Pipelined Scheduling of Tiled Nested Loops onto Clusters of SMPs using Memory Mapped Network Interfaces

Maria Athanasaki, Aristidis Sotiropoulos, Georgios Tsoukalas and Nectarios Koziris
National Technical University of Athens
Dept. of Electrical and Computer Engineering
Computing Systems Laboratory
e-mail: {maria, sotirop, gtsouk, nkoziris}@cslab.ece.ntua.gr

Abstract

This paper describes the performance benefits attained using enhanced network interfaces to achieve low latency communication. We present a novel, pipelined scheduling approach which takes advantage of DMA communication mode, to send data to other nodes, while the CPUs are performing calculations. We also use zero-copy communication through pinned-down physical memory regions, provided by NIC's driver modules. Our testbed concerns the parallel execution of tiled nested loops onto a cluster of SMP nodes with single PCI-SCI NICs inside each node. In order to schedule tiles, we apply a hyperplane-based grouping transformation to the tiled space, so as to group together independent neighboring tiles and assign them to the same SMP node. Experimental evaluation illustrates that memory mapped NICs with enhanced communication features enable the use of a more advanced pipelined (overlapping) schedule, which considerably improves performance, compared to an ordinary blocking schedule, implemented with conventional, CPU and kernel bounded, communication primitives.

Keywords: memory mapped network interfaces, DMA, pipelined schedules, tile grouping, communication overlapping, SMPs

1 Introduction

Modern high performance communication architectures allow new, low latency messaging protocols [5, 6, 7, 17] to provide the vehicle of very efficient communication in clusters. Available bandwidth is constantly increasing, while there is a trend towards offloading host CPU from the burden of communication [17] through the use of bus mastering, DMA enabled NICs. In this way, CPU has more time to spend on useful application calculations.

When a (user level) process needs to access a conventional network interface, overall communication is delayed [13], since, through a system call, the OS switches to kernel level and assumes the copying of data from user areas to kernel areas for protection. Nevertheless, modern network technologies (i.e. SCI, Myrinet, etc.) are mitigating this startup latency with optimized communication protocols (i.e. VIA) with Zero-Copy [4], DMA support and User-Level [3] characteristics.

Not only these novel network interfaces are reducing the message startup latency, but they can also alleviate the communication burden from CPU. Current parallel applications should be rescheduled to exploit these enhanced features. The parallel execution of any computationally intensive code, containing nested loops, is a very good testbed for such enhanced communication architectures for clusters. Parallel loop execution requires for frequent synchronization points and extensive exchange of data between different nodes. Thus, loops are most suitable for being rescheduled, if we adopt zero-copy, DMA enabled, messaging features. The key issue is to mitigate communication overhead by efficiently controlling the computation to communication grain. When using enhanced network interfaces, the objective should also be to hide as much as possible this communication overhead, gaining extra cycles for useful computation, since the CPU is now disengaged.

In the past, many researchers presented methods for controlling the computation to communication grain for parallel loop execution. In order to alleviate the communication overhead, Irigoin and Triolet proposed supernode partitioning [12] of the Iteration space, where neighboring iteration points are grouped together to build a larger computation node (tile) that can be atomically executed without any in-

0-7695-1524-X/02 $17.00 ©2002 IEEE

tervention. Hodzic and Shang [11] proposed a method to correlate optimal tile size and shape, based on overall completion time reduction. Their approach considers a straightforward time schedule, where each processor executes all tiles along a specific dimension, by interleaving computation and communication phases.

In [9] an alternative method for the problem of scheduling the tiles to single CPU nodes was proposed. Each atomic tile execution involves a communication and a computation phase and this is repeatedly done for all time planes. This sequence of communication and computation phases is compacted, by overlapping them for the different processors. The proposed method acts like enhancing the performance of a processor's datapath with pipelining [14], 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. Since data communications involve some startup latencies, the computation grain is adjusted to make room for this overhead and try to overlap with all communication, which can be done in parallel. Previous work in the field of UET-UCT scheduling of grid graphs in [2], has shown that this schedule is optimal when the computation to communication ratio is one.

In this paper we extend the method proposed in [9] for executing tiled iteration spaces in SMP nodes. We group together neighboring tiles along a hyperplane. Hyperplane-grouped tiles are concurrently executed by the CPUs of the same SMP node. In this way, we eliminate the need for tile synchronization and communication between intranode CPUs. As far as scheduling of groups is concerned, we take advantage of the overlapping schedule of [9] in order to "hide" each group communication volume within the respective computation volume.

We compare our method, using blocking schedules and vertical grouping of neighboring tiles along a specific dimension. Vertically grouped tiles are assigned to the same node, and an optimal hyperplane time schedule is applied. All experimental results show that when the hyperplane grouping of tiles together with the overlapping schedule are applied, the overall completion time is considerably reduced, under the condition of controlling the computation to communication grain. One can easily deduce that the performance of modern communication architectures is enhanced, provided that we carefully design the time schedule and work partitioning among the CPU's of SMP nodes.

The rest of this paper is organized as follows: Basic hardware concepts used in the experiments are introduced in Section 2. In Section 3 the non-overlapping and overlapping schedules are briefly revised. In Section 4 we supply an algorithm for the application of the overlapping scheme proposed in [9] on clusters of SMP nodes and we investigate the resulting time schedule. In Section 5 we describe the experiments executed on a cluster of SMPs using PCI-SCI Network Interface cards in order to verify our theory. Finally, in Section 6 we summarize our results and propose future work.

2 Background concepts

2.1 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 [10], 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.

2.1.1 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 Fig. 1 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.

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

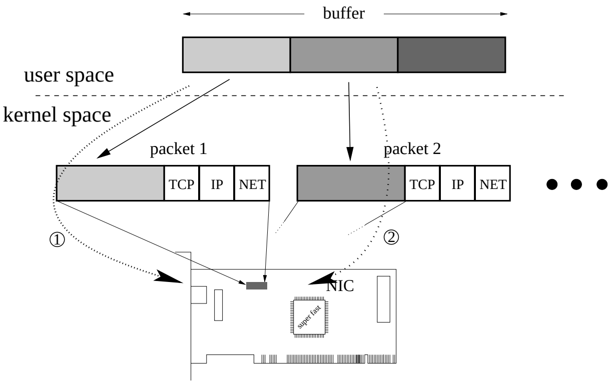

Figure 1. Single-Copy Protocol and packetization process

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.

