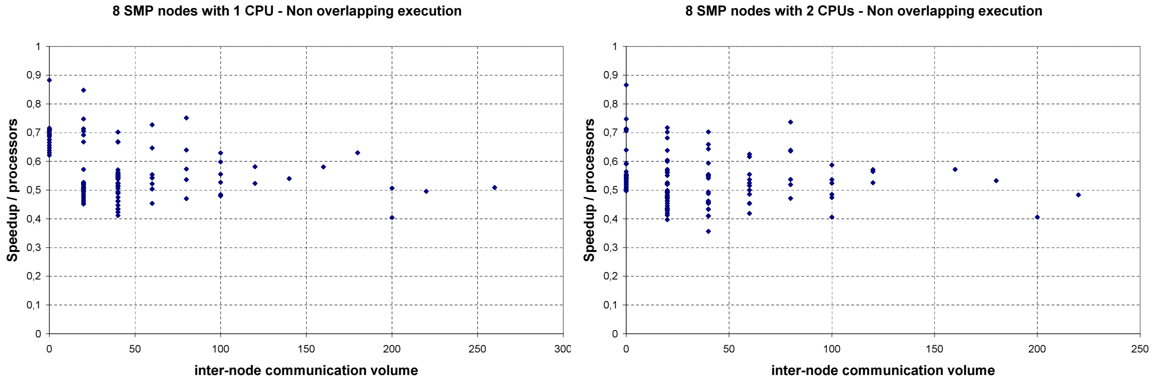
Figure 5.24: Simulation Data: Execution of ADI onto a cluster of 4 SMP nodes, following the non-overlapping execution policy

As in the case of the ADI benchmark, we simulated the execution of this code segment on a cluster with a fixed number of SMP nodes and a fixed number o CPUs inside each node. We tested all possible values of parameters p_i, m_i, b_i, so as to locate the configuration that gives the best performance. In Tables 5.10, 5.11, 5.12 we have used the ratio $\frac{Speedup}{Number\ of\ Processors\ Used}$ as an index of the efficiency of a schedule. The maximum value of this fraction may theoretically equal to 1. The closer to 1 ratio $\frac{Speedup}{Number\ of\ Processors\ Used}$ is, the more efficient the respective schedule is considered.

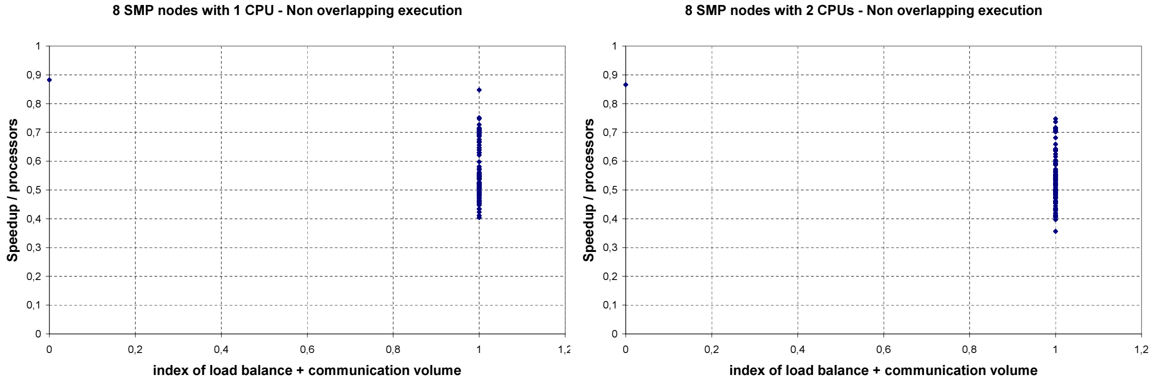
For each cluster size, we have denoted the configuration that gives the best performance. Then, we have indicated the optimal cyclic configuration and the optimal cluster configuration.



(a)



(b)



(c)

Figure 5.25: Simulation Data: Execution of ADI onto a cluster of 8 SMP nodes, following the non-overlapping execution policy

In the last column of Tables 5.10, 5.11, 5.12 we have indicated the percent reduction in efficiency of the cyclic or cluster schedule, in comparison to the optimal block-cyclic schedule.

One can easily deduce that for such a non-rectangular tile space, the cluster assignment schedule is totally out of a question. This is due to the fact that when a processor starts executing the tiles assigned to it, the processors that have previously started executing computations, have almost finished with them. Thus, the execution of the tile space is almost not parallelized.

On the other hand, when the overlapping execution policy is followed, the cyclic assignment schedule can achieve an almost optimal performance, as deduced from Table 5.11. When the non-overlapping execution scheme is followed, the cyclic assignment schedule may be up to 26%

Table 5.9: ADI - Simulation Data

The maximum values of ratio $(Speedup)/(Number\ of\ Processors\ Used)$ are achieved when the cluster assignment scheme is followed, at least along dimensions with more than one SMP nodes.

	p_2	p_3	m_2	m_3	b_2	b_3	*Speedup/processors* *Non-overlapping*	*Overlapping*
2 SMPs × 1 CPU	1	2	1	1	1	10	0.976	0.998
	2	1	1	1	10	1	0.976	0.998
	1	2	1	1	2	10	0.976	0.998
	2	1	1	1	10	2	0.976	0.998
2 SMPs × 2 CPUs	1	2	2	1	2	10	0.975	0.997
	2	1	1	2	10	2	0.975	0.997
	1	2	2	1	5	10	0.975	0.997
	2	1	1	2	10	5	0.975	0.997
2 SMPs × 4 CPUs	1	2	4	1	5	10	0.975	0.997
	2	1	1	4	10	5	0.975	0.997
	1	2	4	1	1	10	0.975	0.996
	2	1	1	4	10	1	0.975	0.996
2 SMPs × 8 CPUs	1	2	4	2	5	5	0.950	0.994
	2	1	2	4	5	5	0.950	0.994
	1	2	4	2	1	5	0.949	0.991
	2	1	2	4	5	1	0.949	0.991
4 SMPs × 1 CPU	2	2	1	1	10	10	0.949	0.991
	1	4	1	1	1	5	0.91	0.99
	4	1	1	1	5	1	0.91	0.99
	1	4	1	1	2	5	0.909	0.99
	4	1	1	1	5	2	0.909	0.99
	1	4	1	1	4	5	0.907	0.989
	4	1	1	1	5	4	0.907	0.989
4 SMPs × 2 CPUs	2	2	1	2	10	5	0.926	0.99
	2	2	2	1	5	10	0.926	0.99
	1	4	2	1	2	5	0.908	0.989
	4	1	1	2	5	2	0.908	0.989
4 SMPs × 4 CPUs	1	4	4	1	1	5	0.908	0.988
	4	1	1	4	5	1	0.908	0.988
	1	4	4	1	5	5	0.906	0.988
	2	2	2	2	5	5	0.906	0.988
	4	1	1	4	5	5	0.906	0.988
8 SMPs × 1 CPU	2	4	1	1	10	5	0.882	0.983
	4	2	1	1	5	10	0.882	0.983
	2	4	1	1	5	5	0.847	0.979
	4	2	1	1	5	5	0.847	0.979
	2	4	1	1	2	5	0.751	0.964
	4	2	1	1	5	2	0.751	0.964
8 SMPs × 2 CPUs	2	4	2	1	5	5	0.866	0.982
	4	2	1	2	5	5	0.866	0.982

