Using Overdecomposition to Overlap Communication Latencies with Computation and Take Advantage of SMT Processors

Lars Ailo Bongo[1], Brian Vinter[2], Otto J. Anshus[1],
Tore Larsen[1] and John Markus Bjørndalen[1]
[1)] Department of Computer Science, University of Tromsø, Norway
{larsab, otto, tore, johnm}@cs.uit.no
[2)] DIKU, University of Copenhagen, Denmark
vinter@diku.dk

Abstract

Parallel programs running on clusters are typically decomposed and mapped to run with one thread per processor each working on its disjoint subset of the data. We evaluate performance improvements and limitations for a micro-benchmark and the NAS benchmarks, by using overdecomposition to map multiple threads to each processor to overlap computation with communication. The experiment platform is a cluster with Pentium 4 symmetric multithreading (SMT) processor nodes interconnected through Gigabit Ethernet. Micro-benchmark results demonstrate execution time improvements up to 1.8. However, for the NAS benchmarks overdecomposition and SMT provides only slight performance gains, and sometimes significant performance loss. We evaluated improvement and limitation sensitivity to problem size, communication structure and whether SMT is enabled or not. We found that performance improvements are limited by: applications having communication dependencies that limit thread-level parallelism, increase in cache misses, or increased systems activity. Our study contributes a better understanding of these limitations.

1 Introduction

In this paper we investigate when and how overdecomposition may be applied to improve performance without any changes to source-code for MPI-based [17] parallel scientific applications running on clusters of simultaneous multithreading (SMT) enabled single-processor Pentium 4 nodes interconnected through low-cost Gigabit Ethernet.

As shown in figure 1, scientific parallel applications are typically decomposed such that one processor in the cluster runs one thread for a disjoint subset of the data.

Increasing the decomposition of the data will increase the number of threads and may allow for overlapping computation with communication to improve single-application performance. However, increasing the decomposition will typically also increase the number of messages exchanged and the latencies and other costs associated with those message transfers. Our goal is to identify when and how we may increase the decomposition to achieve the performance benefits of overlapping computation and communication while not incurring communication costs that alleviate the increased performance.

The paper makes three contributions:

- We provide an experimental evaluation of the performance benefits of overdecomposition for parallel application with a wide range of communication characteristics.
- We also provide an experimental evaluation of the benefit of SMT for parallel applications implemented using MPI.
- We provide insight into system software issues that effect overdecomposition improvements by describing and using an analysis methodology that combines message traces, operating system counters and hardware performance counters.

2 Experiment setup

2.1 Hardware platform

All experiments were run on a cluster of 44 nodes interconnected over Gigabit Ethernet. Each node is a single processor system with 2 GB RAM and local disk. The processor used is a 90 nm 3.2 GHz version of the Intel Pentium 4. This is an SMT processor applying the second iteration

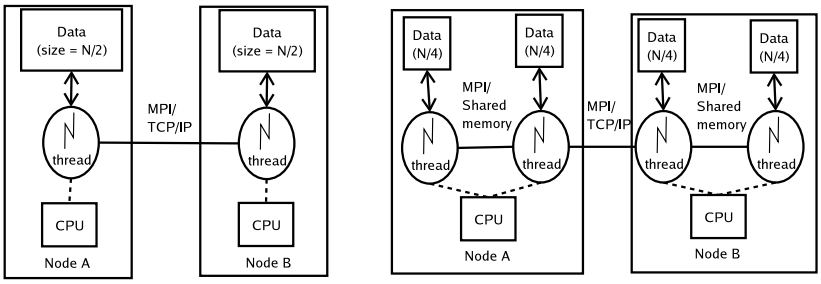

Figure 1. A parallel application without (left), and with overdecomposition (right).

of Intel's Hyperthreading (HT) Technology [1] which offers several improvements over previous implementations in terms of increased or enhanced resources and more dynamic resource allocation.

Each processor has a 12 KB L1 execution trace cache for microoperations, 16 KB 8-way L1 data cache, and a 1 MB 8-way unified L2 cache. Memory access latencies measured using Cachebench [13] are: L1 data: 1.25 ns, L2 unified: 8.78 ns, and main memory: 36.6 ns.