2.1.2 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. The allocated memory is first "pinned down" and then mapped to user's virtual memory (Fig. 2). 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 Fig. 1 was DMA enabled, a new DMA invocation should take place for each {data,TCP,IP,NET} packet, which would be time consuming.

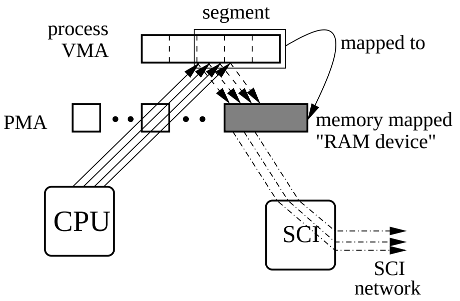

Figure 2. Locked and memory mapped "RAM device" for SCI communications

2.2 Algorithmic model - Notation

In order to evaluate the performance benefits gained by such advanced architectures, we consider algorithms with perfectly nested FOR-loops and uniform data dependencies:

```
 FOR $j_1$=$l_1$ TO $u_1$ DO
    ...
    FOR $j_n$=$l_n$ TO $u_n$ DO
       Loop Body
    ENDFOR
    ...
ENDFOR
```

where l_i, u_i are affine functions of the outer loop indices.

Throughout this paper the following notation is used: N is the set of natural numbers, n is the number of nested FOR-loops of the algorithm. $J^n \subset Z^n$ is the set of loop indices: $J^n = \{j(j_1, ..., j_n) | j_i \in Z \wedge l_i \leq j_i \leq u_i, 1 \leq i \leq n\}$. Each point in this n-dimensional integer space is a distinct instantiation of the loop body. A dependence vector is denoted $d_i = (d_{i1}, ..., d_{in})$. The Dependence Matrix D of an algorithm A is the concatenation of all dependence vectors of this algorithm: $D = [d_1|d_2|...|d_q]$.

Tiling transformation is defined by the $n \times n$ tiling matrix H, or dually by the inverse tiling matrix P, as they are defined in section 2 of [9]. The resulting Tile space J^S and the Tile Dependence matrix D^S are defined as follows: $J^S = \{j^S | j^S = \lfloor Hj \rfloor, j \in J^n\}$, $D^S = \{d^S | d^S = \lfloor H(j_0 + d) \rfloor, d \in D, j_0 \in J^n | 0 \leq \lfloor Hj_0 \rfloor \leq 1\}$ where j_0 denotes the index points belonging to the first complete tile starting from the origin of the Iteration space J^n. The Tile space can be also written as $J^S = \{j^S(j_1^S, \ldots, j_n^S) | j_i^S \in Z \wedge l_i^S \leq j_i^S \leq u_i^S, 1 \leq i \leq n\}$, 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 [1, 8]

Given an algorithm with Dependence matrix D, for a tiling to be legal, it must hold $HD \geq 0$ [12, 15]. In this paper we assume that all dependence vectors are smaller than the tile size, thus they are entirely contained in each supernode's area, which means that $|HD| < 1$ [18] 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^n.

3 Non-overlapping vs. Overlapping Schedule

In [11], Hodzic and Shang have presented a scheme for scheduling loops that have been transformed through a tiling transformation. Their approach is to minimize total execution time, as follows: Firstly the optimal tiling matrix H is determined and then it 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 $T = P(t_{comp} + t_{comm})$, where P is the number of time hyperplanes needed to execute the algorithm, t_{comp} the execution time of a tile and t_{comm} the communication time.

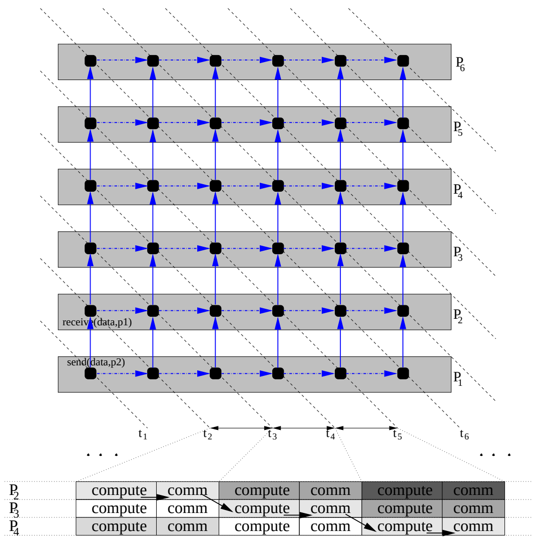
P6
P5
P4
P3
P2
P1
receive(data,p1)
send(data,p2)
t1
t2
t3
t4
t5
t6
. . .
. . .
P2 compute comm compute comm compute comm
P3 compute comm compute comm compute comm
P4 compute comm compute comm compute comm

Figure 3. Non-overlapping Time Schedule

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 Tile space J^S has only the unitary dependence vectors (see §2.2), the optimal linear time schedule can be easily proven to be: $\Pi = [1 \;\; 1 \ldots 1]$. In Fig. 3, the **non-overlapping schedule** is shown for a Tile space using six processors. All tiles along the same dimension are mapped to the same processor. If we cluster together the receive and send subphases and call them "communication subphase", then we see that the overall schedule has computation subphases interleaved with communication ones.

This quite straightforward model of execution results in very good execution times, since it exploits all inherent parallelism at the tile level. However, an important drawback of this execution model 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 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. When communication work is finished, processor receives an interrupt. 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 are not directly dependent to the data computed at this step. A valid time execution scheme 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 (Fig. 4). In this case, every processor computes a tile, and receives+sends data needed in next step or produced in the previous one, respectively.

