เดียวกันกับหน่วยประมวลผลที่มากกว่า 4 หน่วย แล้วนำมาแสดงในแบบเดียวกัน เพื่อทำให้ผู้อ่านรายงานมีความมั่นใจในประสิทธิภาพของโปรแกรมที่งานวิจัยนี้สร้างขึ้นมากยิ่งขึ้น

3.4 สรุปเนื้อหางานวิจัย

งานวิจัยได้แสดงให้เห็นว่าการนำสายการทำงานมาใช้กับระบบความจำร่วมนั้นให้ประสิทธิภาพของโปรแกรมดีมากขึ้น ทั้งนี้ระบบซีวีเอ็มที่ได้รับการพัฒนาขึ้น สามารถช่วยในการออฟติไมซ์การทำงานในส่วนอื่นๆ ได้ด้วย เช่น ความสามารถในการเลี่ยงปัญหาที่มีจำนวนสายการทำงานมากเกินไป และ ปัญหาเรื่องการทำงานบนหน่วยประมวลผลที่ไม่เท่าเทียมกัน งานวิจัยนี้ยังคงมีปัญหาหลักในเรื่องที่ขาดทรัพยากรที่จะนำมาทดสอบเพื่อให้ตัวเลขผลการทดลองที่มีความถูกต้องสมบูรณ์มากขึ้น ซึ่งหัวหน้าโครงการยังคงจะทำการวิจัยต่อไป เมื่อแก้ปัญหาในจุดนี้ได้แล้ว ด้วยความหวังว่าเครื่องที่ HPCC ซึ่งมีทั้งหมด 8-16 หน่วยประมวลผล จะสามารถนำมาทดสอบได้จนครบตามที่ต้องการ เพื่อสรุปผลการทดลองและงานวิจัยที่ได้ค้นพบส่งเป็นรายงานในวารสารวิชาการต่อไป

4 ผลที่ได้จากโครงการวิจัย

1) งานตีพิมพ์ในการประชุมวิชาการระดับนานาชาติสองรายงาน
   - K. Thitikamol and P. Keleher, "Thread Scheduling and Grain Emulation", The Fourteenth IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2002), Cambridge, USA, 2002.
   - K. Thitikamol, "Thread Placement Algorithms for Load Balancing and Adaptive Parallelism in Software DSMs", The Third International Symposium on Communications and Information Technologies (ISCIT 2003), Songkha, Thailand, 2003.

2) โปรแกรม ซีวีเอ็ม ที่เป็นโปรแกรมระบบความจำร่วมแบบขนาน หลังจากที่หัวหน้าโครงการทำต่อจนเสร็จสมบูรณ์แล้วจะนำไปไว้ที่ http://www.cs.umd.edu/projects/cvm ที่เป็น website ของอาจารย์ที่ปรึกษา และจะสามารถให้ผู้สนใจนำไปทดลองวิจัยได้เฉพาะเพื่อการศึกษาและไม่แสวงหากำไรจากเทคนิคเหล่านี้

5 ภาคผนวก

5.1 คำขอบคุณ

ขอขอบคุณ เพื่อนๆ และผู้ให้ความอนุเคราะห์ ตามรายชื่อต่อไปนี้

1) รศ. เอกวิชญ์ นันทจิวรวัฒน์, สถาบันเทคโนโลยีนานาชาติสิรินธร
2) ดร. ศิษเดช ทองสิมา, ศูนย์เทคโนโลยีชีววิทยาและพันธุกรรมแห่งชาติ
3) ดร. ศรเทพ วรรณเรัตน์, ศูนย์เทคโนโลยีอิเลคทรอนิกส์และคอมพิวเตอร์แห่งชาติ
4) ดร.ปิยวุฒิ ศรีรัชกุล, ศูนย์เทคโนโลยีอิเลคทรอนิกส์และคอมพิวเตอร์แห่งชาติ
5) ผศ. ดร. ภูชงค์ อุทโยภาศ, คณะวิชาวิศวกรรมศาสตร์ มหาวิทยาลัยเกษตรศาสตร์
6) ผศ. ดร. วรา วราวิทย์, คณะวิชาวิศวกรรมศาสตร์ สถาบันเทคโนโลยีพระจอมเกล้าพระนครเหนือ