2.2 Software platforms

The cluster nodes run the Linux 2.4.18 uni-processor kernel for the experiments where SMT is disabled, and Linux 2.4.18smp or 2.6.9smp for the SMT experiments. The 2.4 kernel was the first Linux kernel with explicit support for Intel HT Technology. The 2.6 kernel further improve the handling of HT.

The Native POSIX Thread Library (NPTL) [5] was used. NPTL synchronization variables are implemented using the fast user-space locking system call (futex) which handles any non-contended case without requiring a system call.

The communication runtime system used was LAM/MPI version 7.1.1 [11]. LAM/MPI supports hierarchy aware collective operation and shared memory intra-node communication. But when applying overdecomposition multiple processes must be used. For the SOR experiments PastSet [25] was used instead of LAM/MPI. PastSet differs from LAM/MPI in that it supports multi-threading, buffers are explicitly allocated, the communication system has helper threads, and the same protocol is used for all message sizes.

2.3 Benchmarks

The successive over-relaxation (SOR) kernel was chosen under the assumption that the latency of its blocking point-to-point communication operations can easily be overlapped with computation. The benchmark is run for three problem sizes: large, medium and small. For these communication operations contribute to respectively 25%, 50% and 75% of the execution time, when run on 32 nodes with SMT disabled. SOR is compiled with gcc 3.2.3.

Benchmark	Messages	Coll.	Asynch.
BT	Many small	No	Yes
CG	Many small, few large	Manual	Yes
EP	Few small	Yes	No
FT	Few large	Yes	No
IS	Few large	Yes	No
LU	Many small	No	No
MG	Many medium	No	Yes
SP	Many medium	No	Yes

Table 1. NAS benchmark communication behavior. Small message is less than 1 KB, large more than 1 MB. Yes for collectives if execution time is dominated by them. Yes for asynchronous if asynchronous operations are used.

The NAS benchmarks [18] are widely used to evaluate different aspects of parallel architectures. They represent a variety of communication behaviors as shown in table 1. We use the NAS 2.4 MPI implementation with the class B and C problem sizes. The benchmarks were compiled using the Intel Fortran 8.1, and Intel C++ 8.1 compilers.

2.4 Data collection

PAPI [4] is used to access the Intel Pentium performance counters. The Linux kernel is patched with *perfctr* 2.6.9 to provide *virtual performance counters*. These are per-thread counters that increase only when the thread runs user level code. Since this release of *perfctr* lacks SMT support, we have no hardware counter data for the SMT experiments.

Linux maintains process statistics including user level time and system level time per thread, and idle and interrupt handling time per processor context.

For runtime monitoring we use runtime statistical pro-

filing (as described in the next section). For the SOR experiments we use EventSpace [2] to collect message traces for post-mortem analysis. EventSpace allows us to record timestamps inside the communication system, such as before and after writing a message to a buffer.

2.4.1 Overhead

For SOR, the overhead for reading the OS resource counters was less than the variation in execution time.

Message tracing overhead depends on the communication characteristics of the application. For our experiments, the overhead is typically in the 0–4% range. The PAPI overhead due to the in-kernel collection of data is in the 0–2% range. The perturbation introduced by the data collection may influence which mappings shows best performance, favoring mappings with fewer threads per processor. Similarly, execution time improvements due to overdecomposition may also be negatively affected. Still, we believe the data collected demonstrates important trends such as reduction in idle time and increased overhead.

3 Analysis methodology

We characterize each benchmark by: (i) thread-level parallelism (TLP): number of threads ready to run (or running) application computation code, (ii) memory-wait: time the processor is stalled due to cache misses, (iii) system overhead: number of cycles used for running operating system code, (iv) communication overhead: number of cycles for communication activity, (v) network-wait: time waiting due to network latency, and (vi) synchronization-wait: time waiting for data arrival or thread synchronization.

TLP is estimated from the thread count by subtracting the number threads blocked on communication, assuming the remaining threads are compute ready. To characterize the distribution of TLP over a benchmark run we define TLP_N as the ratio of execution time where TLP is larger than or equal to N. Thus, TLP_1 is the percentage of execution time when at least one thread was, our could have been, computing. Without operating system instrumentation we cannot distinguish between these two states.

Memory wait is calculated based on the recorded number of cache misses and the miss penalties determined previously using Cachebench.

System overhead includes operating system activity for inter-node communication, synchronization overhead, context switches, and TLB misses. System time statistics are maintained by Linux.

