Memory and Network Bandwidth Aware Scheduling of Multiprogrammed Workloads on Clusters of SMPs *

Evangelos Koukis and Nectarios Koziris
National Technical University of Athens
School of Electrical and Computer Engineering
Computing Systems Laboratory
{vkoukis, nkoziris}@cslab.ece.ntua.gr

Abstract

Symmetric Multiprocessors (SMPs), combined with modern interconnection technologies are commonly used to build cost-effective compute clusters. However, contention among processors for access to shared resources, as is the main memory bus and the NIC can limit their efficiency significantly. In this paper, we first provide an experimental demonstration of the effect of resource contention on the total execution time of applications. Then, we present the design and implementation of an informed gang-like scheduling algorithm aimed at improving the throughput of multiprogrammed workloads on clusters of SMPs. Our algorithm selects the processes to be coscheduled so as not to saturate nor underutilize the memory bus or network link bandwidth. Its input data are acquired dynamically using hardware monitoring counters and a modified Myrinet NIC firmware, without any modifications to existing application binaries. Experimental evaluation shows throughput can improve up to 40-48% compared to the standard Linux 2.6 O(1) scheduler.

Keywords: scheduling, SMP clusters, multiprogramming, memory bandwidth, resource contention, performance counters, Linux, Myrinet

1 Introduction

Symmetric Multiprocessors, or SMPs for short, have emerged as a cost-effective solution for constructing scalable clustered systems, when interconnected over a low-latency networking infrastructure. However, their symmetric design, according to which most system resources are shared equally between all processors in the system can have negative impact on their performance and impose significant barriers to scalability, due to processor contention. When SMP nodes are combined to form large compute clusters, two resources for which there is major contention are bandwidth to the shared main memory of each node, and network I/O bandwidth on each node's communication link.

The imbalance between the ever increasing speed of CPUs vs. the relatively slow advances in memory technology has long been the focus of scientific research. As the CPU speed of the fastest available microprocessors increases exponentially, while the speed of memory devices is growing at a slow rate of about 7% per year [14], the ratio of CPU to memory performance or "Machine Balance" [12] becomes a deciding factor in determining overall system performance. The inability of memory to cope with currently available CPUs becomes even more apparent when more than one processors contend for access to data in main memory.

The limited memory bus bandwidth problem in SMPs is aggravated when building clusters of SMPs. Cluster nodes are usually interconnected over high performance interconnection networks such as Myrinet [6], or Infiniband [3]. To relieve the CPU from the communication burden, their NICs feature embedded microprocessors and DMA engines, which undertake almost all network protocol processing leaving the CPU free to perform useful calculations. However, as interconnect technology advances and the available link bandwidth increases rapidly, so does the memory bus bandwidth consumption of the NIC relative to the CPUs of the system, further adding to the problem and adversely affecting the degree of computation to communication overlapping that can be achieved.

When executing multiprogrammed workloads on clusters of SMPs, the effect of *process skew* on the efficiency of the system must also be taken into account. Many parallel applications comprise successive computation and communication steps, with synchronization operations between

*This research is supported by the Pythagoras II Project (EPEAEK II), co-funded by the European Social Fund (75%) and National Resources (25%).

(e.g. barrier primitives). When only local OS scheduling policies are applied, without regard to synchronization, it is often the case that one of the processes of the application may be delayed in reaching a synchronization point, for example because it has been forced to leave the CPU. This prevents all its peers from making progress, and increases the overall execution time. To facilitate the efficient execution of multiprogrammed workloads, gang scheduling techniques [7, 8, 9] have been proposed. Gang scheduling is based on allocating concurrently all required CPUs to the processes of a parallel application whenever it is scheduled to run, thus minimizing the time wasted at synchronization points due to process skew.

In this paper, we try to address the problem of limited memory and network bandwidth by designing and implementing a bandwidth aware, gang-like scheduling policy, adapted to the execution of multiprogrammed workloads on clusters of SMPs. Our policy selects the processes to be coscheduled, aiming at minimizing the interference on the shared memory bus and the NIC communication link of each SMP. In previous work [10], we explored the application of such techniques on a local scale, trying to alleviate the impact of contention on the shared memory bus of a single SMP node. We now extend this approach to a cluster-wide scale and apply it both to contention for access to memory and to contention for access to communication link bandwidth, on every SMP node.

Our goal is to monitor application demands for each of the contented-for resources at run-time, and then use this information to select jobs to be coscheduled, so that processes executing concurrently in the same SMP neither saturate not underutilize the shared memory bus and the node NIC. The scheduler is designed so that no modifications to existing application binary code is required. The required monitoring data are acquired transparently to executing application code, by means of the performance monitoring counters provided by most modern microprocessors [21]. Moreover, custom modifications to the firmware executing on the NICs are introduced, in order to allow estimation of the memory bandwidth consumed by DMA engines on the NIC, and to monitor the contention on the NIC's send and receive queues.

In the rest of this paper, we begin by presenting related work in the area, then demonstrate the problem of memory and network contention by measuring the execution slowdown imposed on mixed benchmark workloads (Section 3). Based on our observations, we propose a bandwidth aware scheduling policy to alleviate the problem (Section 4). Section 5 describes the performance monitoring framework we designed, both at the CPU as well as at the NIC firmware side and a proof-of-concept scheduler implementation. Finally we present an experimental evaluation of our scheduling policy compared to the standard Linux scheduler (Section 6) and our conclusions (Section 7).