In Fig. 4 the **overlapping scheduling** is shown. Consider, for example, processor P_3 at k time step: while it makes the computation for 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. The outcome of this schedule is to have successive computations

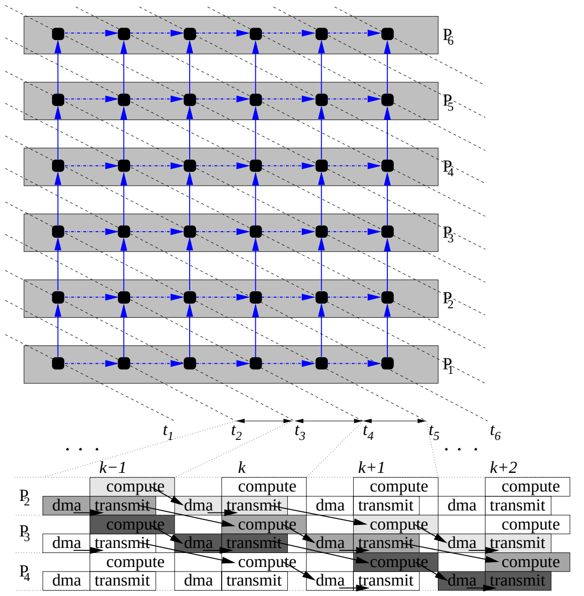

Figure 4. Overlapping Time Schedule

overlapped with communication phases, thus 100% processor utilization. A more detailed description of this schedule can be found in [9] and [16].

If we consider the possibility to overlap computation with communication using the advanced DMA features presented in §2.1, then we 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 sends previously computed data to others as well. Another aspect is that the invocation of DMA communication should be done in user level (User-Level DMA). Furthermore, zero-copy communications should be used, and finally, the software packetization process involved in every communication must be avoided.

According to the previous properties, the total execution time for the overlapping schedule, as deduced from Fig. 4, is given by:

$$T_{overlap} = P(t_{start\_dma} + \max(t_{comp}, t_{comm\_dma}) + t_{synchro}), \quad (1)$$

where P is the number of execution steps of the resulting algorithm. 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.

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.

4 Application of the Overlapping Schedule to SMP nodes

In the sequel, we shall generalize the overlapping schedule proposed in [9] for a cluster of SMP nodes containing more than one CPUs each. In order to mathematically support this generalization, we need to introduce the concept of grouping transformation, which is a supernode transformation applied to tiles.

4.1 Grouping Transformation

We shall group together the tiles of J^S that will be concurrently executed by the CPU's of the same SMP node. That is, we further apply an additional supernode transformation to the Tile space J^S. Thus, from the Tile space J^S we produce the *Group Space*

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

This *grouping transformation* is defined by the $n \times n$ nonsingular matrix H^G. In correspondence to the tiling matrix H [9], we call the $n \times n$ matrix H^G as *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*.

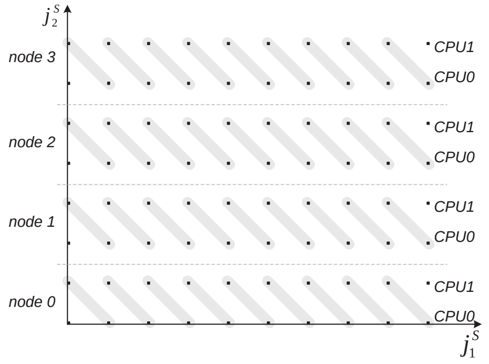

Figure 5. Hyperplane grouping

Example 1: Let us consider a 2-dimensional Tile space J^S and a cluster of SMP nodes containing 2 CPUs each. We want to assign all tiles along of the same dimension j_1^S to the same CPU of an SMP node. Since all CPUs within a node have access to the shared memory, we are assigning neighboring rows of tiles, which exchange data to the CPU's of the same node (Fig. 5). We seek for an appropriate grouping transformation that will group together the tiles that can be executed simultaneously by different CPU's. So, we shall group together the tiles included by the grey areas in Fig. 5. The appropriate inverse grouping matrix is $P^G = \begin{bmatrix} 1 & -2 \\ 0 & 2 \end{bmatrix}$. In the sequel, we shall call this grouping scheme as **hyperplane grouping**. On the contrary, any other grouping scheme along a specific dimension, such as the one presented in Fig. 6, which can be more easily deduced by intuition, will be called **vertical grouping**. It is obvious that the tiles grouped together by a vertical grouping scheme cannot be simultaneously executed unless they are split into subtiles. Thus, additional synchronization overhead is imposed due to subtile dependencies. $\diamond$

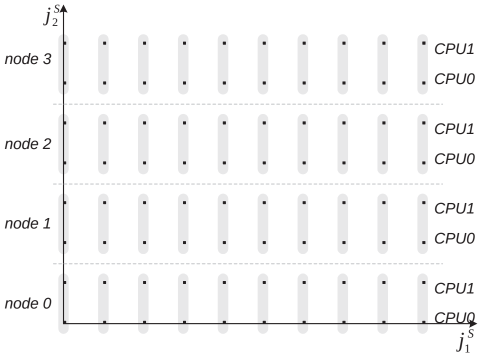

Figure 6. Vertical grouping

4.2 Determining P^G according to the number of CPU's within a node

Consider now the general case, where we have an n-dimensional tiled Iteration space and a cluster of SMP nodes, each with m processors inside. Our objective is to assign the tiles of J^S along of the 1-st dimension to the same CPU of an SMP node. Let us assume that the natural number m can be written as $m = m_2 \times m_3 \times \ldots \times m_n$, where $m_2, m_3, \ldots, m_n \in N$. Then, we select the inverse grouping matrix and the corresponding grouping matrix to be

$$P^G = \begin{bmatrix} 1 & -m_2 & \ldots & -m_n \\ 0 & m_2 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & m_n \end{bmatrix},$$
$$H^G = (P^G)^{-1} = \begin{bmatrix} 1 & 1 & \ldots & 1 \\ 0 & \frac{1}{m_2} & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & \frac{1}{m_n} \end{bmatrix} \quad (2)$$