Communication overhead was typically either to small to be significant, or accounted for elsewhere. The main sources are thread synchronization and memory copying. Both are already accounted for respectively as system overhead and memory wait.

Threads:processor	2:1	4:1	8:1	16:1	32:1
Idle	1435	1565	1417	1644	976
System activity	70	130	250	500	1040
Memory wait	226	603	1492	3147	6576
TLB wait	10	20	41	81	165
Unknown	97	163	483	958	1991
in % of exec	1.0%	2.0%	5.7%	10.3%	18.9%

Table 2. Breakdown of SOR overhead increases relative to the one thread per processor mapping. The measurements are for the medium problem size run on 32 nodes with SMT disabled. Unknown is the difference between estimated and measured execution time reduction. All times are in ms.

Network wait, the time between sending a request and receiving a response, excluding request processing time on the other node, and *synchronization wait*, the time between a receive operation blocked until a send is initiated, are determined from message traces. Wait time at synchronization points is calculated as described in [3].

3.1 TLP and overhead variation

During our analysis we assume that the metrics are similar on all nodes if the benchmark is load balanced. Using the SOR benchmark we measured the variation for the calculated metrics for SOR run on 32 nodes with the medium problem size. The benchmark was run five times.

TLP and data cache miss averages are similar for all nodes for all runs, with standard deviations less than 5% of mean. L1-instruction cache misses and system time have more variation (standard deviation is about 10% of mean).

SOR has non-deterministic waiting pattern where most nodes waits for other nodes, due to a small load imbalance in the communication workload since two of the nodes only have one neighbor. Therefore the variation is large for network wait and synchronization wait (and hence idle time). Which nodes have large synchronization wait change when rerunning an experiment, while network wait is similar for all runs. We believe we still can use the average synchronization wait time for all cluster nodes in the analysis, since the average has less variation.

3.2 Overhead accuracy

The overhead metrics combine data from several sources and abstraction levels. Also, we make several simplifications for system behavior. Here, we evaluate the accuracy

of the estimated overheads. In addition we have verified that TLP results correspond with idle ratio statistics collected by the operating system.

Subtracting all overheads from the reduction in idle time should give the reduction in execution time. The sum of overheads is usually overestimated (table 2). There are two sources of error. First the memory miss penalty is too large. Probably since overlapped cache misses are not taken into account. For SOR increasing the number of threads increases the number of cache misses and hence the miss penalty overestimation. If we assume computation time does not increase, then we can find the overestimation by comparing the sum of memory wait and user level time. The second source of error is system time which is too high for frequently communicating benchmarks.

3.3 Runtime monitor implementation

Our MPI runtime monitor intercepts all communication operations. Statistics about operation times and TLP are updated for each operation. In addition OS statistics and PAPI counters are read at selected collective operations (usually when calling MPI_Init and MPI_Finalize).

Since we do not have any tracing inside the communication system we cannot distinguish between network wait and synchronization wait. TLP counters are in a shared memory map, and these are updated before and after calling a blocking communication operation.

Usually metrics results are presented as statistics over of all nodes over all iterations. But for applications with load balance problems per node statistics are useful. Similarly for applications with several phases, per phase statistics should be used.

4 Performance improvement

Results from running the benchmarks with one thread (or process) per processor with SMT disabled provides insight into which benchmarks have communication wait that can be overlapped with computation. We do similar experiment with SMT enabled, to verify that SMT does not slow down the benchmarks.

We measure overdecomposition execution time improvements with SMT disabled to get insight into the degree of overlapping, and overhead increase we can achieve when threads are not run in parallel on a processor. Then we enable SMT to measure how TLP and the overheads increase when threads can run and compete for resources in parallel.

For all experiments we first analyze the simpler SOR benchmark, before analyzing how the different communication behavior of the NAS benchmarks influence the results. All experiments are repeated ten times and the mean is reported. The standard deviation for the execution times was low if not otherwise noticed, usually less than 2% of mean.