2 Related work

Many research efforts aim at mitigating the effects of memory contention on SMPs by using the cache hierarchy in memory conscious ways. On one hand, there are techniques [1, 17, 18] aiming at better exploiting available caches by employing sub-blocking and partitioning techniques, in order to improve the locality of references and minimize the cache miss rate. On the other hand, schedulers in modern OSs incorporate CPU affinity constraints [20, 19, 16], in order to avoid the increased memory load as a recently migrated process rebuilds its state information in a new processor's private cache hierarchy.

The effect of limited memory bandwidth on process execution in the context of soft- and hard- real time systems has been investigated in [11] and in [5], where techniques are presented to satisfy guaranteed memory bandwidth demands and to throttle lower priority processes so that they do not interfere with the execution of higher priority ones. The impact of contention while accessing the shared memory bus is an aspect of this work, as well, however we seek to increase system throughput in multiprogrammed clusters of SMPs, rather than meet strict deadlines.

The work in [2] is similar to ours, describing a system which uses source code hooks to track the memory bandwidth usage of each application and coordinate their execution, on a single SMP machine. Our approach is based on a monitoring framework which requires no application source code changes, but instead relies on OS and NIC firmware mechanisms in order to transparently monitor memory bandwidth usage. Furthermore, we target clusters of SMPs, taking into account the pressure on the memory bus imposed by the NIC and monitoring contention on the shared communication link while executing communication-intensive applications.

3 Application slowdown due to memory and network contention

In this section, we demonstrate the impact of memory and network contention on overall system performance, by quantifying the slowdown imposed on the execution of computation-intensive workloads. We try to highlight separately the effects of saturation on the shared memory bus and the effects of contention on the cluster interconnect, while at the same time ensuring that the processes being executed do not share processor time or other system resources, and only contend for access to main memory or to the NIC.

Our set of benchmarks, used both to demonstrate the

problem of resource contention and to evaluate the effectiveness of our scheduling policy, comprise the BT, CG, EP, FT, IS, LU, MG, SP applications from the NAS Parallel Benchmarks suite [4].We also developed two microbenchmarks, membench and myribench, which are designed to induce varying degrees of memory and network traffic. A brief description of the set of benchmarks is presented in Fig. 1.

Name	Description	BW req.
BT	Block Tridiagonal Solver	373MB/s
CG	Conjugate Gradient	674MB/s
EP	Random Number Generator	162MB/s
FT	3D Fast Fourier Transform	384MB/s
IS	Integer Sort	475MB/s
LU	LU Solver	491MB/s
MG	3D Multigrid	565MB/s
SP	3D Multi-partition algorithm	567MB/s

Figure 1. Description of benchmarks used

Our experimental platform is a four node SMP cluster. Each node has two Pentium III@1266MHz processors on a Supermicro P3TDE6 motherboard with the Serverworks ServerSet III HC-SL chipset. Two PC133 SDRAM 512MB DIMMs are installed for a total of 1GB RAM per node. Each processor has 16KB L1 I cache, 16KB L1 D cache and 512KB unified L2 cache, with 32 bytes per line. For cluster interconnection each node has a Myrinet M3F-PCIXD-2 NIC in a 64bit/66MHz PCI slot, connected to an M3-SW16-8F line card. The NICs use the LanaiXP@225MHz embedded processor with 2MB of SRAM.

The OS installed is Linux, kernel version 2.6.11. We use the 2.6 branch of the kernel, mainly for its use of the new enterprise-class O(1) scheduler, which has been completely rewritten since 2.4, with cache affinity and SMP scalability in mind. Codes were compiled with the Intel C Compiler v8.1, with maximum optimization for our platform (`-O3 -march=pentiumiii -axK -tpp7`). To estimate memory bandwidth consumption we used the `perfctr` library for manipulation of the CPU performance counters, as described in greater detail in Section 5.

Initially, we try to quantify the effects of memory bus saturation. In order to isolate the impact of memory contention from that of network contention, we confine our experiments to only one of the SMP nodes. Also, this removes the possibility of uncoordinated context switching between the local OS schedulers executing on different SMP nodes, which can influence the execution time of the workloads.

We ran six different sets of experiments (left part of Fig. 2). First, for each benchmark we ran one instance of it, in a single process on one of the two processors. In this case, the process runs with negligible interference, since it is essentially the only runnable one in the system. This run allows us to produce a good estimate of the required memory bandwidth for each benchmark, as depicted in Fig. 1.

We also use the membench microbenchmark, along with the application benchmarks. membench allocates a block of B words, then accesses it sequentially multiple times in an unrolled loop, with variable stride s. By manipulating s we can vary its cache hit ratio and the memory bus bandwidth it consumes. If L is the cache line size in words, then two extremes are possible: If $s = 1$ then the first reference to a cache line is a cache miss and all $L-1$ remaining accesses are hits. In this setup (let us call it membench-min), the microbenchmark exhibits excellent locality of reference. If $s = L$ then *all* accesses reference a different cache line (membench-max). If B is much larger than the L2 cache, membench-max causes back-to-back transfers of whole cache lines from main memory, and can lead to saturation of the memory bus.

The first set of experiments shows that our benchmarks have quite diverse demands for memory bus bandwidth. CG, MG, and SP pose heavy load on the memory subsystem; their memory access pattern involves irregular, sparse array accesses, which exhibit low cache locality and create a large number of requests to be serviced by main memory. IS, FT and BT have medium memory bandwidth requirements, while EP is in the lower end, needing about 160MB/s on our platform.