Table 5.10: SOR - Simulation Data

	p_2	p_3	m_2	m_3	b_2	b_3	*Speedup/processors*	*Efficiency reduction*
1 SMP × 2 CPUs	1	1	1	2	120	5	0.999421271	
	1	1	2	1	1	1	0.988251853	1.2%
	1	1	2	1	60	140	0.534139023	47%
1 SMP × 4 CPUs	1	1	4	1	1	140	0.997985691	
	1	1	4	1	1	1	0.987554938	1%
	1	1	4	1	30	140	0.309307308	69%
1 SMP × 8 CPUs	1	1	8	1	1	140	0.989166534	
	1	1	8	1	1	1	0.978837276	1%
	1	1	8	1	15	140	0.216077827	78%
1 SMP × 10 CPUs	1	1	10	1	1	10	0.980911549	
	1	1	10	1	1	1	0.971447503	1%
	1	1	10	1	12	140	0.198010851	80%

Table 5.11: SOR - Simulation Data, following the overlapping execution policy

	p_2	p_3	m_2	m_3	b_2	b_3	*Speedup/processors*	*Efficiency reduction*
2 SMPs × 1 CPU	2	1	1	1	5	1	0.987745575	
	2	1	1	1	1	1	0.954850866	3.3%
	2	1	1	1	60	140	0.53174481	46%
2 SMPs × 2 CPUs	2	1	1	2	6	1	0.984724165	
	2	1	2	1	1	1	0.970604098	1.4%
	2	1	2	1	30	140	0.308091238	69%
2 SMPs × 4 CPUs	2	1	2	2	2	1	0.972011902	
	2	1	4	1	1	1	0.962034972	1%
	2	1	4	1	15	140	0.215110584	78%
2 SMPs × 8 CPUs	2	1	2	4	3	1	0.923522467	
	2	1	4	2	1	1	0.910115457	1.5%
	2	1	8	1	8	140	0.166069888	82%
4 SMPs × 1 CPU	4	1	1	1	2	1	0.970575774	
	4	1	1	1	1	1	0.954196445	1.7%
	4	1	1	1	30	140	0.305305101	69%
4 SMPs × 2 CPUs	4	1	1	2	2	1	0.963700266	
	4	1	2	1	1	1	0.961991687	0.18%
	4	1	2	1	15	140	0.213112999	78%
4 SMPs × 4 CPUs	4	1	1	4	3	1	0.918472758	
	4	1	2	2	1	1	0.910052738	0.92%
	4	1	4	1	8	140	0.164702948	82%
8 SMPs × 1 CPU	8	1	1	1	1	1	0.945760927	
	8	1	1	1	1	1	0.945760927	0%
	8	1	1	1	15	140	0.208689671	78%
8 SMPs × 2 CPUs	8	1	1	2	2	1	0.895967945	
	8	1	1	2	1	1	0.895508695	0.05%
	8	1	2	1	8	140	0.161913245	82%

slower than the block-cyclic assignment schedule. This is due to the fact that it imposes a very

Table 5.12: SOR - Simulation Data, following the non-overlapping execution policy

	p_2	p_3	m_2	m_3	b_2	b_3	*Speedup/processors*	*Efficiency reduction*
2 SMPs × 1 CPU	2	1	1	1	8	1	0.933471933	
	1	2	1	1	1	1	0.687822177	26%
	1	2	1	1	120	70	0.516923785	45%
2 SMPs × 2 CPUs	2	1	1	2	8	1	0.931415461	
	2	1	2	1	1	1	0.804743867	14%
	2	1	2	1	30	140	0.297544003	68%
2 SMPs × 4 CPUs	2	1	1	4	9	1	0.915342071	
	2	1	4	1	1	1	0.797715939	13%
	2	1	4	1	15	140	0.206804037	77%
2 SMPs × 8 CPUs	2	1	2	4	4	1	0.872820864	
	2	1	4	2	1	1	0.761663718	13%
	2	1	8	1	8	140	0.159950583	82%
4 SMPs × 1 CPU	4	1	1	1	4	1	0.852477691	
	4	1	1	1	1	1	0.678914342	20%
	1	4	1	1	120	35	0.276955342	68%
4 SMPs × 2 CPUs	4	1	1	2	4	1	0.847714428	
	4	1	2	1	1	1	0.797329214	5.9%
	4	1	2	1	15	140	0.18935053	78%
4 SMPs × 4 CPUs	4	1	1	4	4	1	0.832239744	
	4	1	2	2	1	1	0.761153699	8.5%
	4	1	4	1	8	140	0.146127364	82%
8 SMPs × 1 CPU	4	2	1	1	4	4	0.765480087	
	8	1	1	1	1	1	0.672901118	12%
	8	1	1	1	15	140	0.164661875	78%
8 SMPs × 2 CPUs	8	1	1	2	2	1	0.742509583	
	8	1	2	1	1	1	0.731556309	1.5%
	8	1	2	1	8	140	0.130694603	82%

dense communication pattern. Thus, the block-cyclic assignment scheme achieves the happy medium between communication load and concurrent execution on different processors.

6

Conclusion

In this thesis, we have added some notions to the difficult problem of automatic parallelization of nested `for`-loops.

In [GAK03], [GDAK02a], [GDAK04], a complete framework for automatically producing parallel SPMD code has been presented. However, we assumed that there are always as many processors as needed, or, that processes are scheduled by the operating system on the available processors. However, as explained in §5.1, this scheduling may not be optimal. Chapter 5 of this thesis is now presenting a solution to this problem. In addition, we had not taken into account multi-level parallel architectures. This case is coped with by Chapter 4 and §3.3 of this thesis.

In [Sot04], Sotiropoulos has presented an innovating parallel scheduling, which can exploit advanced communication features of modern clusters, such as Direct Memory Accessing and Zero-Copy protocols [KSG03], [GSK01]. This thesis is now modifying the schedule proposed by Sotiropoulos, in order to exploit the proximity of processors within the same SMP node.

Thus, this thesis can be considered as the last among realized steps for the parallelization of nested `for`-loops:

1. First of all, one should conduct a dependence analysis of the code segment, as described in [Ban88], [Pug92]. We assume that this step gives uniform dependences, as described in §2.3 and in §B.2.

2. Then, we select the optimal tiling, according to cache locality or communication overhead minimization criteria, as described in [KRC99], [LRW91], [WL91a], [PHP03], [MHCF98] and [AKN95], [RR02], [BDRR94], [Xue97a], [Xue00], [RR04].

3. Sequential code is converted to serial tiled code, according to the tiling transformation selected in step 2, as described in [GAK02b], [GAK03] and in §3.2 of this thesis. This conversion is consisted of two substeps:

   (a) Producing the bounds of the tile space from the bounds of the iteration space (§3.2.1) and

   (b) Producing the appropriate boundary expressions for traversing the internal of each tile, as well as determining the incremental steps of each loop index (§3.2.2).

4. A communication policy (overlapping or non-overlapping) may be selected [GSK01], [KSG03], according to the hardware technology that will be used. If the network interconnection supports Direct Memory Access (DMA) protocols, we highly recommend the selection of the overlapping communication policy. If DMA is not supported by hardware, then overlapping communication will not be really implemented. Thus, writing code for overlapping communication over this hardware architecture will only introduce unnecessary delays to the final program.

5. If our cluster is consisted by Symmetric Multiprocessors (SMPs), then the proximity of processors in the same SMP node can be exploited by applying a grouping transformation to the tile space, produced in step 3a, and then scheduling groups instead of tiles, as described in [ASTK02b], [AST$^+$05] and in Chapter 4 of this thesis.

6. If the number of rows of tiles produced by step 3a exceeds the number of CPUs available, then it is advised to apply a static scheduling of tiles or groups, as described in [AKK04] and in Chapter 5 of this thesis. If the tile space (step 3a) is rectangular, then we need not take into account load balancing issues. Thus, we may select between the cyclic assignment schedule (§5.2) and the cluster assignment schedule (§5.4). The cyclic assignment schedule is preferable when the overlapping communication policy has been selected in step 4, while cluster assignment schedule is preferable when the non-overlapping communication policy has been selected. If the tile space is not rectangular, then the block-cyclic assignment schedule constitutes a useful compromise of the advantages and disadvantages of cyclic and cluster assignment schedules.