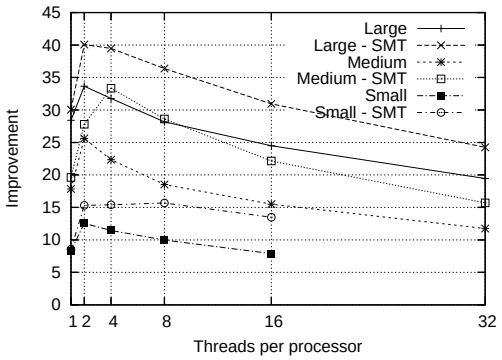

Figure 2. SOR execution time improvements relative to sequential code.

4.1 Baseline

For problem constrained scaling with SMT disabled, execution time is reduced for SOR for all three problem sizes when increasing the number of nodes from 1 to 44. Similarly, execution time is reduced for all NAS benchmarks with both problem sizes when increasing the number of nodes from 1 to 32 or 36 (BT and SP can only be run with a square number of processes). For the remaining experiments we use either 32, 36, or 44 nodes.

The SOR problem sizes were chosen such that 25%, 50% and 75% of the execution time is spent blocked in communication operations. For these respectively 20%, 40% and 55% is due to network latency, the remaining is for synchronization wait.

For most NAS benchmarks wait operations, collective operations or blocking receiving operations contribute significantly to the execution time (table 3).

In conclusion, all benchmarks scale to the cluster size used, and most have operations that can partially or totally be overlapped with computation by using overdecomposition.

4.2 Overdecomposition

SOR was run with 2, 4, 8, 16 and 32 threads per processor (below we use 2:1 when referring to a mapping with two threads per processor core). Execution time improves compared to the one thread per processor mapping for all problem sizes (figure 2). The large problem size has best parallel efficiency, but the relative reduction in execution time is largest for the small problem size (1.5). The best mappings have few threads; 2:1 with SMT disabled. The