For the second set of experiments, we ran two instances of every benchmark, on one process each, in parallel. There is no processor sharing involved, however the applications suffer a significant performance hit. The more memory intensive benchmarks experience slowdowns in the range of 84-96%. The two instances of CG are effectively serialized, because of interference on the shared memory bus. We should also note that the degree of slowdown decreases along with the memory bandwidth requirements of applications.

In the next set of experiments, we examine the behavior of the benchmarks when the memory bus is already saturated, by running an instance of membench-max along with an instance of each benchmark. Again, CG, MG, SP slow down considerably, this time however IS, BT and FT are also affected significantly, running 80.4%, 42.9%, 57.4% slower respectively.

The next three sets of experiments put the Myrinet NIC into play, demonstrating that the NIC of a modern cluster interconnect can consume a major portion of the available memory bandwidth and its interference must be taken into account when studying the memory performance of clusters of SMPs. The previous three sets were repeated, this time however the firmware on the NIC was programmed to perform read/write DMA accesses to RAM. There is no change in the CPU time made available to the workload, since the DMA engine is controlled by the firmware with-

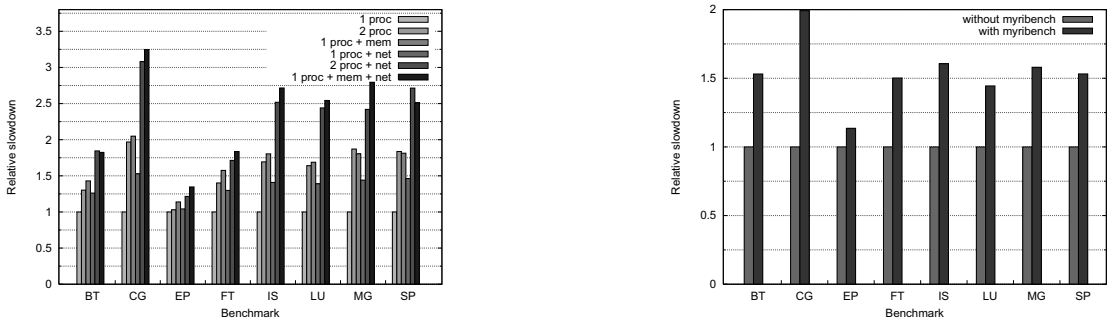

Figure 2. Application slowdown due to memory (left) and network contention (right)

out any CPU involvement. The results of the experiments indicate that the memory bandwidth available to the CPUs is decreased significantly. NIC accesses interfere with access to main memory by the CPUs, even when only one CPU is executing application code. In fact, the slowdown for this case is comparable to the slowdown caused by both CPUs executing application code simultaneously.

The previous experiments aimed at demonstrating the impact of contention on the memory bus. The objective of the next experiment is to quantify the contention for access to NIC resources that takes place in a cluster of SMPs. This time, we run an instance of each benchmark along with an instance of `myribench`, which is designed to stress-test the Myrinet interconnect by employing all-to-all communication between all hosts on the network. `myribench` runs on four processes, one on each node. The results are presented in the right part of Fig. 2. We note that network contention can indeed slowdown the execution of most of our benchmarks, in the case of `CG` even by a factor of 2.

4 Memory and network bandwidth aware scheduling policy

The experimental results of the previous section clearly demonstrate the impact of memory and network contention on the achievable system throughput on a cluster of SMPs. We should note however, that not all applications behave the same in the presence of memory or network contention. In fact, it is the the more memory-intensive applications (`MG`, `CG`, `SP`) that suffer the greatest slowdown when the shared memory bus is saturated. Similarly, applications performing fine-grained communication are more likely to suffer slowdowns when network latencies increase and the available network bandwidth decreases due to contention on the interconnect.

Motivated by these observations, we design a scheduling policy aimed at maximizing system throughput when executing computation-intensive multiprogrammed workloads on clusters of SMPs. Its main principle is that resource contention between processors in an SMP node can be avoided by careful selection of processes to be executed. A multiprogrammed workload will likely be diverse in the resources demanded. Thus, we can schedule applications with low resource demands simultaneously with applications with high resource demands. This reduces the amount of serialization that takes place on the shared memory bus, and during message exchange via the NIC. This way, a high degree of parallelism is sustained.

Our algorithm accepts as input a set of applications to be scheduled on a cluster of SMPs. Every application comprises a set of related processes. A *process* is the schedulable entity, as seen from the point of view of the underlying OS, i.e. the Linux kernel. In the generic case, each application runs on processes across multiple nodes.

The algorithm is designed to execute locally on each SMP node of the cluster. It assumes a real-time monitoring framework, which allows local schedulers to sample profiling data on currently executing processes, regarding their use of memory bandwidth and network resources. This framework is described in more detail in Section 5.

Since it is based on a gang scheduling policy, processors across all nodes of the cluster are to be allocated to all related application threads simultaneously, ensuring that process skew is minimized and the application makes progress at the fastest possible rate. Thus, the local schedulers need to communicate periodically, in order to make scheduling decisions and perform synchronous context switches.