7. Finally, serial tiled code, produced in step 3, can be converted into parallel code, taking into account the decisions of steps 4, 5, 6, and allocating data to processes, as described in [GDAK02a], [Gou03] and in §3.3 of this thesis.

Although a lot of research has been conducted in this area, we cannot yet automatically produce optimal parallel tiled code for the execution of code segments with nested `for`-loops onto parallel architectures.

- First of all, we have not yet investigated the interaction among the tile selection techniques (step 2) and subsequent steps (4, 5, 6). It is strongly possible that the application of different communication policies or assignment schemes will modify the criteria for the selection of the optimal tiling transformation. Thus, maybe an overall analysis of problems corresponding to steps 2, 4, 5 and 6 would modify the final parallel code produced in step 7.

- In addition, we may incorporate in the previous procedure the data layout and indexing techniques described in [AK04], [AKT05]. In these papers, E. Athanasaki et al. have presented an alternative array data layout, which stores array elements in memory in the order they are fetched in cache by the tiled nested `for`-loop code segment. Then, the combination of parallelization and peak cache performance is expected to further boost the efficiency of the final parallel code. However, incorporating these techniques, will add one more parameter in the tile selection methods applied in step 2.

- Another issue that has not been yet investigated is false sharing inside SMP nodes ([CS99], pages 123-156, [TLH94], [KCRB03]). Is there such a possibility? How can it be evicted? Since tiling has initially been designed for parallelization onto clusters with distributed memory, or for exploiting cache locality on single processing units, these questions have not been yet addressed in the literature.

- Furthermore, one should find out if these techniques can be applied to code segments with imperfectly nested `for`-loops. As described in [AMP00b], [AMP00a], [Xue96], [SL99], [Kul98], [LLL01], every imperfectly nested `for`-loop can be converted into perfectly nested `for`-loop, using `if` statements. However, the techniques described in the above papers are mainly aimed for cache locality optimization, not for parallelism. The computation load of iterations will not be equal. Thus, tiling into equal sized tiles will result into computation load imbalance. On the other hand, the results of this thesis and of referenced related work have been based on the assumptions that tiles are identical.

- Similarly, if the computing system is heterogeneous, tiling into identical tiles will not give equal computation times for all of them. This fact will not be consistent with the underlying assumptions of this thesis and of referenced related work. Then, the techniques presented in this thesis might be combined or enhanced with the ones proposed in [Mor98], [KP96], [CZL95], [CZL97]. However, the methods proposed by above papers cannot replace the schemes proposed in this thesis, since, they concern the parallelization of `doall` loops ([CZL95], [CZL97]), or employ a dynamic scheduling algorithm ([KP96]).

- In order to further reduce the execution time of parallel programs on SMP nodes, we should also query which CPUs of an SMP node should communicate with other SMP nodes. Should each CPU exchange data that concern only its own work? Or should a single processor undertake the communication needed for the whole SMP node? In case the second possibility is taken, how shall we balance the computation+communication load of CPUs?

Appendices

A

Summary of Notations

Symbol	*Explanation*	*Page*
N	set of natural numbers	12
N^*	set of natural numbers, excluding 0, $N^* = N - \{0\}$	12
Z	set of integer numbers	12
Z^*	set of integer numbers, excluding 0, $Z^* = Z - \{0\}$	12
n	Dimensions of the iteration space	12
J^n	Iteration space	13
$\vec{j} = (j_1, \ldots, j_n)$	Iteration coordinates vector	12
J^S	Tile space	30
$\vec{j^S} = (j_1^S, \ldots, j_n^S)$	Tile coordinates vector: $\vec{j^S} = \lfloor H\vec{j} \rfloor$	30
TOS	Tile origin space	30
$\vec{j_0} = (j_{01}, \ldots, j_{0n})$	Tile origin	30
TIS	Tile iteration space	30
$TTIS$	Transformed tile iteration space	59
$\vec{j'} = (j'_1, \ldots, j'_n)$	Instance of the transformed tile iteration space $$\vec{j'} = H'(\vec{j} - \vec{j_0}) \Leftrightarrow \vec{j} = P'(V\vec{j^S} + \vec{j'})$$	63
DS	Data space $DS = \{f_w(\vec{j}) \| \vec{j} \in J^n\}$	75
LDS	Local data space $$LDS = \left\{ \vec{j''} \in Z^n \middle\| \begin{array}{c} 0 \leq j''_k < off_k + m_k v_{kk}/\widetilde{h'}_{kk}, k = 1, \ldots, n, k \neq i \\ \wedge 0 \leq j''_i < off_i + \|t\| v_{ii}/\widetilde{h'}_{ii} \end{array} \right\}$$	78
$\vec{j''} = (j''_1, \ldots, j''_n)$	Instance of the local data space $\vec{j''} = map(\vec{j'}, t)$	78
J^G	Group space	90
$\vec{j^G} = (j_1^G, \ldots, j_n^G)$	Group coordinates vector $\vec{j^G} = \lfloor H^G \vec{j^S} \rfloor$	90

Symbol	***Explanation***	***Page***
H	Tiling matrix	28
g	Smallest natural number such that gH is an integer matrix	31
P	Inverse tiling matrix	28
V	Diagonal matrix with v_{kk} the smallest integer such that $v_{kk}\vec{h_k}$ to be integral	61
H'	Transformation matrix from TIS to TTIS ($H' = VH$)	59
P'	Transformation matrix from TTIS to TIS ($P' = H'^{-1}$)	59
$\widetilde{H'}$	Hermite normal form of matrix H'	61
H^G	Grouping matrix	90
P^G	Inverse grouping matrix	90
D	Dependence matrix	16
D'	Transformed dependence matrix $D' = H'D$	
D^S	Tile dependence matrix	32
Π	Linear time scheduling vector	19
Π^G	Linear time scheduling vector concerning groups	98
$m = m_1 \times \cdots \times m_{i-1} \times$ $\times m_{i+1} \times \cdots \times m_n$	Number of CPUs inside an SMP node	75, 93
$p = p_1 \times \cdots \times p_{i-1} \times$ $\times p_{i+1} \times \cdots \times p_n$	Number of available SMP nodes	127
$sm\vec{p}\_id$	SMP node identification vector	75
$cp\vec{u}\_id$	processor identification vector inside an SMP node	75
$p\vec{i}d$	global processor identification vector $pid_x = pid_x = cpu\_id_x + smp\_id_x m_x \Leftrightarrow$ $cpu\_id_x = pid_x \% m_x,\ smp\_id_x = \lfloor pid_x / m_x \rfloor$	75
$\wp$	Makespan = Number of time steps needed for the completion of the execution	20
i	The longest dimension of the tile space	76, 108
l_k, u_k	Lower and upper bounds of the iteration space $k = 1, \ldots, n$	12
l_k^S, u_k^S	Lower and upper bounds of the tile space $k = 1, \ldots, n$	30
w_k^S	Width of a rectangular tile space along dimension k, $w_k^s = u_k^s - l_k^S + 1,\ k = 1, \ldots, n$	101