5.2 กิจกรรมที่นำเอาผลงานไปใช้

ก) แสดงสไลด์และผลงานจากแนวความคิดที่ได้จากการวิจัยนี้ ในงาน National Workshop on Cluster Computing 2003 ที่สถาบันเทคโนโลยีแห่งเอเชีย

ข) แสดงสไลด์และผลงานในหัวข้อที่เกี่ยวกับการผสมผสานเทคนิคที่ได้จากโครงการวิจัยนี้ ให้แก่นักศึกษาปริญญาโททางวิทยาศาสตร์คอมพิวเตอร์ ที่มหาวิทยาลัยมหิดล พญาไท

5.3 สำเนาเอกสารที่ได้ส่งไปร่วมการประชุมวิชาการ

(พลิก)

Thread Placement Algorithms for Load Balancing and Adaptive Parallelism in Software DSMS

Kritchalach Thitikamol*
Sirindhorn International Institute of Technology
Thammasat University, Khong Laung, Prathumthani 12121
kt@siit.tu.ac.th

Abstract

This paper presents an empirical result of using two thread placement algorithms for load balancing and adaptive parallelism in non-dedicated environments. The first algorithm called MC places threads on nodes so parallel load is balanced and thread locations incur minimum inter-node communication. The second algorithm called LS relates thread's computation and communication to approximate balanced load and uses a simple hill-climbing algorithm to find optimized thread locations.

Our results showed that MC outperformed LS in all cases but both algorithms were not as good as adaptive parallelism when the number of available nodes is high. Later, we found that the small improvement in MC and LS was caused by time-sharing effects on non-dedicated nodes as performance of the modified algorithm to avoid such effects, called MC*, was consistently higher than adaptive parallelism. MC* achieves 85-90% of ideal speedups by average in all of our tests.

1. Introduction

Shared-memory machines and applications become increasingly popular as business organizations buy more affordable, small-scaled parallel technologies, such as SMPs, multi-core CPU machines, and small clusters of SMPs, for their enterprise computing. Traditional parallel folks suggest users to avoid non-dedicated executions, e.g. not to run parallel program with another guest program, for a good reason—to achieve high performance and avoid time-sharing effects [8].

Nevertheless, since new technology keeps increasing CPU speed, more work can be processed quickly and it is more cost-effective to share CPUs most of the time. Time sharing in parallel computing has benefits such as reducing job wait time and increasing overall throughput. Unfortunately, it is the interaction between parallel applications and operating systems that degrades parallel performance. Thus, our research goal is to find an efficient, practical, and easy to use parallel development tools to solve this problem.

As guest processes demand resources at variable rate and dynamically acquire resources at different time, load balancing in non-dedicated environments is more complex than in other environments [6]. Well-known solutions are in one of the three approaches: avoiding any guest process at all time, load balancing and scheduling, and adaptive parallelism that combines the other approaches together.

First, the avoidance approach requires parallel system software to relocate parallel computation from non-dedicated to dedicated nodes. At runtime, images of a running application must be able to move on the fly. Condor [9] is the first system to use this approach and utilize idle workstations aggressively. The worst case of Condor occurs when no more dedicated nodes can be found in a cluster. In fact, we do not think this approach fits well the execution environment on SMPs and a small cluster of workstations so we will not discuss this approach here.

Second, the load balancing approach adopts many techniques from traditional parallel researches [10]. This approach attempts not to avoid non-dedicated nodes but to reduce negative effects of communication bottleneck and imbalance computation caused by resource sharing among nodes. Typically, this application-specific approach was considered difficult for novice users and many researchers could not obtain speedups in non-dedicated environments. Several also studied sources of the problem and showed that not only computation load must be balanced some enhancements such as co-scheduling and optimized global synchronization are needed to use this approach efficiently [1].

Third, adaptive parallelism merges load balancing and load avoidance approach into a state-of-the-art approach. Piranha [4], for example, balanced parallel load only on nodes not affected by guest processes. Using adaptive parallelism, parallel applications must be able to adjust portions of local computation freely on a set of nodes, resulting in a dynamic change of the number of nodes. Unlike Condor, adaptive parallelism does not explore extra dedicated nodes more than what an application has at the moment of its adaptation. This way the application keeps load in balance and effectively avoids non-dedicated nodes at the same time. Adaptive parallelism has one drawback: application performance is always bounded by the number of nodes that the application can use. This leads us to an interesting question: Can load balancing approach really outperform adaptive parallelism approach? In this paper, we compare effectiveness of the two approaches using our thread placement algorithms under the same condition.