The algorithm accepts as input a set of n applications to be scheduled. The size of the cluster is s SMP nodes, each one of them containing P CPUs. Each application i, where $0 \leq i \leq n$, comprises p_i processes on each SMP node, with $p_i < P$. Every job is characterized by a global *Job ID*. Also, the schedulers keep track of the local Process IDs associated with each application. This information is organized in a doubly linked list and stored locally.

Time is divided in quanta of q sec. At the end of every time quantum, the algorithm must decide on the set of applications to be scheduled for execution in the duration of the next one. This is a two phase process. The first phase

runs on all SMP nodes in parallel and determines candidates for execution among the applications. The second phase is a reduction phase in which the selection of applications to be scheduled is finalized.

The first, local phase of the algorithm is displayed as Algorithm 1. Before every time quantum, a local set of candidates to be scheduled is compiled, by allocating the P local CPUs to applications. As it does so, each local scheduler keeps track of the memory and network bandwidth that has not been allocated so far, denoted by BW^M_{rem} and BW^N_{rem} respectively. Initially, the available memory bandwidth is M while the available network bandwidth is N. These figures are architecture-specific and can be derived from microbenchmarking (e.g. as in Section 3).

The procedure of compiling the local set of applications is presented in Algorithm 2 and works as follows: Initially, the set is empty. The first application on the list is always added to it, and is allocated the number of processors it requires, in order to prevent processor starvation and ensure that every application will have a chance to execute. Following that, the algorithm continues selecting applications and adding them to the set, until all processors have been allocated. Applications are selected based on a heuristic which estimates how well the resource demands of the application fit the remaining memory and network bandwidth.

Since it is impossible to know the memory and network bandwidth demands of the application beforehand, a prediction is made based on the bandwidth consumption of its processes during the w previous time quanta. The notation $BW^{C,-k}_{ij}$ denotes the memory bandwidth consumed in the k-th previous time quantum by the CPU on which the j-th process of application i executed. Similarly, $BW^{N,-k}_{ij}$ is used to denote the network bandwidth consumed, when the NIC is transferring data on behalf of the same process. The total memory bandwidth consumption of this process is thus

$$BW^{M,-k}_{ij} = BW^{C,-k}_{ij} + BW^{N,-k}_{ij}$$

since all data to be exchanged over the interconnect must be transferred from/to main memory using DMA. The runtime parameter w defines a sliding window over which the memory and network access rates of every process are averaged. The choice of w could affect the efficiency of our scheduler: It needs to be large enough to smooth out very short spikes of activity, while at the same time being small enough for our scheduler to adapt quickly to changes in the behavior of processes.

The heuristic for selecting candidates for execution tries to balance the load both on the shared memory bus and the local NIC, by ensuring that they are neither saturated nor underutilized. It divides the remaining memory and network bandwidth among the yet un-allocated processors (BW^M_{rem}/P_{rem} and BW^N_{rem}/P_{rem} respectively). The most fitting application for execution is considered to be the one whose resource consumption per process best matches those two values. If we visualize the bandwidth requirements of applications in a 2D space, with the memory and network bandwidth on the horizontal and the vertical axis respectively, we can see that a good (inverse) fitness heuristic can be the Euclidean distance between the ideal application characteristics, at point

$$\left(\frac{BW^M_{rem}}{P_{rem}}, \frac{BW^N_{rem}}{P_{rem}}\right)$$

and the estimated requirements of an application (Fig. 3).

The use of this metric favors scheduling together jobs with high bandwidth demands along with lower resource demanding jobs. Once demanding applications have been selected, the ratios BW^M_{rem}/P_{rem} and BW^N_{rem}/P_{rem} become very low, so jobs with low demands are more likely to be selected next. The inverse also holds.

In the second phase of the scheduling algorithm (see Algorithm 1), a reduction operation takes place. The result of the local election of candidate applications on node l is array $S_l[]$, where element $S_l[i]$ is 1 if application i has been selected as a candidate for execution on this node and 0 otherwise. In the reduction phase the array $S_{total} = \sum_{l=1}^{s} S_l$ is computed and sorted in decreasing order. In the sequel, local schedulers allocate processors to applications according to their placement in it (i.e. their popularity), until all processors have been allocated.The reduction phase is necessary, in order to have a representative view on the resource requirements of applications; it is possible that an application might distribute the load unevenly between its processes, so the profiling data could differ significantly among SMP nodes.

The algorithm makes the assumption that applications run across all cluster nodes. This was done to simplify the analysis, by not introducing the concept of space sharing in the discussion, and to keep the emphasis on the effects of resource contention. But there is also one more reason why it makes sense to employ process topologies spread across the cluster, rather than packing processes belonging to the same parallel job as close together as possible: Running a parallel job of $2N$ processes in a $2N \times 1$ configuration of $2N$ nodes, rather than on a $N \times 2$ configuration of N nodes, means there is double the aggregate memory bandwidth available. As an example, on our experimental platform, a `CG` job runs 91% faster on the former configuration, and an `SP` job runs 67% faster.