B

Algorithmic Model - Summary of assumptions

B.1: We consider an n-dimensional perfectly nested `for`-loop:

```
for (j_1=l_1; j_1 ≤ u_1; j_1++){
    ...
    for (j_n=l_n; j_n ≤ u_n; j_n++){
        Loop Body
    }
    ...
}
```

where l_1 and u_1 are integer parameters, l_k and u_k $(k = 2, \dots, n)$ are functions of the outer loop indices. Specifically, they may have the form:

$$l_k = max(\lceil f_{k1}(j_1, \dots, j_{k-1}) \rceil, \dots, \lceil f_{kr}(j_1, \dots, j_{k-1}) \rceil)$$

and

$$u_k = min(\lfloor g_{k1}(j_1, \dots, j_{k-1}) \rfloor, \dots, \lfloor g_{kr}(j_1, \dots, j_{k-1}) \rfloor),$$

where f_{ki} and g_{ki} are affine functions. (see page 12)

B.2: All dependence vectors are uniform, i.e. independent of the indices of computations. (see page 16)

B.3: There are at least n linearly independent dependence vectors. Thus, the class of dependence matrix D equals to n. (see page 29)

B.4: Anti-dependences and output dependences have been eliminated using more variables [CDRV98]. (see page 16)

B.5: All dependence vectors are smaller than the tile size, thus they are entirely contained in each tile's area. This means that the tile dependence matrix D^S contains only 0's and 1's. (see page 33)

B.6: The processing architecture consists of an homogeneous cluster of single CPU or SMP nodes. (see page 75)

C

Simple Mathematical Formulas

Lemma C.1 *If all n points $\vec{y_i}$, $i = 1, \ldots, n$ belong to a convex space J^n, then every point*

$$\vec{y} = a_1\vec{y_1} + \cdots + a_n\vec{y_n} \tag{C.1}$$

where $a_i \in [0, 1]$ and $a_1 + \cdots + a_n = 1$, belongs to J^n.

Geometrically, this statement can be expressed as follows: If all points $\vec{y_i}$, $i = 1, \ldots, n$ belong to a convex space J^n, then all points located among them belong to J^n.

Proof: *If all points $\vec{y_i}$, $i = 1, \ldots, n$ belong to J^n, then it holds $B\vec{y_i} \leq \vec{b}$ for all $i = 1, \ldots, n$. Consequently, $B\vec{y} = \sum_{i=1}^{n} a_i B\vec{y_i} \leq \sum_{i=1}^{n} a_i\vec{b} = \vec{b}$. Thus, point $\vec{y}$ also belongs to J^n.* $\dashv$

Lemma C.2 *Function*

$$f(x_1, \ldots, x_n) = x_1 + \cdots + x_n,$$

where $x_1 \times \cdots \times x_n = c$ and $x_1, \ldots, x_n > 0$, is minimized when

$$x_1 = \cdots = x_n = c^{\frac{1}{n}}$$

Proof: *Function*

$$f(x_1, \ldots, x_n) = x_1 + \cdots + x_n,$$

where $x_1 \times \cdots \times x_n = c \Rightarrow x_n = \frac{c}{x_1 \times \cdots \times x_{n-1}}$, can be rewritten as follows:

$$f(x_1, \ldots, x_{n-1}) = x_1 + \cdots + x_{n-1} + \frac{c}{x_1 \times \cdots \times x_{n-1}}.$$

Therefore $\frac{\partial f}{\partial x_{n-1}} = 1 - \frac{c}{x_1 \times \cdots \times x_{n-2}} x_{n-1}^{-2}$ and $\frac{\partial^2 f}{\partial x_{n-1}^2} > 0$, $\forall x_{n-1}$.

Thus, function $f(x_1,\ldots,x_{n-1})$ is minimized in respect to the value of x_{n-1} when $\frac{\partial f}{\partial x_{n-1}} = 0 \Rightarrow x_{n-1} = \left(\frac{c}{x_1\times\cdots\times x_{n-2}}\right)^{\frac{1}{2}}$. *For this value of x_{n-1} we can write:*

$$f(x_1,\ldots,x_{n-2}) = x_1+\cdots+x_{n-2}+2\left(\frac{c}{x_1\times\cdots\times x_{n-2}}\right)^{\frac{1}{2}}.$$

After we have eliminated variables $x_{n-i+1},\ldots,x_n$ this way, we conclude that function f can be expressed as

$$f(x_1,\ldots,x_{n-i}) = x_1+\cdots+x_{n-i}+i\left(\frac{c}{x_1\times\cdots\times x_{n-i}}\right)^{\frac{1}{i}}.$$

Therefore $\frac{\partial f}{\partial x_{n-i}} = 1-\left(\frac{c}{x_1\times\cdots\times x_{n-i-1}}\right)^{\frac{1}{i}} x_{n-i}^{-\frac{i+1}{i}}$ *and* $\frac{\partial^2 f}{\partial x_{n-i}^2} > 0,\ \forall x_{n-i}$. *Thus, function $f(x_1,\ldots,x_{n-i})$ is minimized in respect to the value of x_{n-i} when* $\frac{\partial f}{\partial x_{n-i}} = 0 \Rightarrow x_{n-i} = \left(\frac{c}{x_1\times\cdots\times x_{n-i-1}}\right)^{\frac{1}{i+1}}$. *For this value of x_{n-i} we can write that*

$$f(x_1,\ldots,x_{n-i-1}) = x_1+\cdots+x_{n-i-1}+(i+1)\left(\frac{c}{x_1\times\cdots\times x_{n-i-1}}\right)^{\frac{1}{i+1}}.$$

If we continue the elimination of the variables in this way, we conclude that the minimization of f is achieved when $x_1 = c^{\frac{1}{n}}$. After a backwards substitution of the variables in the expressions $x_{n-i} = \left(\frac{c}{x_1\times\cdots\times x_{n-i-1}}\right)^{\frac{1}{i+1}}$ *we conclude that the minimum value of*

$$f(x_1,\ldots,x_n) = x_1+\cdots+x_n$$

is achieved when $x_1=\cdots=x_n=c^{\frac{1}{n}}$. ⊣

Lemma C.3 *Function*

$$f(x_1,\ldots,x_n) = \frac{a_1}{x_1}+\cdots+\frac{a_n}{x_n},$$

where $x_1\times\cdots\times x_n = c$, $a_1,\ldots,a_n$ are positive constants and $x_1,\ldots,x_n$ are positive, is minimized when

$$x_i = a_i\left(\frac{c}{a_1\times\cdots\times a_n}\right)^{\frac{1}{n}}, i=1,\ldots,n$$

Proof: *It holds that* $\frac{a_1}{x_1}\times\cdots\times\frac{a_n}{x_n} = \frac{a_1\times\cdots\times a_n}{c}$*=constant. Thus, according to Lemma C.2, function $f(x_1,\ldots,x_n)$ is minimized when* $\frac{a_1}{x_1}=\cdots=\frac{a_n}{x_n}=\left(\frac{a_1\times\cdots\times a_n}{c}\right)^{\frac{1}{n}} \Rightarrow x_i = a_i\left(\frac{c}{a_1\times\cdots\times a_n}\right)^{\frac{1}{n}}$, $i=1,\ldots,n$. ⊣

Lemma C.4 *If $a\in Z$ and $b,c\in N^*$, it holds that*

$$\left\lceil\frac{\lceil\frac{a}{b}\rceil}{c}\right\rceil = \left\lceil\frac{a}{bc}\right\rceil \tag{C.2}$$

and

$$\lfloor\frac{\lfloor\frac{a}{b}\rfloor}{c}\rfloor = \lfloor\frac{a}{bc}\rfloor \tag{C.3}$$

Proof: *There is a pair of $x \in Z$, $y \in N$ such that $a = bcx - y$ and $0 \leq y \leq bc - 1$. Thus, it holds that*

$$\lceil\frac{a}{bc}\rceil = \lceil\frac{bcx - y}{bc}\rceil = x$$

In addition, there is a pair of $y_1, y_2 \in N$ such that $y = by_1 + y_2$ and $0 \leq y_1 \leq c - 1$, $0 \leq y_2 \leq b - 1$. Thus, it holds that

$$\lceil\frac{\lceil\frac{a}{b}\rceil}{c}\rceil = \lceil\frac{\lceil\frac{bcx - by_1 - y_2}{b}\rceil}{c}\rceil = \lceil\frac{cx - y_1}{c}\rceil = x$$

Thus, formula (C.2) is valid.