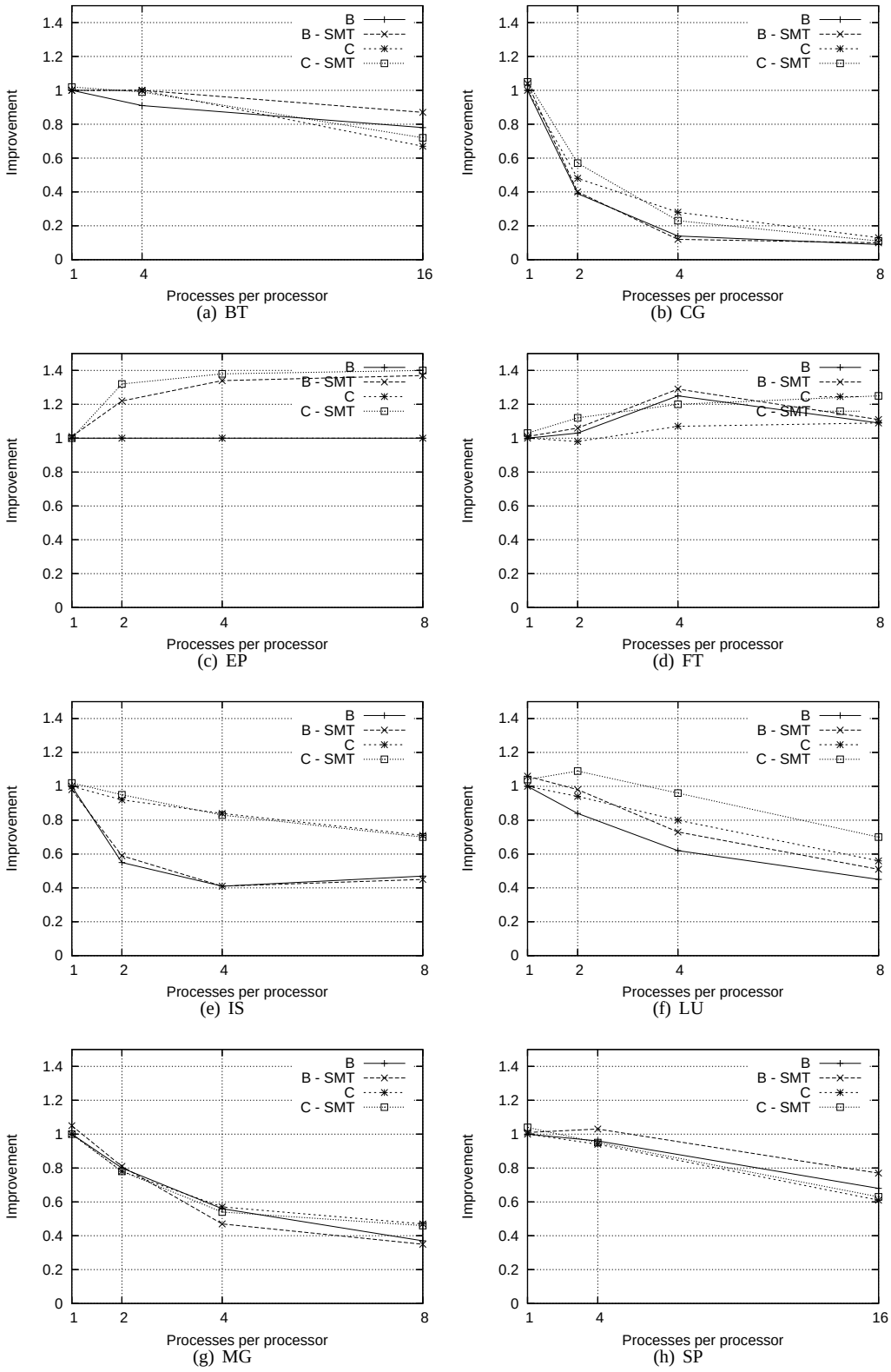

Figure 3. NAS benchmark execution time improvements relative to one thread per processor mapping. Experiments were run on 32 or 36 (BT and SP) nodes.

Benchmark	class B	class C
BT	wait (54%), waitall (10%)	wait (32%), waitall (8%)
CG	wait (53%), send (24%)	send (40%), wait (32%)
EP	none	none
FT	alltoall (62%)	alltoall (53%)
IS	alltoallv (54%), allreduce (29%)	alltoallv (47%), allreduce (20%)
LU	recv (12%), send (10%), wait (7%)	recv (9%), send (8%), wait (4%)
MG	wait (24%), send (16%)	send (18%), wait (8%)
SP	waitall (80%)	waitall (60%)

Table 3. MPI operations contributing to more than 4% of the execution time.

results shows that overdecomposition can improve application performance even on uni-processors.

Figure 3 shows that with SMT disabled overdecomposition improves performance significantly only for FT for both class B and class C. However, performance decreased for for CG, IS and MG.

4.3 Overdecomposition with SMT

Enabling SMT does not change 1:1 mapping execution time, but the improvements with overdecomposition are better. For SOR the best improvement is 1.81 compared to the 1:1 mapping. For the large problem size the parallel efficiency is improved from 30 to 40 (figure 2). The best performance is for mappings with more threads than processor contexts (four threads per 2-way SMT core).

Figure 3 shows that for the NAS benchmarks, enabling SMT gives performance improvement for EP, FT, LU and SP (only for class B). For BT and LU performance was unchanged, while CG and IS got a significant slowdown. For most experiments overdecomposition had best performance with two processes per processor.

The benchmarks for which performance improves have: (i) few and small messages, (ii) few large collective operation messages, and (iii) many small blocking point-to-point messages.

Performance is either not changed or decreased for benchmarks with many asynchronous point-to-point operations with medium or small sized messages. The IS benchmark has two execution phases with almost all communication taking place in the second phase, as well as a global synchronize operation between the phases preventing any overlap.

In conclusion, applying overdecomposition demonstrates a potential performance gain for some application characteristics, but should not be applied indiscriminately as it may result in unchanged or reduced performance for other applications. The mappings with best performance have few threads per processor, but some have multiple threads per processor context. Also, the best performance improvements are for problem sizes where more than 50% of the 1:1 execution time is due to communication.

5 Performance limitations

In this section we analyze how many threads are running at the same time, and which overheads increase most for the different benchmarks. Finally, we measure the effect of synchronization variable implementations, user-level schedulers and operating system kernels.

5.1 Thread level parallelism

When run with SMT disabled, SOR does not have enough TLP_1 to fully utilize the single processor context even with 32 threads per processor. The TLP limitation is not due to system code using the processor, since with more than four threads the idle ratio increases. Rather the limitation is due to data dependencies in the application and scheduling policies in the system software.

Enabling SMT improves TLP_1 for SOR, but still TLP_1 decrease when there are too many threads per processor. Also, when the problem size gets smaller the ratio of execution time where at least two threads are runnable decreases. Often it can be as low as 5%, even for configurations with 32 threads.