5 Implementation details

5.1 Scheduler implementation

The scheduling policy described in the previous section is well suited to an implementation in userspace. According

$$distance\,(i) = \sqrt{\left(\frac{1}{w}\frac{1}{p_i}\sum_{k=1}^{w}\sum_{j=1}^{p_i} BW_{ij}^{M,-k} - \frac{BW_{rem}^{M}}{P_{rem}}\right)^2 + \left(\frac{1}{w}\frac{1}{p_i}\sum_{k=1}^{w}\sum_{j=1}^{p_i} BW_{ij}^{N,-k} - \frac{BW_{rem}^{N}}{P_{rem}}\right)^2}$$

Figure 3. The heuristic to determine application fitness

Algorithm 1: reschedule_at_alarm

```
1  begin
2      /* 1st phase, local decision */
3      foreach executing application do
4          stop all of its processes
5          move it at the end of the app. list
6      end foreach
7      invoke performance monitoring framework
8      select set of candidates for execution in array S_l[]
9      /* 2nd phase, reduction */
10     sum up all local arrays to S_total[]
11     sort array S_total[] in decreasing order
12     perform a barrier to sync the context switch
13     set a timer to expire after q sec
14     while there are unallocated processors do
15         resume applications based on position in
           sorted S_total[]
16     end while
17 end
```

Algorithm 2: select_applications_for_next_tq

```
1  begin
2      foreach application i in list do
3          S_l[i] ← 0
4      end foreach
5      /* The first application always gets scheduled */
6      S_l[0] ← 1; P_rem ← P − p_0
7      BW^M_rem ← M − BW^M_0; BW^N_rem ← N − BW^N_0
8      while P_rem > 0 do
9          bestd ← +∞; best ← −1
10         foreach application i in list do
11             d_i ←
               distance(i, P_rem, BW^M_rem, BW^N_rem)
12             if p_i ≤ P_rem ∧ d_i < bestd do
13                 bestd ← d_i; best ← i
14             end if
15         end foreach
16         S_l[best] ← 1; P_rem ← P_rem − p_best
17         BW^M_rem ← BW^M_rem − BW^M_best
18         BW^N_rem ← BW^N_rem − BW^N_best
19     end while
20 end
```

to this approach, scheduling decisions are made by a process running with elevated privileges and are enforced using standard OS signaling mechanisms in order to suspend and resume the applications being managed. The userspace scheduler ensures that the number of runnable processes at every instant does not exceed the number of available CPUs, so that no context switching is done by the kernelspace OS scheduler and there is no time-sharing involved.

Implementing our scheduling policy in userspace offers several advantages: A userspace implementation is much simpler and less error-prone than one in kernelspace. Moreover, a userspace process can be better informed of higher-level semantic relationships between processes. While the kernel views and schedules distinct processes, using node-local data, a userlevel scheduler can cooperate with the job submission system. Thus it can take into account for example that certain processes form an MPI job and schedule them accordingly. Supporting that kind of functionality in the kernel would require the extension of existing user/kernel interfaces for process management and much greater effort. The biggest drawback of a userlevel implementation is the lack of a simple, efficient notification mechanism of state changes in managed processes, e.g. when a process leaves the CPU and blocks on I/O. In such case the CPU would be left idle, while a kernel scheduler would context switch to a different runnable process.

This section describes a proof-of-concept userspace implementation of our policy, the MEMory and network Bandwidth aware Userspace Scheduler (*MemBUS*). MemBUS is designed to run locally, on every SMP node of the cluster, in an `mbus` process. It runs as a privileged process, so as to be able to create processes belonging to different users, and listens for control requests at a local, UNIX domain socket. A small control program (`mbus_ctl`) can be used to send requests for job creation and termination, through this socket. The `mbus` process is responsible for creating application processes to be managed, and attaching to them using the `ptrace` system call. This way it can control them completely, creating and sampling performance counters to monitor their resource consumption. The standard `SIGSTOP` and `SIGCONT` signals are used to perform the context switch.

We have also developed an interface between the `mpirun`-based job submission system of MPICH-GM (the port of MPICH to run over Myrinet) and MemBUS. We modified the job initialization scripts, so that jobs are not

spawned using the standard `rsh` mechanism, but instead `mbus_ctl` is used. Each request submitted via `mbus_ctl` is tagged with a *Job ID*, so that MemBUS can treat all processes belonging to the same job as a gang. The PID of `mpirun` where the request originated is passed as the Job ID, assuming that a frontend node is used for job submission to the cluster, so that the Job IDs of two different jobs will never collide.

As described previously, the local schedulers need to communicate, in order to perform the reduction operation and context switch in a synchronous way. Currently, this is done with a simple TCP/IP based mechanism. Initially, one of the local schedulers is in charge of accepting TCP/IP connections from all the others. At every quantum, it receives the lists of preferred applications for execution (packed in XDR representation, to allow for heterogeneity between nodes in the future),and computes $S_{total}[]$. Then it signals the other schedulers, so that they context switch simultaneously. As long as the size of the cluster remains small, performing the reduction operation centrally should not be a bottleneck. However, to improve the scalability of our scheduler we plan to rewrite this part of MemBUS using MPI primitives. MPI can choose the optimal way of performing the reduction and synchronization part, possibly organizing the nodes in a tree to enhance parallelism.

5.2 Monitoring CPU memory bandwidth consumption

Since CPUs in SMPs communicate with main memory through a multi-level hierarchy of cache memories, estimating the memory bandwidth consumption of a CPU essentially means being able to monitor the bus transactions used to load and store cache lines to and from the highest level cache, that is closest to main memory.

To monitor the memory behavior of applications without needing any source code modifications we decided to use the performance monitoring feature, as provided by most modern microprocessors in the form of performance monitoring counters. These are machine-specific registers which can be assigned to monitor events in various functional units of the processor, such as the number of floating-point instructions executed, or the number of branch mispredictions. In our case, we are interested in monitoring the Data Cache Unit, and more specifically the number of bus transactions to transfer whole cache lines between the main memory and the L2 cache (cache fill or writeback operations).