*This work was supported in part by Thailand Research Fund (PDF/72/2544)

Let bar_e : barrier epoch time (user-defined or iteration time) on node G_e
$wait_e$: waiting delay of remote communication on node G_e
os_e : operating system cost such as thread context switch delay on node G_e
N : the number of processors going to use (avoiding highest loaded nodes when possible)
$|G_e|$: the number of threads on node G_e

Then,

MC = *1.* find computation load per thread P_e on G_e that is $P_e = (bar_e - wait_e - os_e)/|G_e|$

2. find the new number of threads NG_e on node G_e from: $NG_e = \frac{P_e}{\sum_{i=1}^{N} P_i} \times \sum_{i=1}^{N} |G_i|$

If NG_e is not integer, truncate NG_e in step2 and redistribute remaining threads to the minimum loaded nodes until all individual threads are assigned to all N nodes.

3. After integer solution of step 2 is known, put individual threads onto N nodes, each with NG_e threads using a heuristic described in. [11] The heuristic uses correlation values to put threads to locations that minimize inter-node communication.

Figure 1: MC balances computation and minimizes communication in separation.

2. Algorithms and mechanisms

We modified DCVM, a software distributed shared memory system (DSM), that supports coarse-grain dynamic load balancing using thread migration [12]. To allow adaptive parallelism like in Piranha, DCVM must be able to move all threads out of a node and produce an empty node that needs no CPU cycles. However, under lazy-released protocol [7] used by DCVM, remote threads may delay shared-page requests to empty nodes. Thus, on every empty node, DCVM must create a skeleton thread to help processing an incoming page request and the skeleton thread is put to sleep as soon as the page request is done.

Next, we implemented two thread placement algorithms that compute thread configurations with a possibility of node expansion or reduction. Figure 1 shows the first algorithm called MC. MC evaluates computation capacities on local nodes (step 1 and 2). Then, it applies MINCOMM algorithm to put individual threads on nodes. Details of MINCOMM can be read in [11]. The second algorithm called LS used a simple hill-climbing to search all thread location candidates for the best with minimum load imbalance. Figure 2 shows the definition of LS. Basically, a simple load model, L, is used to approximate local computation and communication delay of various attempts to move a thread from a heavily-loaded node to other nodes. The approximate load is a linear proportion of computation and communication numbers against the sum of *correlation value* ($C[i,j]$) used in [14]. Note that current LS search does not utilize look-ahead iteration. Similar to MC, LS may not find an optimal solution due to the use of hill-climbing and no structure to guarantee whether local maxima are also global maxima.

3. Experimental results

We ran eight shared memory parallel applications on four 266 MHz Pentium-II, 256 MB Linux PCs connected by Myrinet gigabit switch and collected performance numbers under three different arrangements. Table 1 and Table 2 list the shared memory parallel applications and details of

Let G : a set of nodes
G_i : a set of threads on node i
G_{ij} : thread j on node i
H : a set of highest load nodes
$C[i,j]$: value of correlation between thread i and thread j captured at runtime by DCVM (see) [14]
M_e : average communication delay per cut-around correlation values $\sum_{s \in G_e, t \notin G_e} C[s,t]$ on node G_e
P_e : average computation delay per number of threads on node G_e (same as in MC)
Note: M_e and P_e are collected before executing *LS*

Then

$$LS = Min_{\forall p \in G - H, \forall q \in H, \forall x \in q} \{Max\{Load(x,p,q)\} - Min\{Load(x,p,q)\}\}$$

where $Load(x,p,q)$ is a set of load values after moving $x \in q$ to p that is

$$Load(x,p,q) = \{L(G_p \cup \{x\}), L(G_q - \{x\}), \forall_{i \in G - \{p,q\}} L(G_i)\} \text{ and } L(G_e) = P_e \times |G_e| + M_e \times \sum_{s \in G_e, t \notin G_e} C[s,t]$$