Similarly, there is a pair of $w \in Z$, $z \in N$ such that $a = bcw + z$ and $0 \leq z \leq bc - 1$. Thus, it holds that

$$\lfloor\frac{a}{bc}\rfloor = \lfloor\frac{bcw + z}{bc}\rfloor = w$$

In addition, there is a pair of $z_1, z_2 \in N$ such that $z = bz_1 + z_2$ and $0 \leq z_1 \leq c - 1$, $0 \leq z_2 \leq b - 1$. Thus, it holds that

$$\lfloor\frac{\lfloor\frac{a}{b}\rfloor}{c}\rfloor = \lfloor\frac{\lfloor\frac{bcw + bz_1 + z_2}{b}\rfloor}{c}\rfloor = \lfloor\frac{cw + z_1}{c}\rfloor = w$$

Thus, formula (C.3) is valid. $\dashv$

Lemma C.5 *If $a \in Z$ and $b \in N^*$, it holds that*

$$\lfloor\frac{a - 1}{b}\rfloor = \lceil\frac{a}{b}\rceil - 1 \tag{C.4}$$

Proof: *There is a pair of $x \in Z$, $y \in N$ such that $a = bx - y$ and $0 \leq y \leq b - 1$. Thus, it holds that*

$$\lceil\frac{a}{b}\rceil - 1 = \lceil\frac{bx - y}{b}\rceil - 1 = x - 1$$

In addition, since $0 \leq b - y - 1 \leq b - 1$, it holds that

$$\lfloor\frac{a - 1}{b}\rfloor = \lfloor\frac{bx - y - 1}{b}\rfloor = \lfloor\frac{b(x - 1) + (b - y - 1))}{b}\rfloor = x - 1$$

$\dashv$

Lemma C.6 *If $a \in Z$ and $b \in N^*$, it holds that*

$$-\lfloor\frac{a}{b}\rfloor = \lceil-\frac{a}{b}\rceil \tag{C.5}$$

Proof: *There is a pair of $x \in Z$, $y \in N$ such that $a = bx + y$ and $0 \leq y \leq b - 1$. Thus, it holds that*

$$-\lfloor \frac{a}{b} \rfloor = -\lfloor \frac{bx+y}{b} \rfloor = -x$$

In addition, it holds that

$$\lceil -\frac{a}{b} \rceil = \lceil \frac{-bx-y}{b} \rceil = -x$$

⊣

Lemma C.7 *If $a, b \in N^*$, it holds that*

$$\lceil \frac{a}{\lceil \frac{a}{b} \rceil} \rceil \leq b \tag{C.6}$$

Proof: *There is a pair of $x, y \in N$ such that $a = bx - y$ and $0 \leq y \leq b - 1$. Thus, it holds that*

$$\lceil \frac{a}{\lceil \frac{a}{b} \rceil} \rceil = \lceil \frac{a}{x} \rceil = b + \lceil \frac{-y}{x} \rceil \overset{(C.5)}{=} b - \lfloor \frac{y}{x} \rfloor \leq b$$

⊣

Lemma C.8

$$a_1 + a_1 \sum_{i=2}^{n} \left[(a_i - 1) \prod_{k=i+1}^{n} a_k \right] = \prod_{i=1}^{n} a_i \tag{C.7}$$

Proof:

$$\begin{aligned}
&a_1 + a_1 \left[(a_2 - 1)a_3 \dots a_n + (a_3 - 1)a_4 \dots a_n + \dots + (a_{n-2} - 1)a_{n-1}a_n + (a_{n-1} - 1)a_n + a_n - 1 \right] = \\
&= a_1 + a_1 \left[(a_2 - 1)a_3 \dots a_n + (a_3 - 1)a_4 \dots a_n + \dots + (a_{n-2} - 1)a_{n-1}a_n + a_{n-1}a_n - 1 \right] = \\
&= a_1 + a_1 \left[(a_2 - 1)a_3 \dots a_n + (a_3 - 1)a_4 \dots a_n + \dots + a_{n-2}a_{n-1}a_n - 1 \right] = \\
&= \dots = \\
&= a_1 + a_1 \left[a_2 a_3 \dots a_n - 1 \right] = a_1 a_2 \dots a_n
\end{aligned}$$

⊣

Bibliography

[ABR96] R. Andonov, H. Bourzoufi, and S. Rajopadhye. Two-Dimensional Orthogonal Tiling: from Theory to Practice. In *Proceedings of the 1996 International Conference on High-Performance Computing (HiPC'96)*, pages 225–231, Trivandrum, India, Dec. 1996.

[ABRY03] R. Andonov, S. Balev, S. Rajopadhye, and N. Yanev. Optimal Semi-Oblique Tiling. *IEEE Transactions on Parallel and Distributed Systems*, 14(9):944–960, Sep. 2003.

[ACN$^+$00] R. Andonov, P. Calland, S. Niar, S. Rajopadhye, and N. Yanev. First Steps Towards Optimal Oblique Tile Sizing. In *8th International Workshop on Compilers for Parallel Computers*, pages 351–366, Aussois, Jan. 2000.

[AI91] C. Ancourt and F. Irigoin. Scanning Polyhedra with DO Loops. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP)*, pages 39–50, Williamsburg, VA, April 1991.

[AK04] E. Athanasaki and N. Koziris. Fast Indexing for Blocked Array Layouts to Improve Multi-Level Cache Locality. In *Proceedings of the 8-th Workshop on Interaction between Compilers and Computer Architectures (INTERACT'04)*, pages 109–119, Madrid, Spain, Feb. 2004. Held in conjunction with HPCA-10.

[AKK03] M. Athanasaki, E. Koukis, and N. Koziris. Efficient Scheduling of Tiled Iteration Spaces onto a Fixed Size Parallel Architecture. In *Proceedings of the 9th Panhellenic Conference in Informatics*, pages 178–192, Thessaloniki, Greece, Nov. 2003.

[AKK04] M. Athanasaki, E. Koukis, and N. Koziris. Scheduling of Tiled Nested Loops onto a Cluster with a Fixed Number of SMP Nodes. In *Proceedings of the 12-th Euromicro Conference on Parallel, Distributed and Network based Processing (PDP04)*, pages 424–433, A Coruna, Spain, Feb. 2004. IEEE Computer Society Press.

[AKN95] A. Agarwal, D. Kranz, and V. Natarajan. Automatic Partitioning of Parallel Loops and Data Arrays for Distributed Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 6(9):943–962, 1995.

[AKPT99] T. Andronikos, N. Koziris, G. Papakonstantinou, and P. Tsanakas. Optimal Scheduling for UET/UET-UCT Generalized N-Dimensional Grid Task Graphs. *Journal of Parallel and Distributed Computing*, 57(2):140–165, May 1999.

[AKPT00] T. Andronikos, N. Koziris, G. Papakonstantinou, and P. Tsanakas. Optimal Scheduling for UET-UCT Grids Into Fixed Number of Processors. In *Proceedings of 8th Euromicro Workshop on Parallel and Distributed Processing (PDP2000), IEEE Press*, pages 237–243, Rhodes, Greece, Jan. 2000.

[AKT05] E. Athanasaki, N. Koziris, and P. Tsanakas. A Tile Size Selection Analysis for Blocked Array Layouts. In *Proceedings of the 9-th Workshop on Interaction between Compilers and Computer Architectures (INTERACT'05)*, pages 70–80, San Francisco, CA, Feb. 2005. Held in conjunction with HPCA-11.