There are two obstacles for performance monitoring counters to be used effectively by MemBUS. First, the instructions for performance counter manipulation are usually privileged and can only be issued from kernelspace. Second, they are a per CPU, not a per process resource. If we are to count bus transactions and other events individually per process, i.e. only when that process is executing on a (random) CPU, the counters need to be *virtualized*, similarly to the way each process has a private view of the processor's register set, although it runs in a timesharing fashion and may migrate to other processors. Thus, the OS needs to be extended, so that it sets up monitoring of a process's events before context switching into it, and samples the performance monitoring counters when its time quantum expires.

In our case, the virtual performance counter functionality was provided under Linux by the `perfctr` library [15]. `perfctr` comprises a Linux kernel module and a userspace library. The kernel module code runs in privileged kernelspace and performs functions such as programming the performance counters and sampling their values at every context switch, while the userspace library communicates with the kernel module via the system call layer and exposes a portable interface to programmers, in order to set up and monitor virtual performance counters.

5.3 Monitoring NIC bandwidth consumption

To accurately determine memory bus bandwidth usage, our scheduler needs to take into account the contention between CPUs and the NIC for access to data residing in main memory. However, the OS-bypass, User Level Networking characteristics of modern cluster interconnection architectures make it difficult to intercept the communication process and monitor the communication load in a way that is transparent to the application. The greatest part of the communication functionality now resides within the NIC and is implemented in firmware executing on embedded microprocessors onboard it. Since the OS is not in the critical path of communication, we have to make modifications to the NIC firmware and provide a method for MemBUS to access the monitoring information.

Our testbed is based on Myrinet NICs and the GM message passing API for communication between nodes. Myrinet uses point-to-point 2+2Gbps fiber optic links, and wormhole-routing crossbar switching. In contrast to conventional networking technologies such as TCP/IP, it aims at minimizing latency by offloading communication functions to an embedded RISC microprocessor, called the Lanai, thus removing the OS from the critical path.

GM [13] is the low-level message passing interface for Myrinet, providing high bandwidth, low latency ordered delivery of messages. It comprises firmware executing on the Lanai, a Linux kernel module and a userspace library. GM allows a process to exchange messages directly from userspace, by mapping part of Lanai memory (a so called *GM port*) to its virtual address space.

This is a privileged operation that is done via a system call to the GM kernel module, but afterwards the process may directly manipulate the corresponding send and receive queues in order to request DMA transfers to and from pinned userspace buffers. All protection and communication functions (memory range checking, message matching, packet (re-)transmissions), are undertaken by the Lanai.

To monitor contention on the NIC, we modified the firmware portion of GM-2, adding two 64-bit counters per GM port, which reside in Lanai memory. The value of the counters is updated by the firmware whenever a DMA transaction completes from or to host main memory, so that they reflect the total amount of data transferred in each direction. The kernel module portion of GM-2 was also extended to include a "read counters" request, which is used by the scheduler in order to periodically sample the values of these counters and copy them to userspace.

6 Experimental evaluation

To evaluate the efficiency of our scheduling policy, we experimentally compare the system throughput achieved both under our userspace implementation and when a modern SMP scheduler, the Linux 2.6 O(1) scheduler is used. We measure application throughput for a series of workloads, combining applications with both low and high resource requirements. The experimental setup is the same as described in Section 3.

To estimate the throughput sustained during workload execution we made small modifications to the benchmarks, so that they execute an infinite number of iterations and output the current iteration count periodically to an in-memory log file. The measurement script coordinating the execution of the workloads makes sure that all applications receive a signal to terminate (`SIGALRM`) when a predetermined time period (wall clock time) expires, then analyzes the log files in order to compute the attained throughput.

Our first set of experimental workloads consisted of two instances of a resource-intensive benchmark, `CG` (comprising four processes, one on each cluster node) along with two instances of each application benchmark (in the same configuration). By measuring the performance of each workload, we can observe the behavior of the Linux scheduler and MemBUS for various combinations of applications with higher (`CG`, `MG`, `SP`) or lower (`BT`, `EP`) requirements. We chose to include `CG` in all workloads of this set, in order to see how MemBUS increases the throughput of an application which is very sensitive to memory and network contention, as it is combined in workloads with applications of various requirements.

Initially, each workload was allowed to run for a predetermined duration of 10 minutes under Linux. This duration is enough to amortize any initialization costs imposed at the beginning of each benchmark's execution, so that we can be sure that the results reflect the benchmarks doing actual, useful work. All processes were spawned directly under the control of the Linux scheduler, using the `mpirun` script for job submission with no modifications. The processes were left to execute uncoordinated, without any intervention, in order to observe their behavior without applying any optimization to their execution. The results presented are averaged over 10 runs of each workload, for 10 independent 10-minute periods.

For the second set of experiments, each workload was executed under MemBUS, using the MemBUS-specific `mpirun` script. All workloads were executed with the time quantum of MemBUS set to $q = 0.5s$, which is about two or three times the time quantum of the underlying Linux scheduler (100-200ms). Higher values of q would make MemBUS too insensitive to quickly changing application behavior, while lower values would be meaningless, since they would interfere with the decisions made by the Linux scheduler and could increase the scheduling overhead considerably. All results presented in this section use a value of $w = 2$, to smooth out any short-lived spikes in resource consumption.