Figure 2: LS minimizes load imbalance using approximated load model $L(G_e)$

Table 1: Parallel applications used in our suite

Applications	Problem Sizes	Shared Pages (8K/page)	Epochs	Total Iterations
ADI	64K	2320	12	24
EXPL	512x512	2512	12	24
FFT	64x64x128	3587	12	12
Gauss	2048x2048	2050	12	48
Spatial	4096 mols	399	12	12
OR	2048x2048	4097	12	12
SWM	512x512	2005	12	24
Water	512 mols	43	12	12

Table 2: Non-dedicated arrangements

Test Name	CPU utilization observed on each processor before tests				Remaining CPU power (1 node = 100% and assume fair scheduling)
	P0	P1	P2	P3	
A1	0	0	0	100%	350%
A2	0	0	100%	100%	300%
A3	0	100%	100%	100%	250%

three non-dedicated arrangements used in our experiments respectively. All the applications were from well-known benchmark suites, modified to support thread migrations, and linked with DCVM runtime library to allow dynamic load balancing and adaptive parallelism. To perform adaptive parallelism, we simply applied MC with larger or smaller N, the number of nodes to our algorithms, and avoid non-dedicated nodes first. In the past, we also tested a version of LS that allowed adaptive parallelism but its performance was inferior to MC so we did not discuss it in this paper.

Figure 3 shows *average* performance of all eight applications under the three arrangements, each with one of the five actions: no load balancing (NLB), adaptive parallelism (AP), MC thread placement (MC), LS thread placement (LS), and a variation of MC called MC*. About the last algorithm, we will discuss in the next section. The performance numbers on y-axis were normalized against approximate speedups to the remaining CPU cycles (see the last column of Table 2). We want y-axis to represent how efficient each algorithm can harvest remaining CPU cycles per arrangement.

We expected that if the applications were not affected by non-dedicated executions at all, their normalized speedups would be exactly 1.0. That means the portion of remaining CPU cycles can be utilized on all four nodes, some of which with a guest process. For example, if application speedup is four on four dedicated nodes, the approximate speedup-bound with a guest program on one node will be 3.5. So, if we can measure a speedup of 3.5 with non-dedicated execution, the normalized speedup will be 1.0. We believe such an approximation is reasonable if load balancing is succeeded in parts because we used an infinite-loop program as the guest process.

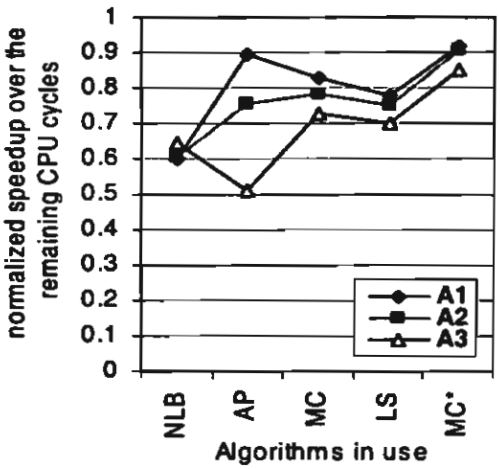

Figure 3: Normalized, average speedup from various approaches (NLB, AP, MC, LS and MC*) in three non-dedicated arrangments (A1, A2 and A3)

First, in Figure 3, we observe that without load balancing, application performances are reduced by 40%. If we assume that half of the CPU power on each node is divided equally and synchronously shared by parallel and guest process, we should see around 50% degradation, [13].

Using adaptive parallelism, our applications avoided nodes that contained a guest process. Consequently their average speedups were reduced as more nodes were out. In arrangement A1 and A2, adaptive parallelism produced speedups higher than no load balancing. This confirms benefits of the rule-of-thumb that suggests parallel users to avoid non-dedicated executions. On the contrary, if the applications were about to run sequentially, using two nodes with no load balancing would be a better choice. Note that so far we discussed only the performance gain to parallel applications and did not consider the increasing complexity in workload and subsequent degradation to the guest processes.

Comparing our thread placement algorithms, we saw that MC was slightly better than LS in all arrangements. On the other hand, we expect the poor performance of LS was caused by imprecision of our load model (L) as well as the hill-climbing design. We were further disappointed by the overall performance of MC and LS that was not as good as adaptive parallelism in A1 and A2.