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

In order to prove that H^G defines a legal grouping transformation, it suffices to prove that $H^G D^S \geq 0$, where D^S is the dependence matrix of the Tile Space J^S and that any two tiles $(j^S, j^{S'} \in j^G)$ within the same group are independent. We have assumed (see §2.2) 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 $j^S, j^{S'} \in J^S$ belong to the same group j^G then it holds that: $\lfloor H^G j^S \rfloor = \lfloor H^G j^{S'} \rfloor \Rightarrow j_1^S + j_2^S + \ldots + j_{n-1}^S + j_n^S = j_1^{S'} + j_2^{S'} + \ldots + j_{n-1}^{S}{}' + j_n^{S'}$ In addition, if there is a direct or an indirect dependence from j^S to $j^{S'}$ then it holds that $j^{S'} = j^S + \sum_{i=1}^n \lambda_i d_i$, where $\lambda_i \in N$ and d_i is a unitary dependence vector. Thus, $j_i^{S'} = j_i^S + \lambda_i$, $i = 1, \ldots, n$. Therefore, the equality $j_1^S + j_2^S + \ldots + j_{n-1}^S + j_n^S = j_1^{S'} + j_2^{S'} + \ldots + j_{n-1}^{S}{}' + j_n^{S'}$ can be rewritten as follows: $\lambda_1 + \lambda_2 + \ldots + \lambda_n = 0$. As $\lambda_1, \ldots, \lambda_n \in N$, it holds that $\lambda_1 = \ldots = \lambda_n = 0$. Consequently, there is no direct or indirect dependence between two tiles belonging to the same group $j^G \in J^G$ and all tiles of a group in J^G can be computed simultaneously by the CPU's of an SMP node. Thus the above grouping transformation is valid according to our algorithmic model.

Example 2: We have a cluster of SMP nodes with 2 CPU's and a NIC each. We assume a 2-dimensional rectangular Tile space J^S. Let us assign the tiles along of the dimension j_1^S to the same CPU, as indicated in Fig. 7 by the grey arrows. The CPU's of the same SMP node will undertake two neighboring rows of tiles.

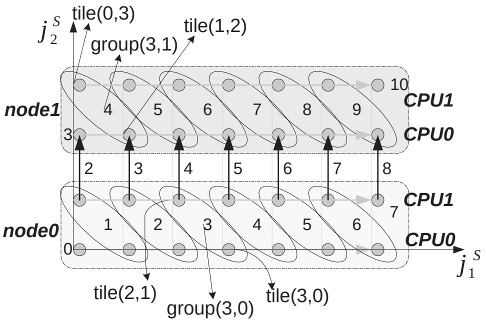

Figure 7. 2D example

Then, during the time step t=0, the CPU-0 of the SMP node0 computes tile $(0, 0)$. During the time step $t = 1$ the

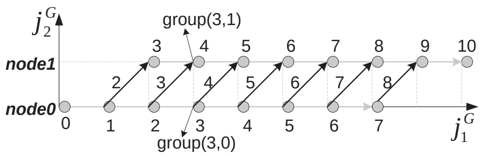

Figure 8. Group Space for the 2D example

CPU-0 of node0 computes tile $(1,0)$, while the CPU-1 of the same SMP node computes tile $(0,1)$. Similarly, during the time step $t = 2$ the CPU-0 computes tile $(2,0)$, while the 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 node1. During the time step t=3, the CPU's of node0 can continue the execution as above, while the CPU's of node1 start executing the same routine with the rows of tiles $(x,2)$ and $(x,3)$.

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 perform *grouping* to the Tile space J^S, as indicated in Fig. 7 and derive the Group Space J^G. The appropriate grouping matrices according to the formula (2) for this case are $P^G = \begin{bmatrix} 1 & -2 \\ 0 & 2 \end{bmatrix}$ and $H^G = (P^G)^{-1} = \begin{bmatrix} 1 & 1 \\ 0 & \frac{1}{2} \end{bmatrix}$. 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 $j^G = \lfloor H^G(1,0)^T \rfloor = \lfloor H^G(0,1)^T \rfloor = (1,0)^T$. Similarly, tiles $(2,0)$ and $(1,1)$ are grouped together in $j^G = (2,0)^T$. In Fig. 7 the time step when each group will be computed is shown, together with the time step where each data transfer will take place. In Fig. 8, the corresponding Group Space is also shown.

Table 1. Execution steps of a 2-D example on a cluster of SMP nodes with 2 CPU's each

time	node0			node1		
step	CPU0	CPU1	group	CPU0	CPU1	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	$\begin{pmatrix}4\\0\end{pmatrix}$	$\begin{pmatrix}3\\1\end{pmatrix}$	$\begin{pmatrix}4\\0\end{pmatrix}$	$\begin{pmatrix}1\\2\end{pmatrix}$	$\begin{pmatrix}0\\3\end{pmatrix}$	$\begin{pmatrix}3\\1\end{pmatrix}$

In Table 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 $j^G = (j_1^G, j_2^G) \in J^G$ will be executed during the time step $t(j^G) = j_1^G + j_2^G$ in the SMP node j_2^G. Therefore the linear time scheduling vector for this example is $\Pi^G = (1,1)$. $\diamond$