[AL93] S. P. Amarasinghe and M. S. Lam. Communication Optimization and Code Generation for Distributed Memory Machines. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'93)*, Albuquerque, New Mexico, USA, June 1993.

[AMC97] V. Adve and J. Mellor-Crummey. Advanced Code Generation for High Performance Fortran. In *Languages, Compilation Techniques and Run Time Systems for Scalable Parallel Systems*, chapter 18, Lecture Notes in Computer Science Series. Springer-Verlag, 1997.

[AMP00a] N. Ahmed, N. Mateev, and K. Pingali. Synthesizing Transformations for Locality Enhancement of Imperfectly-nested Loop Nests. In *Proceedings of the 14th International Conference on Supercomputing (ICS2000)*, pages 141–152, Santa Fe, New Mexico, United States, 2000.

[AMP00b] N. Ahmed, N. Mateev, and K. Pingali. Tiling Imperfectly-nested Loop Nests. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, Dallas, Texas, United States, 2000.

[AST+05] M. Athanasaki, A. Sotiropoulos, G. Tsoukalas, N. Koziris, and P. Tsanakas. Hyperplane Grouping and Pipelined Schedules: How to Execute Tiled Loops Fast on Clusters of SMPs. *The Journal of Supercomputing*, 33(3):197–226, Sep. 2005.

[ASTK02a] M. Athanasaki, A. Sotiropoulos, G. Tsoukalas, and N. Koziris. A Pipelined Execution of Tiled Nested Loops on SMPs with Computation and Communication

Overlapping. In *Proceedings of the Workshop on Compile/Runtime Techniques for Parallel Computing, in conjunction with 2002 International Conference on Parallel Processing (ICPP-2002)*, pages 559–567, Vancouver, Canada, Aug. 2002.

[ASTK02b] M. Athanasaki, A. Sotiropoulos, G. Tsoukalas, and N. Koziris. Pipelined Scheduling of Tiled Nested Loops onto Clusters of SMPs using Memory Mapped Network Interfaces. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing (SC2002)*, Baltimore, Maryland, Nov. 2002. IEEE Computer Society Press.

[Ban88] Uptal Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, 1988.

[Ban93] Uptal Banerjee. *Loop Transformations for Restructuring Compilers*, pages 81–92. Kluwer Academic Publishers, 1993.

[Ban94] Uptal Banerjee. *Loop Parallelization*. Kluwer Academic Publishers, 1994.

[BDRR94] P. Boulet, A. Darte, T. Risset, and Y. Robert. (Pen)-ultimate tiling? *INTEGRATION, The VLSI Jounal*, 17:33–51, 1994.

[BDRV99] P. Boulet, J. Dongarra, Y. Robert, and F. Vivien. Static Tiling for Heterogeneous Computing Platforms. *Parallel Computing*, 25:547–568, 1999.

[Ber66] A. Bernstein. Analysis of Programs for Parallel Programming. *IEEE Transactions on Computers*, 15(5):757–763, Oct. 1966.

[Blu96] M. Blumrich. *Network Interface for Protected, User-Level Communication*. PhD thesis, Princeton University, April 1996.

[BW95] A.J.C. Bik and H.A.G. Wijshoff. Implementation of Fourier-Motzkin Elimination. In *First Annual Conference of the ASCI*, pages 377–386, The Netherlands, 1995.

[CDR97] P. Y. Calland, J. Dongarra, and Y. Robert. Tiling with Limited Resources. In *Application Specific Systems, Architectures, and Processors, ASAP'97*, pages 229–238. IEEE Computer Society Press, July 1997. Extended version available on the web at http://www.ens-lyon.fr/~yrobert.

[CDRV98] P. Y. Calland, A. Darte, Y. Robert, and F. Vivien. On the Removal of Anti and Output Dependences. *International Journal of Parallel Programming*, 26(2):285–312, 1998.

[CKE+04] G. S. Choi, J.-H. Kim, D. Ersoz, A. B. Yoo, and C. R. Das. Coscheduling in Clusters: Is It a Viable Alternative? In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing (SC2004)*, Pittsburgh, PA, USA, Nov. 2004.

[CMZ92] B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. In *Proceedings of the Third Workshop on Compilers for Parallel Computers*, pages 121–160, July 1992.

[CS99] David E. Culler and Jaswinder Pal Singh. *Parallel Computer Architecture - A Hardware/Software Approach*. Morgan Kaufmann, 1999.

[CTHI98] F. O' Carroll, H. Tezuka, A. Hori, and Y. Ishikawa. The Design and Implementation of Zero Copy MPI Using Commodity Hardware with a High Performance Network. In *Proceedings of the International Conference on Supercomputing*, pages 243–249, Melbourne, Australia, 1998.

[CZL95] M. Cierniak, M. Zaki, and W. Li. Loop Scheduling for Heterogeneity. In *Proceedings of the 4th IEEE International Symposium on High Performance Distributed Computing (HPDC'95)*, pages 78–85, Washington D.C., Aug. 1995.

[CZL97] M. Cierniak, M. Zaki, and W. Li. Compile-Time Scheduling Algorithms for a Heterogeneous Network of Workstations. *The Computer Journal*, 40(6):356–372, 1997.

[DDRR97] F. Desprez, J. Dongarra, F. Rastello, and Y. Robert. Determining the Idle Time of a Tiling: New Results. *Journal of Information Science and Engineering*, 14:167–190, March 1997.

[DGAK03] N. Drosinos, G. Goumas, M. Athanasaki, and N. Koziris. Delivering High Performance to Parallel Applications Using Advanced Scheduling. In *Proceedings of the Parallel Computing 2003 (ParCo 2003)*, Dresden, Germany, Sep. 2003.

[DGK+00] I. Drossitis, G. Goumas, N. Koziris, G. Papakonstantinou, and P. Tsanakas. Evaluation of Loop Grouping Methods based on Orthogonal Projection Spaces. In *Proceedings of the International Conference on Parallel Processing*, pages 469–476, Toronto, Canada, Aug. 2000.

[DK04] N. Drosinos and N. Koziris. Performance Comparison of Pure MPI vs Hybrid MPI-OpenMP Parallelization Models on SMP Clusters. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium 2004 (IPDPS 2004)*, page 15, Santa Fe, New Mexico, April 2004.

[DRR96] M. Dion, T. Risset, and Y. Robert. Resource-constrained Scheduling of Partitioned Algorithms on Processor Arrays. *INTEGRATION, The VLSI Jounal*, 20, 1996.

[FHK+91] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran-D Language Specification. Technical Report TR-91-170, Dept. of Computer Science, Rice University, Dec. 1991.

[FLV95] A. Fernandez, J. Llaberia, and M. Valero. Loop Transformations Using Nonunimodular Matrices. *IEEE Transactions on Parallel and Distributed Systems*, 6(8):832–840, Aug. 1995.

[GAK02a] G. Goumas, M. Athanasaki, and N. Koziris. Automatic Code Generation for Executing Tiled Nested Loops Onto Parallel Architectures. In *Proceedings of the 2002 ACM Symposium on Applied Computing (SAC 2002)*, pages 876–881, Madrid, Spain, March 2002.

[GAK02b] G. Goumas, M. Athanasaki, and N. Koziris. Code Generation Methods for Tiling Transformations. *Journal of Information Science and Engineering*, 18(5):667–691, Sep. 2002.

[GAK03] G. Goumas, M. Athanasaki, and N. Koziris. An Efficient Code Generation Technique for Tiled Iteration Spaces. *IEEE Transactions on Parallel and Distributed Systems*, 14(10):1021–1034, Oct. 2003.