Consequently, we decided to enhance MC further so the modified algorithm tried to avoid scheduling and time sharing effects in non-dedicated environments [2, 13]. The new MC named MC* yields CPU cycles via micro-sleep every time it waited for remote messages. As noted in [13], this enhancement helped reducing penalty from time-sharing effects and required no modifications to operating systems. Figure 3 confirms our expectation that load balancing could perform better than adaptive parallelism.

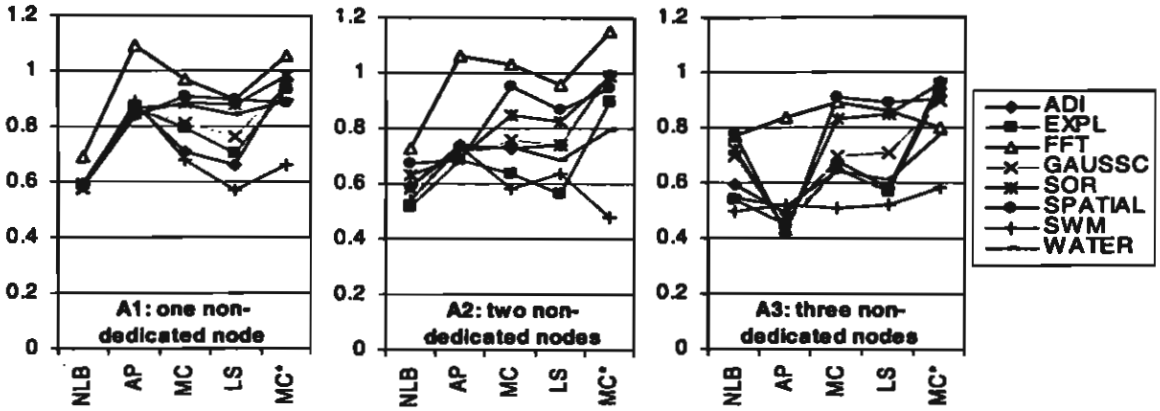

Figure 4: Normalized performance of individual parallel applications used in our suite.

We found that the performance of MC* was consistently better than adaptive parallelism in all three arrangements. MC* showed only 10% to 15% less efficient than the ideal speedups.

To note that most applications behaved well with our hybrid load balancing approach but some did not, Figure 4 plotted our results per application. Overall, SWM is the only application contradicting a number of our findings. Its performances with MC* were not always better than adaptive parallelism. Further investigation suggested that both LS and MC (and so MC*) in SWM failed to reduce communication during the load balancing process. In the future, we will try to make DCVM realize this kind of situations and decide to choose adaptive parallelism over load balancing. This will lead to a smart system that can exploit benefits of space-sharing and time-sharing more efficiently and transparently. Majority of the results in Figure 4 are consistent with Figure 3 so we do not discuss them further here.

Finally, the improvement we made to MC* repeated lessons learn from failures not to address negative impacts between parallel applications and operating systems in previous work. Obviously, it is the effect of time sharing that allows MC* to be better than applying either adaptive parallelism or load balancing alone. Although one would argue that we hardly ran parallel applications on non-dedicated nodes, he/she did not say that we could not do so effectively. Our future investigation will go to explain how we can build such system that is less complicated, easy to use and more practical on inexpensive SMPs or multi-core CPU PCs where non-dedicated executions are norm.

4. Related work

Shared memory parallel applications must handle network latency efficiently and minimize data sharing across nodes to produce speedup. Multithreading has been used to solve these problems on both hardware and software systems for a long time [5]. DCVM is one of many systems that use multithreading to speed up parallel applications.

Previous work in non-dedicated environments takes task- and process-based approach to enhance application performance where node dedication cannot be guaranteed. Condor [9], Piranha [4], and AppLes [3] are examples of software systems using load sharing, adaptive parallelism, and application-level scheduling to solve non-dedicated problems respectively. On the other hand, DCVM attempts to take benefits from multiple approaches that are ready to be used *selectively* for performance optimization.

5. Conclusions

We introduce two thread placement algorithms, MC and LS, and present an empirical result based on the real implementations in DCVM. Our approach is to allow load balancing and adaptive parallelism to be used dynamically and co-exist at application level.