4.3 Linear time schedule of groups

In order to produce an optimal time schedule, 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 may be written as a sum of unitary dependence vectors. So it will be included in the following proof as an indirect dependence. Thus, applying the above grouping transformation, the 1-st column-vector of the Tile Dependence matrix $D^S = I$ is transformed to the vector $d_n^{G'} = H^G d_n^S = (1,0,\ldots,0)^T$. In addition, the j-th column-vector of the Dependence matrix $D^S = I$, $j = 2,\ldots,n$, is transformed to the vector $d_j^{G'} = H^G d_j^S = (1,0,\ldots,0,\frac{1}{m_j},0,\ldots,0)^T$. This vector imposes the dependencies $(1,0,\ldots,0, \lfloor\frac{1}{m_i}\rfloor,0,\ldots,0)^T = (1,0,\ldots,0,0,0,\ldots,0)^T$ and $(1,0,\ldots,0,\lceil\frac{1}{m_j}\rceil,0,\ldots,0)^T = (1,0,\ldots,0,1,0,\ldots,0)^T$ in the Group Space. 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 $j^G \in J^G$ is computed during the time step $t = \Pi^G j^G$. Consider the last $(n-1)$ coordinates of a group indicating which SMP node of the cluster will execute this group. Then the groups $j^G = (j_1^G,\ldots,j_n^G)$ and $j^{G'} = (j_1^G+1, j_2^G,\ldots,j_n^G)$ 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 $\Pi^G j^{G'} - \Pi^G j^G = \pi_1^G$ 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 the groups $j^G = (j_1^G,\ldots,j_n^G)$ and $j^{G'} = (j_1^G+1, j_2^G,\ldots,j_{i-1}^G, j_i^G+1, j_{i+1}^G,\ldots,j_n^G)$. These groups are executed in neighboring SMP nodes, thus a communication step is required between their computation steps. It means that their time distance $\Pi^G j^{G'} - \Pi^G j^G = \pi_1^G + \pi_i^G$ 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 [16],[9], for the single CPU pipelined schedule, the Π was $(1,2,\ldots,2)$ according to the UET-UCT theory. 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 Fig. 7 and the resulting Group Space in Fig. 8 the relative positions of groups $(3, 0)$ and $(3, 1)$), so the optimal overlapping schedule is achieved by $(1, 1, \ldots, 1)$.

4.4 CPU Tile Assignment

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 $j^S = (j_1^S, \ldots, j_n^S)$, belonging to group $j^G = (j_1^G, \ldots, j_n^G)$, 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 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 $j^S = (j_1^S, \ldots, j_n^S)$ 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 inverse grouping matrix diagonal elements m_x equals to 1 then the corresponding coordinate of the CPU identification vector can be omitted, as it will always equal 0.

4.5 Generalization: Grouping 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 inverse grouping matrix and the corresponding grouping matrix are

$$P^G = \begin{bmatrix} m_1 & \ldots & 0 & 0 & 0 & \ldots & 0 \\ \vdots & \ddots & \vdots & \vdots & \vdots & & \vdots \\ 0 & \ldots & m_{i-1} & 0 & 0 & \ldots & 0 \\ -m_1 & \ldots & -m_{i-1} & 1 & -m_{i+1} & \ldots & -m_n \\ 0 & \ldots & 0 & 0 & m_{i+1} & \ldots & 0 \\ \vdots & & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \ldots & 0 & 0 & 0 & \ldots & m_n \end{bmatrix},$$

$$H^G = \begin{bmatrix} \frac{1}{m_1} & \ldots & 0 & 0 & 0 & \ldots & 0 \\ \vdots & \ddots & \vdots & \vdots & \vdots & & \vdots \\ 0 & \ldots & \frac{1}{m_{i-1}} & 0 & 0 & \ldots & 0 \\ 1 & \ldots & 1 & 1 & 1 & \ldots & 1 \\ 0 & \ldots & 0 & 0 & \frac{1}{m_{i+1}} & \ldots & 0 \\ \vdots & & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \ldots & 0 & 0 & 0 & \ldots & \frac{1}{m_n} \end{bmatrix}, \quad (3)$$

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

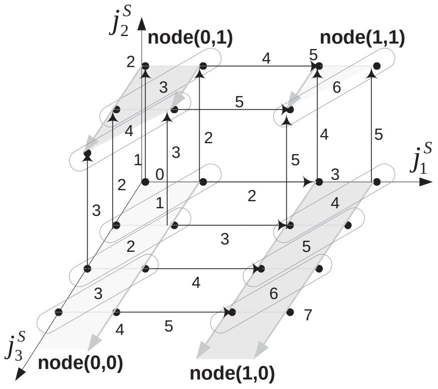

Figure 9. 3D example

Example 3: We have a cluster of SMP nodes with 2 CPU's and a NIC each. We assume a 3-dimensional rectangular Tile space J^S. Let us assign the tiles along of the dimension j_3^S to the same CPU, as indicated in Fig. 9 by the grey arrows. The CPU's 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 (3), we choose the grouping matrices to be

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

In Fig. 9 we show the grouping of tiles and when each computation step and each communication step will be executed. 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 $t(j^G) = j_1^G + j_2^G + j_3^G$. Therefore, the linear time scheduling vector for this example is $\Pi^G = (1, 1, 1)$.

Let us assume that the rectangular Tile space has bounds: $0 \leq j_1^S < j_{1max}^S$, $0 \leq j_2^S < j_{2max}^S$, $0 \leq j_3^S < j_{3max}^S$, where j_{1max}^S is an even number. Then the bounds of the corresponding Group Space will be $0 \leq j_1^G \leq \lfloor \frac{j_{1max}^S - 1}{2} \rfloor = \frac{j_{1max}^S}{2} - 1, 0 \leq j_2^G \leq j_{2max}^S - 1, 0 \leq j_3^G \leq j_{1max}^S + j_{2max}^S + j_{3max}^S - 3$. During the first time step $t = 0$, the group $(0, 0, 0)$ will be computed. During the last time step $t = \frac{3j_{1max}^S}{2} + 2j_{2max}^S + j_{3max}^S - 5$, the group $(\frac{j_{1max}^S}{2} - 1, j_{2max}^S - 1, j_{1max}^S + j_{2max}^S + j_{3max}^S - 3)$ will be computed. Thus, the number of steps required for the completion ofthe algorithm is $P = \frac{3j_{1max}^S}{2} + 2j_{2max}^S + j_{3max}^S - 4$.

4.6 Comparison

In this section we shall compare vertical grouping, which is indicated in Fig. 6, with the proposed scheme of hyperplane grouping, which is shown in Fig. 6, 7 in the case of a 2-dimensional algorithm and a cluster of SMP's with 2 CPU's each.