[GDAK02a] G. Goumas, N. Drosinos, M. Athanasaki, and N. Koziris. Compiling Tiled Iteration Spaces for Clusters. In *Proceedings of the 2002 IEEE International Conference on Cluster Computing*, pages 360–369, Chicago, Illinois, Sep. 2002.

[GDAK02b] G. Goumas, N. Drosinos, M. Athanasaki, and N. Koziris. Data Parallel Code Generation for Arbitrarily Tiled Nested Loops. In *Proceedings of the 2002 International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 610–616, Las Vegas, Nevada, USA, June 2002.

[GDAK04] G. Goumas, N. Drosinos, M. Athanasaki, and N. Koziris. Automatic Parallel Code Generation for Tiled Nested Loops. In *Proceedings of the 2004 ACM Symposium on Applied Computing (SAC 2004)*, pages 1412–1419, Nicosia, Cyprus, March 2004.

[Gou03] G. Goumas. *Αυτόματη Παραγωγή Παράλληλου SPMD Κώδικα για Μετασχηματισμούς Υπερκόμβων σε Φωλιασμένους Βρόχους*. PhD thesis, School of Electrical and Computer Engineering, National Technical University of Athens, Dec. 2003.

[GSK01] G. Goumas, A. Sotiropoulos, and N. Koziris. Minimizing Completion Time for Loop Tiling with Computation and Communication Overlapping. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium (IPDPS'01)*, San Francisco, April 2001.

[HCF97] K Hogstedt, L. Carter, and J. Ferrante. Determining the Idle Time of a Tiling. In *Principles of Programming Languages (POPL)*, pages 160–173, Jan. 1997.

[HCF99] K. Hogstedt, L. Carter, and J. Ferrante. Selecting Tile Shape for Minimal Execution time. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 201–211, 1999.

[HCF03] K Hogstedt, L. Carter, and J. Ferrante. On the Parallel Execution Time of Tiled Loops. *IEEE Transactions on Parallel and Distributed Systems*, 14(3):307–321, March 2003.

[Hel99] H. Hellwagner. The SCI Standard and Applications of SCI. In H. Hellwagner and A. Reinefield, editors, *Scalable Coherent Interface (SCI): Architecture and Software for High-Performance Computer Clusters*, pages 3–34. Springer-Verlag, Sep. 1999.

[Hol92] E. H. Hollander. Partitioning and Labeling of Loops by Unimodular Transformations. *IEEE Transactions on Parallel and Distributed Systems*, 3(4):465–476, July 1992.

[HP96] M. Haghighat and C. Polychronopoulos. Symbolic Analysis for Parallelizing Compilers. *ACM Transactions on Programming Languages and Systems*, 18(4):477–518, July 1996.

[HP03] J. Hennessy and D. Patterson. *Computer Architecture - A Quantitative Approach.* Morgan Kaufmann Publishers, San Francisco, CA, 3rd edition, 2003.

[HS98] E. Hodzic and W. Shang. On Supernode Transformation with Minimized Total Running Time. *IEEE Transactions on Parallel and Distributed Systems*, 9(5):417–428, May 1998.

[HS02] E. Hodzic and W. Shang. On Time Optimal Supernode Shape. *IEEE Transactions on Parallel and Distributed Systems*, 13(12):1220–1233, Dec. 2002.

[ID98] S. Ioannidis and S. Dwarkadas. Compiler and Run-Time Support for Adaptive Load Balancing in Software Distributed Shared Memory Systems. In *Proceedings of the 4th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers (LCR'98)*, pages 107–122, Pittsburgh, PA, USA, May 1998.

[IT88] F. Irigoin and R. Triolet. Supernode Partitioning. In *Proceedings of the 15th Ann. ACM SIGACT-SIGPLAN Symp. Principles of Programming Languages*, pages 319–329, San Diego, California, Jan. 1988.

[Jim99] M. Jimenez. *Multilevel Tiling for Non-Rectangular Iteration Spaces.* PhD thesis, Universitat Politecnica de Catalunia, 1999.

[KCN91] C.-T. King, W.-H. Chou, and L. Ni. Pipelined Data-Parallel Algorithms: Part II Design. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):430–439, Oct. 1991.

[KCRB03] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. Reducing False Sharing and Improving Spatial Locality in a Unified Compilation Framework. *IEEE Transactions on Parallel and Distributed Systems*, 14(4):337–354, April 2003.

[KMP+95] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. The Omega Library Interface Guide. Technical Report CS-TR-3445, CS Dept., Univ. of Maryland, College Park, March 1995.

[KP96] T. Kim and J. Purtilo. Load Balancing for Parallel Loops in Workstation Clusters. In *Proceedings of the 1996 International Conference on Parallel Processing (ICPP '96)*, Bloomingdale, Illinois, Aug. 1996.

[KRC99] M. Kandemir, J. Ramanujam, and A. Choudary. Improving Cache Locality by a Combination of Loop and Data Transformations. *IEEE Transactions on Computers*, 48(2):159–167, Feb. 1999.

[KSG03] N. Koziris, A. Sotiropoulos, and G. Goumas. A Pipelined Schedule to Minimize Completion Time for Loop Tiling with Computation and Communication Overlapping. *Journal of Parallel and Distributed Computing*, 63(11):1138–1151, Nov. 2003.

[Kul98] D. Kulkarni. Transformations for Improving Data Access Locality in Non-Perfectly Nested Loops. In *Proceedings of 1998 International Conference on Parallel Architectures and Compilation Techniques (PACT'98)*, pages 314–321, Paris, France, 1998.

[Li93] W. Li. *Compiling for NUMA Parallel Machines*. PhD thesis, Cornell Univ., Ithaca, New York, 1993.

[LL98] A. Lim and M. Lam. Maximizing parallelism and minimizing synchronization with affine partitions. *Parallel Computing*, 24:445–475, May 1998.

[LLL01] A. Lim, S. Liao, and M. Lam. Blocking and Array Contraction Across Arbitrarily Nested Loops Using Affine Partitioning. In *Proceedings of the 8th ACM SIGPLAN symposium on Principles and Practices of Parallel Programming (PPoPP'01)*, pages 103–112, Snowbird, Utah, United States, 2001.

[LRW91] M. Lam, E. Rothberg, and M. Wolf. The Cache Performance and Optimizations of Blocked algorithms. In *Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 63–74, Santa Clara, California, April 1991.

[MA01] N. Manjikian and T. S. Abdelrahman. Exploiting Wavefront Parallelism on Large-Scale Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 12(3):259–271, March 2001.

[MHCF98] N. Mitchell, K. Hogsted, L. Carter, and J. Ferrante. Quantifying the Multi-Level Nature of Tiling Interactions. *International J. Parallel Programming*, 1998.

[ML94] E. P. Markatos and T. J. LeBlanc. Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 5(4):379–400, April 1994.

[Mor98] P. Morin. Coarse Grained Parallel Computing on Heterogeneous Systems. In *Proceedings of the 1998 ACM Symposium on Applied Computing (SAC'98)*, pages 628–634, Atlanta, Georgia, United States, 1998.

[MPI94] Message Passing Interface Forum MPIF. A Message-Passing Interface Standard. Technical Report ut-cs-94-230, University of Tennessee, Knoxville, TN, USA, 1994.

[MPI97] Message Passing Interface Forum MPIF. MPI-2: Extensions to the Message-Passing Interface. Technical report, University of Tennessee, Knoxville, TN, USA, July 1997.

[Myr02] Myricom. GM: A Message-Passing System for Myrinet Networks, 2002. `http://www.myri.com/scs/GM/doc/html`.

[OSKO95] H. Ohta, Y. Saito, M. Kainaga, and H. Ono. Optimal Tile Size Adjustment in Compiling General DOACROSS Loop Nests. In *International Conference on Supercomputing*, pages 270–279, New York, 1995. ACM Press.