With SMT disabled, increasing the number of processes per processor does not always increase TLP_1 for the NAS benchmarks. For EP and MG the processor is saturated, but for the other benchmarks processor utilization is usually less than 76% (table 4).

Enabling SMT may increase TLP_1 with a few percentages. Also, as shown in table 5 TLP_2 is low for BT, CG, LU and SP.

5.2 Overhead increases

For SOR, all overheads increase. The increase in data cache misses is most significant for the medium problem

Benchmark	B	C
BT	44–76%	70–67%
CG	21–7%	27–9%
EP	100-99%	100-99%
FT	32–64%	33–75%
IS	19–32%	31–42%
LU	67–38%	77–56%
MG	20–53%	68–41%
SP	23–35%	47–48%

Table 4. TLP_1 increase when the number of processes per processor is increased from 1 to 8 or 16 (BT and SP). SMT is disabled.

Benchmark	B, 2	C, 2	B, 4	C, 4
BT	12%	15%	9%	6%
CG	8%	3%	8%	3%
EP	99%	99%	97%	99%
FT	35%	64%	20%	39%
LU	5%	7%	4%	2%
MG	40%	39%	39%	37%
SP	8%	21%	6%	16%

Table 5. Maximum TPL_2 and TLP_4 for the class B and class C problem sizes (minimum is always zero). SMT is enabled (for some benchmark these numbers are higher when SMT is disabled).

size, but with the small problem size system activity becomes the most significant overhead. Also, network wait which is the overhead we are trying to overlap, increase when the processor load increase with more threads.

Enabling SMT does not increase per thread user level time or system level time for SOR. Thus, we can assume that cache miss penalties and system activity increase are similar. However, the reduction in idle time is larger, giving a larger reduction in execution time.

Table 6 shows that for most NAS benchmarks either memory wait or system activity dominate the increase in overheads. Usually, the dominating overhead does not depend on problem size, but on the process to processor ratio. With four or less processes per processor, cache miss penalty increase most. But with more processes, system activity increase more. Also, cache misses may not always increase with more processes, but system activity always increases.

Of the cache misses the largest penalty is due to L1-D or L2 caches misses. However, with class B; L1-I and TLB miss penalty may be significant.

For most benchmarks the increase in user and system

Benchmark	Class B	Class C
BT	Memory, system	System, memory
CG	Memory, system	Memory, system
EP	None	None
FT	(System)	(System)
IS	None	None
LU	Memory, system	Memory, system
MG	System	System
SP	Memory, system	Memory, system

Table 6. Significant overheads.

time is similar with and without SMT. But for CG and SP both are lower, and for MG system time increase is lower.

Table 7 summarizes which parts of the platform limits overdecomposition performance for the NAS benchmarks.

5.3 System software

Using oprofile [19] we find that most kernel samples for SOR with 1:1 mapping are for the Ethernet driver, while for 32:1 most are for synchronization and context switches. Since synchronization may cause a context switch, we cannot differentiate between these.

We evaluated system software effect on TLP and the system activity overhead using two synchronization variable implementations, two user-level schedulers, and two operating system kernels. The results are for SOR run with the medium problem size.

We replaced NPTL [5] with LinuxThreads [12], and as expected system overhead increased, due to more system calls for synchronization. However, for small messages sizes TLP improved. LinuxThreads improved TLP_2 two threads were runnable from 2% to 34%. The reduction is caused by difference in scheduling policy. With LinuxThreads, synchronization variable calls are likely to cause a context switch.

We implemented two user level schedulers in the PastSet communication system. The first attempts to reduce cache misses by only allowing one or two threads to run computation code at the same time. The second attempts to better overlap inter-node communication by reordering the computation order of the threads in one node. However, due to TLP limitations most of the time there is only one runnable thread, and hence user-level scheduling will not work.

Replacing the 2.4 SMP Linux kernel with the SMT optimized 2.6 kernel does not significantly improve TLP for SOR. Also, system overhead does not significantly change.