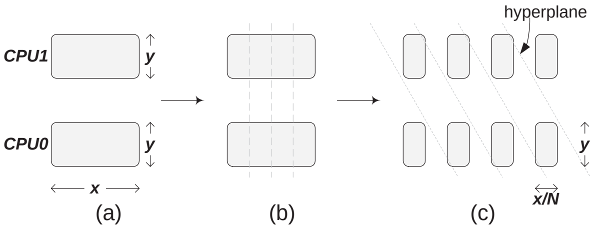

Figure 10. Splitting tiles in vertical scheme

As we have already mentioned, vertical grouping cannot exploit the computational power of both CPU's of our SMP's unless we split each tile into smaller subtiles and compute some of them in parallel, as shown in Fig. 10. Let us assume that a CPU needs time α for the computation of a tile with dimensions x, y (Fig. 10a). Consequently, it will need time $\frac{\alpha}{N}$ for the computation of a respective subtile with dimensions $\frac{x}{N}$, y (Fig. 10c). The subtiles which are created can be computed by 2 CPU's in $N + 1$ computational steps, interleaved with N synchronization steps, following an optimal linear time schedule $(1, 1)$ as in Fig. 10c. If the average time consumed for the synchronization of 2 CPU's of an SMP node is t_{synch_in}, then the total time required for the computation of a pair of initial tiles is:

$$\beta = \alpha \frac{N + 1}{N} + N t_{synch\_in}. \tag{4}$$

The time required for the computation of a pair of tiles is minimized when

$$N = \sqrt{\frac{\alpha}{t_{synch\_in}}}. \tag{5}$$

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

If we consider an Iteration Space with size $X \times Y$, tiled with rectangular tiles with size $x \times y$, (for example in Fig. 6,5 we have $\frac{X}{x} = 10, \frac{Y}{y} = 8$), 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 algorithm is $P = \frac{X}{x} + \frac{Y}{2y} - 1$. The minimum duration of a time step 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 $T_{blocking,vertical} = P(\beta_{min} + t_{comm}) \simeq (\frac{X}{x} + \frac{Y}{2y})(\beta_{min} + t_{comm})$.

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 algorithm is $P = \frac{X}{x} + \frac{Y}{y} - 2$. According to the formula $T_{overlap} = P(t_{start\_dma} + \max(t_{comp}, t_{comm\_dma}) + t_{synchro})$ (explained in [9]), 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 $T_{non-blocking,vertical} = P(t_{start\_dma} + max(\beta_{min}, t_{comm\_dma}) + t_{synchro}) \simeq (\frac{X}{x} + \frac{Y}{y})(t_{start\_dma} + max(\beta_{min}, t_{comm\_dma}) + t_{synchro})$. 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 algorithm is $P = \frac{X}{x} + \frac{3Y}{2y} - 2$. According to the formula $T_{overlap} = P(t_{start\_dma} + \max(t_{comp}, t_{comm\_dma}) + t_{synchro})$, 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 $T_{non-blocking,hyperplane} = P(t_{start\_dma} + max(\alpha, t_{comm\_dma}) + t_{synchro}) \simeq (\frac{X}{x} + \frac{3Y}{2y})(t_{start\_dma} + max(\alpha, t_{comm\_dma}) + t_{synchro})$. 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, the overlapping scheme in combination with vertical grouping is more efficient than the non-overlapping scheme, in case that $\beta_{min} \geq t_{comm}$, when $t_{comm} > \frac{\lambda}{2}(t_{start\_dma} + \beta_{min} + t_{synchro})$. 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 a comparatively great amount of computations should be performed within each node. 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 CPU's.

5 Experimental results

In [16] the pipelined schedule proposed in [9] was applied, using a cluster of single CPU nodes with Dolphin's PCI-SCI NICs. In this paper, in order to evaluate the proposed methods, we now use a Linux SMP cluster with 8 identical nodes. Each node had 128M of RAM and 2 Pentium III 800 MHz CPU's. The cluster nodes are interconnected with an SCI ring, using SCI Dolphin's PCI-SCI D330 cards. SCI NICs support shared memory programming, either through PIO messaging or through DMA. We are using the NICs' kernel-level DMA support for messaging. Invoking kernel system calls, causes extra CPU cycles overhead. However, we avoid extra copying from user space to kernel space (physical memory) when using DMA by allocating user level pages, which correspond to physically contiguous pre-reserved memory regions, for DMA communications.

We performed several series of experiments in order to evaluate and compare the practical execution times of vertical vs. hyperplane grouping schemes and blocking vs. non-blocking schemes. The hyperplane grouping scheme, in combination with non-blocking communication among the SMP nodes resulted in the minimum total execution times.

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 the k-axis are mapped onto the same processor, as proposed in [9]. 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 of the dimensions i and j and $\frac{Z}{z}$ tiles along of the 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.

Table 2. 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)
	}

We applied both vertical and hyperplane grouping, using both blocking and non-blocking communication primitives. 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.

First of all, as far as the implementation of vertical grouping is concerned, we experimentally verified formula (5), in order to find 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 dependencies. The experimental results, as long as the theoretically expected curve, are plotted in Fig. 11. The theoretical plot was calculated using the formula (4) 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 $0,15\%$. 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 trivial, 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.