Our experiments showed that: with no load balancing, the overall performance was degraded by 40%. Adaptive parallelism improved speedups in most cases, except sequential runs. MC and LS improved overall performance slightly about 5% to 10% over no load balancing and that was less than adaptive parallelism. Further investigations led us to MC*, an enhancement of MC to reduce time-sharing effects using a technique presented in [13]. MC* showed the highest performance among the algorithms we tested. It generated small 10% to 15% overhead and more CPU cycles were harnessed better than using either adaptive parallelism or load balancing approach. However, SWM was the only application that performed badly when thread placement failed to find optimal thread locations. We suggested a solution that if such a case could be detected, adaptive parallelism should have been activated by DCVM. We are making research progress to DCVM in this direction.

6. References

[1] Arpaci, A. and D. Culler. Extending Proportional-Share Scheduling to a Netswork of Workstations. in PDPTA. 1997.

[2] Arpaci-Dusseau, A.C., D.E. Culler, and A.M. Mainwaring. Scheduling with Implicit Information in Distributed Systems. in Sigmetrics'98 Conference on the Measurement and Modeling of Computer Systems. 1998.

[3] Berman, F. and R. Wolski. Scheduling from the Perspective of the Application. in Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing. 1996.

[4] Carriero, N., et al., Adaptive Parallelism and Piranha. IEEE Computer, January 1995. 28(1): p. 40--49.

[5] Gupta, A., et al. Comparative evaluation of latency reducing and tolerating techniques. in sigarch91. May 1991.

[6] Harchol-Balter, M. and A.B. Downey. Exploiting process lifetime distributions for dynamic load balancing. in SIGMETRICS. 1996. Philadelphia, PA.

[7] Keleher, P., A.L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. in Proceedings of the 19th Annual International Symposium on Computer Architecture. May 1992.

[8] Leutenegger, S. and X.H. Sun. Distributed Computing Feasibility in a Non-Dedicated Homogenous Distributed System. in Supercomputing. 1993. Portland, OR.

[9] Litzkow, M., M. Livny, and M. Mutka. Condor - A Hunter of Idle Workstations. in International Conference on Distributed Computing Systems. 1988.

[10] Stone, H.S., Multiprocessor Scheduling with the aid of Network Flow Algorithms. IEEE Transactions on Software Engineering, January 1977. 32: p. 85--93.

[11] Thitikamol, K. and P. Keleher, Thread Migration and Communication Minimization in DSM Systems. The Proceedings of the IEEE, 1998. 87(3): p. 487-497.

[12] Thitikamol, K. and P. Keleher. Thread Scheduling and Grain Emulation. in The 14th International Conference on Parallel and Distributed Computers and Systems. 2002. Boston, MA: IASTED.

[13] Thitikamol, K. and P. Keleher, Communication-Intensive Parallel Applications and Non-Dedicated Environments, in Proc. of the 3rd Workshop on Runtime Systems for Parallel Programming (RTSPP). April 1999.

[14] Thitikamol, K. and P.J. Keleher. Active Correlation Tracking. in The 19th International Conference on Distributed Computing Systems. June 1999.

THREAD SCHEDULING AND GRAIN EMULATION IN SOFTWARE-DSM SYSTEMS

Kritchalach Thitikamol*
Sirindhorn International Institute of Technology
Thammasat University
Pathumthani 12121, Thailand

Peter J. Keleher
Department of Computer Science
University of Maryland, College Park
MD 20742, USA

ABSTRACT

Many multithreaded systems implement dynamic threads because they allow computation grain to be customized and match available cycles on the fly. In contrast, systems using static threads must rely on users to select a proper degree of multithreading in order to achieve maximum performance. In this paper, we present two algorithms that together provide multithreading performance transparency in a software-DSM system, called CVM. First, our grain emulation algorithm allows static-thread systems to emulate runtime effects of dynamic threads without changing thread implementation. Second, our grain selection algorithm attempts to decide grain size and maximize multithreading performance at runtime.

Our experimental results using five shared memory parallel applications showed that performance effects of emulated grain were consistent with the effects of actual grain in all except two occasions where grain emulations caused unexpected changes in FFT and Shpatial's DSM actions. When we applied grain selection algorithm, CVM produced 30% improvement in Barnes with 64 threads. Other applications also showed some improvements but much less noticeable than Barnes.