6 Discussion and related work

Overlapping I/O wait time with computation to achieve higher CPU utilization is a well known and widely used

Benchmark	BT	CG	EP	FT	IS	LU	MG	SP
Processor idle	Yes			Yes				Yes
Processor saturated			Yes				Yes	
Lack of TLP	Yes	Yes		Yes		Yes	Yes	
Cache misses	Yes	Yes				Yes		Yes
TLB misses								
System activity	Yes	Yes				Yes	Yes	Yes
Comm. phases					Yes			

Table 7. Overdecomposition performance limitations for the NAS benchmarks.

technique. For parallel applications overdecomposition has been described in text books [7], and has been for load balancing by running more threads on underutilized processors [7, 6], and to mask communication latency in a Grid environment [10]. To our knowledge this is the first study on overdecomposition performance improvements on Ethernet clusters with SMT processors. In [10] experiments were conducted to measure application slowdown when the WAN latency between clusters was increased. Our experiments differ in that we attempt to improve the performance of an applications run on a network with a fixed LAN latency. We have unpublished results showing that overdecomposition improvement becomes better for SOR in a WAN environment.

Early simulator results have shown that SMTs [15] can improve parallel application performance [15, 22]. However, recent studies show that SMT has best performance on the POWER5 [9] when cache performance is at its worst, and SMT is not well suited for floating-point workloads and memory bandwidth bound applications [8]; all typical characteristics of parallel scientific applications. Our results show that only four of the NAS benchmarks had significant increase in memory wait time.

A thorough study of SMT on the HT Technology enabled Pentium 4 processors used in our cluster is [23]. The average multithreaded speedup recorded is 1.20 for multithreaded workloads and 1.24 for parallel workloads running on a single node. The applications that were worst affected by running with SMT enabled were those that had the lowest instructions per cycle ratio. Another study [16] on Intel Xeon, shows speedups ranging from 1.05 to 1.28 for data-parallel numerically intensive benchmarks. Intel Xeon performance improvements for web servers were found to depend on the server design and implementation, and could get worse when enabling SMT due to more synchronization in the operating system kernel [21]. Our results for SMT improvement shows smaller improvements for our message passing parallel applications run on a cluster, than the single node shared memory applications in [23, 16]. We do not experience slowdown when using a SMP kernel rather than an uni-processor kernel.

Proposed system support for SMT includes: (i) new synchronization mechanism that permits cheaper synchronization [24], (ii) compiler optimizations including new approaches for inter-thread data-sharing, application of latency-hiding, and loop distribution [14] (iii) kernel mode behavior [20], and (iv) operating system schedulers [22] attempting to benefit from possible constructive inter-thread behavior.

Our results shows that synchronization contributes significantly to system overhead, which is the overhead that increase mostly when the number of threads increase. In addition to using more efficient hardware mechanisms, synchronization variable improvements should also attempt to improve TLP by minimizing the time between unblocking and running a thread. Due to the TLP limitations kernel mode behavior and operating system schedulers are less important, since there usually are few runnable threads. Similarly compiler optimizations and schedulers designed for minimizing competition for processor resources will probably not improve performance since our benchmarks have low TLP, in addition to being memory intensive and hence have low instructions per cycle.

Alternatives to overdecomposition are to rewrite the application to either use both message passing and shared memory, or to use asynchronous communication operations. Both increase the complexity of the parallel program.

7 Conclusion and future work

We evaluated if parallel application performance can be improved by overdecomposition the data into more pieces than there are processors in order to overlap communication operation latencies with computation and taking advantage of SMT processors.

Microbenchmark results are promising with execution time improvements up to 1.8. However, performance improved for only two NAS benchmark, and decreased for three, showing that improvements are sensitive to applications communication structure, cache miss behavior, the problem size used, and also of the underlying system components. The best results were for applications with few

blocking communication operations, and low cache miss penalty to execution time ratio.

Performance improvements are better when SMT is enabled, and never significantly worse. Hence for Pentium 4 based cluster SMT can be enabled as default. But changes to system software are necessary for fully utilizing SMT enabled processors. Especially intra-node communication must be designed to reduce system calls and cache misses, and synchronization primitives must strive to keep the number of runnable processors high.

As future work, we will investigate if the techniques described in this paper can be used with multiprogramming to overlap globally synchronizing operations with computation, without decreasing single application performance. Also, we intend to investigate system changes tailored for the NAS benchmarks for which performance did not improve.

The monitoring tool used for measuring TLP and overhead increase is available at:

http://www.cs.uit.no/~larsab/minim/

Acknowledgment

Thanks to Jon Ivar Kristiansen for help configuring the cluster.