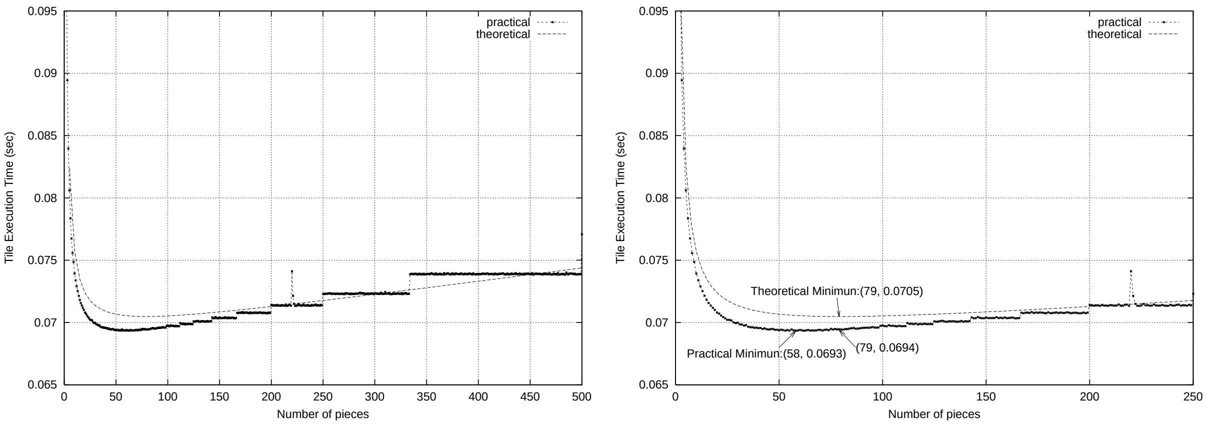

Figure 11. Vertical grouping - Tile execution time in respect to the number of slices a tile is cut

Table 3. Overlapping scheme Implementation

Thread 0:	*Thread 1:*	*Explanation*
foreach group assigned to node(i,j) do{	foreach group assigned to node(i,j) do{	
trigger_interrupt to node(i-1,j)		Inform “previous” nodes:
trigger_interrupt to node(i,j-1)	trigger_interrupt to node(i,j-1)	“I am ready to accept data”
	wait_interrupt from node(i+1,j)	Wait until “next” nodes
wait_interrupt from node(i,j+1)	wait_interrupt from node(i,j+1)	are ready to accept data
	send_dma(node(i+1,j),data)	Initialization of DMA transfer
send_dma(node(i,j+1),data)	send_dma(node(i,j+1),data)	to neighboring nodes
compute_tile(i,j,k,CPU0)	compute_tile(i,j,k,CPU1)	
	wait_dma()	Wait for DMA to complete
wait_dma()	wait_dma()	
	trigger_interrupt to node(i+1,j)	Inform “next” nodes:
trigger_interrupt to node(i,j+1)	trigger_interrupt to node(i,j+1)	“Your data has arrived”
wait_interrupt from node(i-1,j)		Wait until “previous” nodes
wait_interrupt from node(i,j-1)	wait_interrupt from node(i,j-1)	have finished sending data
semaphore_post(sem_s1)	semaphore_post(sem_s2)	Implementation of a barrier
semaphore_wait(sem_s2)	semaphore_wait(sem_s1)	
}	}	

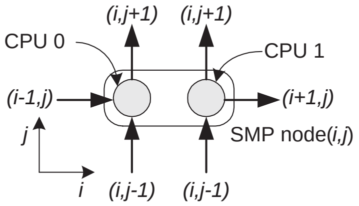

Figure 12. CPU communication directions

Once vertical grouping was implemented and precisely approximated with a theoretical formula, we implemented both blocking and non-blocking communication schemes.As far as the blocking communication scheme is concerned, it was implemented using the pseudo-code of Table 2. On the other hand, the non-blocking scheme was implemented using the pseudo-code of Table 3, because 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 $(i+1,j)$,$(i,j+1)$ (Fig. 12). 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 compute_tile(i,j,k,CPUx) procedure. In order to implement vertical grouping we used the pseudocode of Table 4. The number of subtiles inside a tile was selected according to the formula (5). Notice that, the implementation of hyperplane grouping was much simpler as it is shown in Table 4.

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 [16],[9],[11]). The experimen-