Finally, each benchmark was executed on its own, in order to determine its maximum throughput on our platform. This value can be compared directly to the results of the experimental evaluation after being divided in half, since the number of processes in each workload is double the number of processors.

These initial comparisons showed that MemBUS outperformed the Linux scheduler by a very large margin; there was even a five-fold increase in throughput for the `CG-EP` workload. The reason for the poor performance under Linux was that MPICH-GM, the MPICH implementation over Myrinet, polls while waiting for message completion. Thus, a process may waste CPU time waiting for synchronization with peers which may not yet have been scheduled. In order to isolate the improvement due to better management of memory and network bandwidth from the improvement due to the gang scheduling nature of our algorithm, all runs under Linux were repeated using the *blocking* mode of MPICH-GM. In this mode, a process relinquishes the CPU whenever it needs to wait for message completion. This has two distinct advantages for Linux: First, related processes are coscheduled implicitly across the cluster, since message arrivals cause the corresponding processes to be unblocked and scheduled on a CPU. second, the Linux scheduler, contrary to MemBUS, can compensate for periods during which a process has to wait for message completion, by blocking it and context switching to a different, runnable process.

The results of this part of the experimental evaluation are displayed in the first graph of Fig. 4. For every workload,

the average throughput of the two benchmark processes is normalized relatively to the ideal throughput, and the same is done for CG. The first three bars correspond to the benchmark application of the workload, and represent its throughput under the Linux scheduler, MemBUS and in the ideal case, respectively. The next three bars correspond to CG.

The results show that scheduling under MemBUS can bring significant throughput improvements to the more memory-intensive application of the workload, in our case CG, since it suffers the most when the memory bus is saturated. MemBUS pairs an instance of CG with an instance of a less demanding application, for the majority of time quanta, as is shown by execution logs. This reduces resource contention and leads to more balanced application performance. The throughput increase of CG ranges from from about 6% (for the CG-CG workload) to 44% (for the CG-IS workload). The throughput increase of the two CG instances compensates for the performance degradation of the competing benchmark instances: Under MemBUS, they are routinely co-scheduled against CG, while under Linux they probably spend a percentage of the time executing against each other.

We expect the performance improvement under MemBUS to increase, as the workloads become more diverse in terms of bandwidth requirements. So, for the second part of the experimental evaluation, we introduce the membench-min microbenchmark. Each workload consists of two instances of membench-bin, along with two instances of each benchmark. Linux does not take advantage of the excellent cache locality of membench-min, nor of the fact that it does not perform any network I/O. On the other hand, MemBUS almost eliminates memory and network contention by coscheduling each of its instances with a benchmark instance. As expected, the throughput of the benchmark instances comes close to the ideal one. CG is favored the most, with a 39.0% improvement, while BT and SP perform 24.4% and 19.9% better respectively. However, in many cases, the throughput of membench-min is higher under Linux, actually exceeding the "ideal" case, i.e. membench-min processes get more than their fair share: they perform no network I/O and are scheduled during what would be idle periods, while the benchmark processes block waiting for data to arrive over the network.

Finally, in order to observe the behavior of our scheduling policy in the presence of network contention, the third set of workloads includes two instances of myribench, along with two instances of each application benchmark. MemBUS chooses to execute the two instances of myribench at different time quanta, so as to avoid contention on the NICs. The performance improvement for myribench ranges from 4% in the case of running with EP to 25%, in the case of running with MG. The benchmarks run with reduced contention on the memory bus, since myribench has rather low demands for memory bandwidth, but with increased contention on the network.

All results presented in this section are averaged over multiple runs. In fact, application throughput under the Linux scheduler varied widely between consecutive executions of the same workload, while it remained almost constant under MemBUS. We attribute this to the cache-affine design of the Linux scheduler, which makes interprocessor migrations of processes very rare. If two memory intensive processes are affine to the same processor, they cannot interfere with each other, since they do not run simultaneously. On the other hand, they are very likely to contend for access to resources if they are affine to different processors. Thus, system throughput under Linux depends significantly on the initial allocation to processors, which is an unpredictable, essentially random process.

7 Conclusions - Future work

Contention among processors for shared resources in SMP systems can greatly limit their efficiency. Saturation of the main memory bus and contention for access to the interconnection link reduces the attainable degree of parallelism and imposes large execution slowdowns on multiprogrammed workloads running on SMP clusters.

Motivated by these observations, we introduced a performance monitoring framework, which allows for realtime monitoring of CPU and NIC bandwidth consumption, then used it to implement a bandwidth aware scheduling policy. Experimental comparison between our scheduler and the Linux 2.6 scheduler showed a significant reduction in the CPU time required by high bandwidth processes, leading to significant increase in system throughput, as well as more predictable execution times. We expect the performance improvement to increase with the number of processors and nodes in the system.

In the future, we will continue in two directions. We plan to extend our algorithm so that it can be applied to shared resources apart from the main memory bus and the NIC of SMPs. The functional units and shared levels of the cache hierarchy in the case of SMT processors are one such example. Also, we plan to investigate moving part of our scheduler implementation to kernelspace, so that it can be extended to manage workloads featuring not only computation-intensive but also I/O-intensive applications.