References

[1] D. Boggs, A. Baktha, J. Hawkins, D. T. Marr, J. A. Miller, P. Roussel, R. Singhal, B. Toll, and K. S. Venkatraman. The microarchitecture of the intel pentium 4 processor on 90nm technology. *Intel Technology Journal*, 8, February 2004.

[2] L. A. Bongo, O. Anshus, and J. M. Bjørndalen. EventSpace - Exposing and observing communication behavior of parallel cluster applications. In *Euro-Par*, volume 2790 of *Lecture Notes in Computer Science*, pages 47–56. Springer, 2003.

[3] L. A. Bongo, O. Anshus, and J. M. Bjørndalen. Collective communication performance analysis within the communication system. In *Euro-Par*, volume 3149 of *Lecture Notes in Computer Science*, pages 163–172. Springer, 2004.

[4] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Proc. of the 2000 ACM/IEEE conference on Supercomputing*, 2000.

[5] U. Drepper and I. Molnar. Native POSIX thread library for linux. http://people.redhat.com/drepper/nptl-design.pdf.

[6] R. J. O. Figueiredo and J. A. B. Fortes. Impact of heterogeneity on dsm performance. In *Proc. of Sixth International Symposium on High-Performance Computer Architecture*, 2000.

[7] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors, volume Volume I: General Techniques and Regular Problems*. Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[8] IBM systems & technology group. how DB2 exploits IBM @serverp5 and AIX 5L simultaneous multithreading, October 2004. www-1.ibm.com/servers/eserver/pseries/hardware/whitepapers/p5_db2.pdf.

[9] R. N. Kalla, B. Sinharoy, and J. M. Tendler. Ibm power5 chip: A dual-core multithreaded processor. *IEEE Micro*, 24(2):40–47, 2004.

[10] G. A. Koenig and L. V. Kalé. Using message-driven objects to mask latency in grid computing applications. In *In Proc. of 19th International Parallel and Distributed Processing Symposium*. IEEE Computer Society, 2005.

[11] LAM-MPI homepage. http://www.lam-mpi.org/.

[12] X. Leroy. LinuxThreads. http://pauillac.inria.fr/~xleroy/linuxthreads/.

[13] LLCbench. http://icl.cs.utk.edu/projects/llcbench/.

[14] J. L. Lo, S. J. Eggers, H. M. Levy, S. S. Parekh, and D. M. Tullsen. Tuning compiler optimizations for simultaneous multithreading. *International Journal of Parallel Programming*, 27(6):477–503, 1999.

[15] J. L. Lo, J. S. Emer, H. M. Levy, R. L. Stamm, D. M. Tullsen, and S. J. Eggers. Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. *ACM Trans. Comput. Syst.*, 15(3):322–354, 1997.

[16] D. Marr, F. Binns, D. Hill, G. Hinton, D. Koufaty, J. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, February 2002.

[17] MPI: A Message-Passing Interface Standard. *Message Passing Interface Forum*, Mar. 1994.

[18] NASA. NAS parallel benchmarks. http://www.nas.nasa.gov/Software/NPB/.

[19] Oprofile system-wide profiler for linux. http://oprofile.sourceforge.net.

[20] J. A. Redstone, S. J. Eggers, and H. M. Levy. An analysis of operating system behavior on a simultaneous multithreaded architecture. In *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*. ACM Press, 2000.

[21] Y. Ruan, V. S. Pai, E. Nahum, and J. M. Tracey. Evaluating the impact of simultaneous multithreading on network servers using real hardware. In *SIGMETRICS '05: Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. ACM Press, 2005.

[22] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. In *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*. ACM Press, 2000.

[23] N. Tuck and D. M. Tullsen. Initial observations of the simultaneous multithreading pentium 4 processor. In *12th International Conference on Parallel Architectures and Compilation Techniques*. IEEE Computer Society, 2003.

[24] D. M. Tullsen, J. L. Lo, S. J. Eggers, and H. M. Levy. Supporting fine-grained synchronization on a simultaneous multithreading processor. In *Proc. of the Fifth International Symposium on High-Performance Computer Architecture*, 1999.

[25] B. Vinter. *PastSet a Structured Distributed Shared Memory System*. PhD thesis, Department of Computer Science, University of Tromsø, 1999.