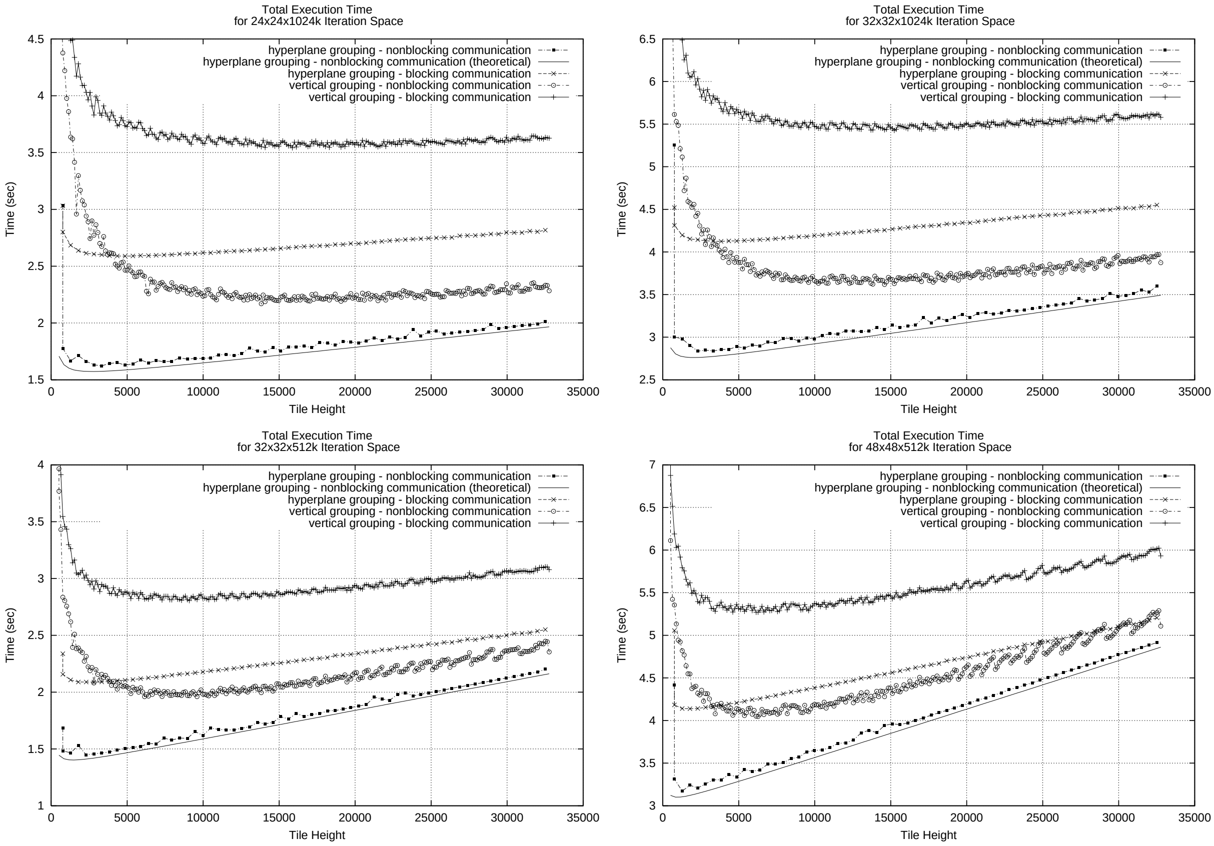

Figure 14. Experimental Results

Table 4. Vertical vs. Hyperplane Grouping

Vertical Grouping Implementation	
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 Implementation	
compute_tile(i,j,k,CPU0):	*compute_tile(i,j,k,CPU1):*
compute each iteration of this tile	compute each iteration of this tile

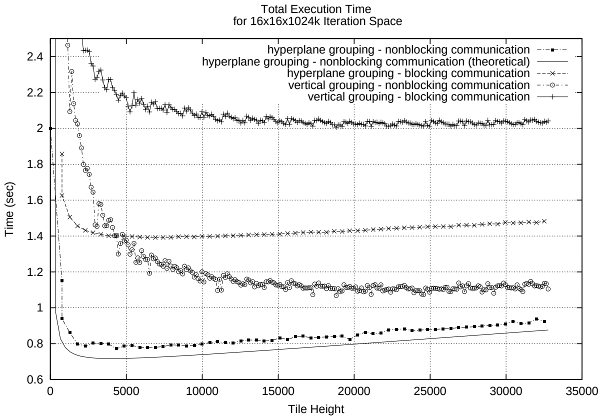

Figure 13. Experimental Results

tal results, shown in Figs 13,14, illustrate that in every case non-blocking communication (overlapping schedule) is preferable to blocking communication (non-overlapping schedule) and hyperplane grouping is preferable to vertical grouping. The lowest minimum is clearly achieved when using hyperplane grouping in combination with overlapping schedule, in all cases.

As far as hyperplane grouping, in combination with non-blocking communication, is concerned, according to our scheduling theory, as in Example 3, the number of time steps required for the completion of an experiment is $P(x, y, z) = \frac{3X}{2x} + \frac{2Y}{y} + \frac{Z}{z} - 4$. Thus, according to the formula (1), $T_{non-blocking,hyperplane} = (\frac{3X}{2x} + \frac{2Y}{y} + \frac{Z}{z} - 4)(t_{start\_dma} + t_{comp} + t_{synchro})$. This formula was used to produce the theoretical curves of Figs 13, 14 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 $39,6nsec$.

It can be easily verified from Figs 13,14 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 results.

6 Conclusions - Future Work

In this paper we presented a novel approach for the time scheduling of tiled nested loops on a cluster of SMP nodes using advanced features (DMA, Zero Copy) of latest communication architectures. We minimized the total execution time by overlapping the computation with communication (as in [16],[9]). In addition, we achieved the maximum CPU's utilization with a proper grouping transformation. What remains open is an analytical computation of the parameters $m_1, \ldots, m_n$ of our grouping matrix according to the initial space shape and communication minimization criteria. Furthermore, a study on the use of multi-NIC SMP nodes can be carried out.

References

[1] 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, Apr 1991.

[2] 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.

[3] M. Blumrich. *Network Interface for Protected, User-Level Communication*. PhD thesis, Princeton University, Apr 1996.

[4] 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.

[5] T. Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, pages 40–53, Copper Mountain, Colorado, Dec 1995.

[6] K. Ghouas, K. Omang, and H. Bugge. VIA over SCI - Consequences of a Zero Copy Implementation and Comparison with VIA over Myrinet. In *Proceedings of the Workshop on Communication Architecture for Clusters (CAC' 2001) in conjunction with Int'l Parallel and Distributed Processing Symposium (IPDPS '01)*, San Francisco, Apr 2001.

[7] F. Giacomini, T. Amundsen, A. Bogaerts, R. Hauser, B. Johnsen, H. Kohmann, R. Nordstrom, and P. Werner. Low Level SCI software functional specification-Software Infrastructure for SCI. ESPRIT Project 23174. http://www.dolphinics.com/ downloads/ nt/ pdf_zip/ SISCI_API-2_1_1.pdf.

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

[9] G. Goumas, A. Sotiropoulos, and N. Koziris. Minimizing Completion Time for Loop Tiling with Computation and Communication Overlapping. In *Proceedings of IEEE Int'l Parallel and Distributed Processing Symposium (IPDPS'01)*, San Francisco, Apr 2001. (best paper award).

[10] 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.

[11] E. Hodzic and W. Shang. On Supernode Transformation with Minimized Total Running Time. *IEEE Trans. on Parallel and Distributed Systems*, 9(5):417–428, May 1998.

[12] 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.

[13] V. Karamcheti and A. Chien. Software Overhead in Messaging Layers: Where Does the Time Go? In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 51–60, Oct 1994.

[14] D. Patterson and J.Hennessy. *Computer Organization & Design. The Hardware/Software Interface*, pages 364–367. Morgan Kaufmann Publishers, San Francisco, CA, 1994.

[15] J. Ramanujam and P. Sadayappan. Tiling Multidimensional Iteration Spaces for Multicomputers. *Journal of Parallel and Distributed Computing*, 16:108–120, 1992.

[16] 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), Int'l Parallel and Distributed Processing Symposium (IPDPS'02)*, Fort Lauderdale, Florida, Apr 2002.

[17] The Virtual Interface Specification Version 1.0. http://www.viarch.org.

[18] J. Xue. On Tiling as a Loop Transformation. *Parallel Processing Letters*, 7(4):409–424, 1997.