KEY WORDS

Multithreaded DSM, Thread Scheduling, Grain Emulation

1. Introduction

Multithreading [1] is an implementation technique that allows a single process to be broken into multiple threads and scheduled to CPU whenever local node is waiting for a remote reply. During multithreaded execution, local computation among threads is overlapped with network latency and thus can be finished faster than single-thread execution. In theory, multithreading is a promising technique for enhancing parallel applications especially in loosely coupled environments like cluster of workstations.

Several factors affect performance of multithreading but the most significant one is the number of threads running on nodes called *degree of multithreading*. Intuitively, increasing degree of multithreading introduces more thread context switches and resource usage. Having not enough threads causes the opposite situation where systems can not effectively overlap local computation with network latency. To achieve maximum performance, multithreaded systems must control effects from degree of multithreading or in another word, they must control basic computation unit called *grain* that is executed without overlapping before global synchronization.

In this paper, we present a grain emulation algorithm that allows static-thread systems to emulate performance effects of various grain sizes. Combining grain emulation with a proposed grain selection algorithm, we show its effectiveness for enhancing multithreading performance in a software distributed shared memory system (DSM), called CVM. In Section 2, we introduce CVM and discuss grain-selection problems. In Section 3, we describe our grain emulation and grain-selection algorithm. In Section 4, we present our experimental results with five iterative shared-memory applications and discuss performance of grain emulation and grain selection algorithm. At the end, we provide related work and conclude the paper.

2. Multithreading and Grain Selection

We introduce CVM and discuss performance problems of multithreaded systems in this Section.

2.1 CVM and Per-node Model

Like other software-DSMs, CVM is a runtime system that provides shared memory semantics across machines connected by a message-passing network. Particularly, CVM installs OS protection traps to catch memory accesses to virtual memory pages and transform them into remote requests according to DSM memory consistency protocol. After shared page is validated, no subsequent DSM actions are necessary until page is invalidated again. The small number of page validations helps reducing DSM overheads and improving multi-processor speedups.

CVM's primary protocol implements a *multiple-writer* version of *lazy release consistency*. In lazy release consistency, a processor delays making modifications to shared data visible to other processors until the time of the next subsequent acquire of a released synchronization. In CVM, consistency states are kept per shared-page so the possibility of protocol actions to be merged or broken into multiple DSM messages by multithreaded execution is not trivial.

* This work was supported in part by Thailand Research Fund (PDF/72/2544).

In multi-writer protocol, two or more nodes can simultaneously modify their copy of a shared-page. These concurrent modifications are merged using *diffs* to summarize the updates. A diff is created by performing a page-length comparison between the current contents of the page and a twin of the page that was created at the first write access. If any concurrent writer summarizes its modifications as a diff, the system can validate or create a copy that reflects all modifications by applying the concurrent diffs to the same copy. Similar to protocol actions, multithreaded executions can increase or decrease possibility of threads sharing same pages by changing page-access order. Nonetheless, only one thread that first accesses shared page sends out remote diff request. The rest are blocked until page validation is completed.

CVM implements non-preemptive threads and uses per-node multithreading model [2]. In per-node model, a thread is created statically at the beginning of application execution and stays active until the program is finished. A thread is also implemented as a process that shares local CVM image with other threads, except its thread stack. Using such a model, CVM allows users to select degree of multithreading at program command line so users can fully control initial grain size, which is *approximately* an inverse proportion to a given degree of multithreading on a homogeneous cluster.

2.2 Grain Selection Problem

In the past, multithreading experiments on software-DSMs [3] studied effects to application performance but did not provide solutions to allocation problem especially when applications were run with too many threads. Deciding optimum grain size was limited to experienced users who understand program decomposition and process allocation on a target parallel machine well. From users' perspective, how the applications should be run to yield the shortest execution time is the most important question.

A majority of grain-selection researchers prefer dynamic threads over static threads because in dynamic thread systems, excessive degree of multithreading can cause threads to be merged or destroyed independently. When more parallelisms can be exploited, new threads can later be created upon demands. Other systems that adapt level of parallelism implicitly may require ability to redistribute global computation and advanced compiler analyses in order to perform proper thread decomposition. Consequently, dynamic threads in these systems are often short-lived and require a number of thread management operations. Choosing dynamic threads over static threads then trades additional off-line processing and execution-time overheads to performance transparency.