References

[1] D. Agarwal and D. Yeung. Exploiting Application-Level Information to Reduce Memory Bandwidth Consumption. In *Proceedings of the 4th Workshop on Complexity-Effective Design, held in conjunction with the 30th International Symposium on Computer Architecture (ISCA-30)*, Jun 2003.

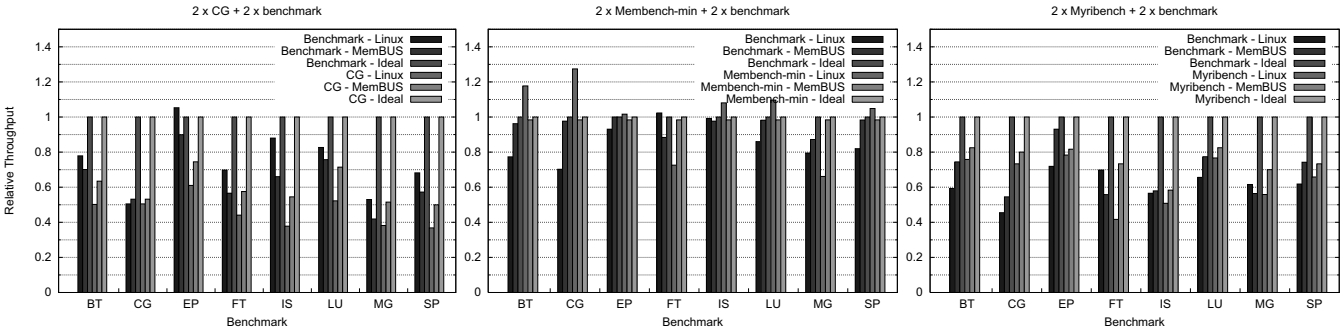

Figure 4. Comparison of application throughput achieved for various workloads under Linux and MemBUS

[2] C. D. Antonopoulos, D. S. Nikolopoulos, and T. S. Papatheodorou. Scheduling Algorithms with Bus Bandwidth Considerations for SMPs. In *Proceedings of the 2003 International Conference on Parallel Processing (ICPP 2003)*, page 547, Oct 2003.

[3] I. T. Association. InfiniBand Architecture Specification, Release 1.0, 2000. http://www.infinibandta.org/specs.

[4] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[5] F. Bellosa. Process Cruise Control: Throttling Memory Access in a Soft Real-Time Environment. Technical Report TR-I4-02-97, IMMD IV - Department of Computer Science, University of Erlangen-Nürnberg, Jul 1997.

[6] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, Feb 1995.

[7] D. G. Feitelson and L. Rudolph. Gang Scheduling Performance Benefits for Fine-Grain Synchronization. *Journal of Parallel and Distributed Computing*, 16(4):306–318, Dec 1992.

[8] A. Hori, H. Tezuka, Y. Ishikawa, N. Soda, H. Konaka, and M. Maeda. Implementation of Gang-Scheduling on Workstation Cluster. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 126–139. Springer-Verlag, 1996.

[9] M. A. Jette. Performance Characteristics of Gang Scheduling in Multiprogrammed Environments. In *Proceedings of the 1998 IEEE/ACM Supercomputing Conference on High Performance Networking and Computing (SC98)*, San Jose, California, Nov 1997.

[10] E. Koukis and N. Koziris. Memory Bandwidth Aware Scheduling for SMP Cluster Nodes. In *Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP'05)*, pages 187–196, 2005.

[11] J. Liedtke, M. Völp, and K. Elphinstone. Preliminary Thoughts on Memory-Bus Scheduling. In *Proceedings of the 9th workshop on ACM SIGOPS European workshop*, pages 207–210. ACM Press, 2000.

[12] J. D. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Technical Committee on Computer Architecture (TCCA) Newsletter*, Dec 1995.

[13] Myricom. GM: A Message-Passing System for Myrinet Networks, 2003. http://www.myri.com/scs/GM-2/doc/html/.

[14] D. Patterson and J.Hennessy. *Computer Architecture. A Quantitative Approach*, pages 373–504. Morgan Kaufmann Pub., San Francisco, CA, 3rd edition, 2002.

[15] M. Pettersson. The Perfctr Linux Performance Monitoring Counters Driver, 2004. http://user.it.uu.se/ mikpe/linux/perfctr/.

[16] M. S. Squillante and E. D. Lazowska. Using Processor-Cache Affinity Information in Shared-Memory Multiprocessor Scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):131–143, 1993.

[17] G. E. Suh, S. Devadas, and L. Rudolph. A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning. In *Proceedings of the Eighth International Symposium on High-Performance Computer Architecture (HPCA'02)*, pages 117–, 2002.

[18] G. E. Suh, L. Rudolph, and S. Devadas. Effects of Memory Performance on Parallel Job Scheduling. *Lecture Notes in Computer Science*, 2221:116–, 2001.

[19] J. Torrellas, A. Tucker, and A. Gupta. Evaluating the Performance of Cache-Affinity Scheduling in Shared-Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 24(2):139–151, 1995.

[20] R. Vaswani and J. Zahorjan. The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed, Shared Memory Multiprocessors. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, pages 26–40. ACM Press, 1991.

[21] M. Zagha, B. Larson, S. Turner, and M. Itzkowitz. Performance Analysis using the MIPS R10000 Performance Counters. In *Proceedings of the 1996 ACM/IEEE Supercomputing Conference on High Performance Networking and Computing (SC96)*, page 16. ACM Press, Nov 1996.