[PB99] S. Pande and T. Bali. A Computation+Communication Load Balanced Loop Partitioning Method for Distributed Memory Systems. *Journal of Parallel and Distributed Computing*, 58:515–545, 1999.

[PC89] J. Peir and R. Cytron. Minimum Distance: A Method for Partitioning Recurrences for Multiprocessors. *IEEE Transactions on Computers*, 38(8):1203–1211, Aug. 1989.

[PH94] D. Patterson and J. Hennessy. *Computer Organization & Design. The Hardware/Software Interface*. Morgan Kaufmann Publishers, San Francisco, CA, 1994.

[PHP03] N. Park, B. Hong, and V. Prasanna. Tiling, Block Data Layout and Memory Hierarchy Performance. *IEEE Transactions on Parallel and Distributed Systems*, 14(7):640–654, July 2003.

[PTK98] G. Papakonstantinou, P. Tsanakas, and N. Koziris. *Απεικόνιση Αλγορίθμων σε Αρχιτεκτονικές Παράλληλης Επεξεργασίας*, page 33. *Παπασωτηρίου - ΕΠΙΣΕΥ/ΕΜΠ*, Athens, Greece, 1998.

[Pug92] William Pugh. The Omega Test: A fast and Practical Integer Programming Algorithm for Dependence Analysis. *Communications of the ACM*, 35(8):102–114, Aug. 1992.

[PW86] D. Padua and W. Wolfe. Advanced Compiler Optimizations for Supercomputers. *Communications of the ACM*, 29(12), 1986.

[Ram92] J. Ramanujam. Non-Unimodular Loop Transformations of Nested Loops. In *Supercomputing 92*, pages 214–223, Minneapolis, Nov. 1992.

[Ram95] J. Ramanujam. Beyond Unimodular Transformations. *Journal of Supercomputing*, 9(4):365–389, Oct. 1995.

[RR02] F. Rastello and Y. Robert. Automatic Partitioning of Parallel Loops with Parallelepiped-Shaped Tiles. *IEEE Transactions on Parallel and Distributed Systems*, 13(5):460–470, May 2002.

[RR04] L. Renganarayana and S. Rajopadhye. A Geometric Programming Framework for Optimal Multi-Level Tiling. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing (SC2004)*, Pittsburgh, PA USA, Nov. 2004.

[RRP03] F. Rastello, A. Rao, and S. Pande. Optimal task scheduling at run time to exploit intra-tile parallelism. *Parallel Computing*, 29(2):209–239, 2003.

[RS92] J. Ramanujam and P. Sadayappan. Tiling Multidimensional Iteration Spaces for Multicomputers. *Journal of Parallel and Distributed Computing*, 16:108–120, 1992.

[Sak97] R. Sakellariou. A Compile-Time Partitioning Strategy for Non-Rectangular Loop Nests. In *Proceeding of the 1997 International Parallel Processing Symposium (IPPS97)*, 1997.

[SC95] J.-P. Sheu and T.-S. Chen. Partitioning and Mapping Nested Loops for Linear Array Multicomputers. *Journal of Supercomputing*, 9:183–202, 1995.

[SF91] W. Shang and J.A.B. Fortes. Time Optimal Linear Schedules for Algorithms with Uniform Dependences. *IEEE Transactions on Computers*, 40(6):723–742, June 1991.

[SF92] W. Shang and J.A.B. Fortes. Independent Partitioning of Algorithms with Uniform Dependencies. *IEEE Transactions on Computers*, 41(2):190–206, Feb. 1992.

[SG97] R. Sakellariou and J. R. Gurd. Compile-Time Minimization of Load Imbalance in Loop Nests. In *Proceeding of the 1997 International Conference on Supercomputing (ICS97)*, Vienna, Austria, 1997.

[SL99] Y. Song and Z. Li. New Tiling Techniques to Improve Cache Temporal Locality. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation (PLDI'99)*, pages 215–228, Atlanta, Georgia, United States, 1999.

[SLR+95] E. Su, A. Lain, S. Ramaswamy, D. J. Palermo, E. W. Hodges, and P. Banerjee. Advanced Compilation Techniques in the PARADIGM Compiler for Distributed Memory Multicomputers. In *Proceedings of the ACM International Conference on Supercomputing (ICS)*, Madrid, Spain, July 1995.

[Sot04] A. Sotiropoulos. *Αποδοτική Αξιοποίηση Σύγχρονων Δικτυακών Τεχνολογιών στην Παράλληλη Εκτέλεση Υπολογισμών σε Συστοιχίες Υπολογιστών Υψηλών Επιδόσεων*. PhD thesis, School of Electrical and Computer Engineering, National Technical University of Athens, Feb. 2004.

[ST91] J.-P. Sheu and T.-H. Tai. Partitioning and Mapping Nested Loops on Multiprocessor Systems. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):430–439, Oct. 1991.

[STK02] A. Sotiropoulos, G. Tsoukalas, and N. Koziris. Enhancing the Performance of Tiled Loop Execution onto Clusters using Memory Mapped Network Interfaces and Pipelined Schedules. In *Proceedings of the 2002 Workshop on Communication Architecture for Clusters (CAC'02), International Parallel and Distributed Processing Symposium (IPDPS'02)*, Fort Lauderdale, Florida, April 2002.

[TKP00] P. Tsanakas, N. Koziris, and G. Papakonstantinou. Chain Grouping: A Method for Partitioning Loops onto Mesh-Connected Processor Arrays. *IEEE Transactions on Parallel and Distributed Systems*, 11(9):941–955, Sep. 2000.

[TLH94] J. Torrellas, H. Lam, and J. Hennessy. False Sharing and Spatial Locality in Multiprocessor Caches. *IEEE Transactions on Computers*, 43(6):651–663, June 1994.

[TN93] T. Tzen and L. Ni. Trapezoid Self-Scheduling: A Practical Scheduling Scheme for Parallel Compilers. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):87–98, Jan. 1993.

[TOP] Top500 list for november 2004. `http://www.top500.org/lists/2004/11/`.

[TX00] P. Tang and J. Xue. Generating Efficient Tiled Code for Distributed Memory Machines. *Parallel Computing*, 26(11):1369–1410, 2000.

[WL91a] M. Wolf and M. Lam. A Data Locality Optimizing Algorithm. In *ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI)*, Toronto, Ontario, June 1991.

[WL91b] M. Wolf and M. Lam. A Loop Transformation Theory and an Algorithm to Maximize Parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, Oct. 1991.

[XC02] J. Xue and W. Cai. Time-minimal Tiling when Rise is Larger than Zero. *Parallel Computing*, 28(6):915–939, 2002.

[Xue94] J. Xue. Automatic Non-unimodular Loop Transformations for Massive Parallelism. *Parallel Computing*, 20(5):711–728, 1994.

[Xue96] J. Xue. Affine-by-Statement Transformations of Imperfectly Nested Loops. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS'96)*, pages 34–38, Honolulu, Hawaii, Apr. 1996.

[Xue97a] J. Xue. Communication-Minimal Tiling of Uniform Dependence Loops. *Journal of Parallel and Distributed Computing*, 42(1):42–59, 1997.

[Xue97b] J. Xue. On Tiling as a Loop Transformation. *Parallel Processing Letters*, 7(4):409–424, 1997.

[Xue00] Jingling Xue. *Loop Tiling for Parallelism.* Kluwer Academic Publishers, 2000.

[ZLP97] M. Zaki, W. Li, and S. Parthasarathy. Customized Dynamic Load Balancing for a Network of Workstations. *Journal of Parallel and Distributed Computing*, 43(2):156–162, June 1997.