On the other hand, CVM implements static threads so it requires fixed per-thread context-switching overheads and no further thread manipulation costs. Moreover, users can select computation grain by writing it in the program and no changes to users' programming preferences and offline requirements. Nevertheless, to avoid application slowdown once users run their applications with too many threads, CVM definitely needs other runtime techniques to make multithreading performance transparent and they must be suitable for static thread systems. Therefore, we invent grain emulation and grain selection algorithm to emulate some behaviors of dynamic threads and solve the grain selection problem transparently.

3. Grain Emulation Algorithm

After a user runs an application with a specific number of threads per node, CVM has no way to adjust actual grain decomposed by selected degree of multithreading on local nodes. However, CVM can decide when threads should be scheduled and try to control multithreading effects. For example, if CVM does not dispatch local threads though node has outstanding remote reply, the execution will result in one thread finished after another. Consequently, it should produce runtime effects *close* to a typical single-thread execution.

Thus, we perform grain emulation as follows:

1. CVM bundles local threads into groups. Figure 1(a), 1(b), 1(c) and 1(d) show groups of 1, 2, 4, and 8 threads, respectively. How to assign threads into groups and how many groups should be used will be discussed in Section 3.1 and 3.2, respectively.
2. After local threads are grouped, CVM enforces multithreading scheduling among threads within a single group at a time. In the worst case shown in Figure 1(a), no latency hiding exists because no overlapping of network latency can occur.
3. At barrier arrivals, CVM continues to another group of threads at *random* and performs step 2) repeatedly until the program ends.

Figure 1(e) also shows unequal number of threads per group that causes average grain size to be a fraction. We will not discuss this kind of emulated grain in this paper but undoubtedly this special case will be necessary when CVM attempts to emulate grain on a node whose number of threads is not divisible by CVM's desired number of groups.

Next, we answer two important problems: 1) how to group individual threads and 2) how to select proper grain size to maximize application performance.

3.1 Grouping Threads

Placing threads in groups requires considering subsequent effects of program decomposition such as communication requirements, needed synchronizations, and in our case effects of shared memory consistency protocols. To group threads, CVM uses *active correlation tracking* to capture global page-sharing information among threads for one application iteration. Then, CVM places local threads that access many *same* shared pages together according to group sizes determined by CVM (see Section 3.2).

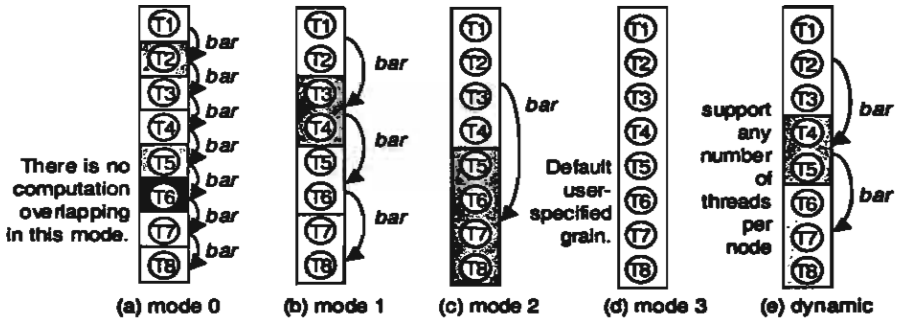

Figure 1: Local Scheduling of Threads to Emulate Multithreading Effects of Various Grain Sizes.

Active correlation tracking [4] is a global process that tracks all shared-page accesses by threads regardless of their locations and summarizes page accesses into thread correlations that represent affinities among threads by the number of shared pages used mutually between any two threads. Placing threads with high correlations together allows DSM protocol actions to be requested minimally and reduce communications among nodes [5].

For readability, we show page-sharing information in *correlation maps*. Figure 2 shows correlation maps of five parallel applications with problem sizes listed in Table 1. Each dimension of the maps represents thread ids so a coordinate contains a correlation value of thread x and y. In Figure 2, correlation values are shown in levels of gray. That is, darker the area, higher the correlation values. All correlation maps are symmetric since our page sharing assumes no direction.

Patterns in Figure 2 confirm that these applications produce nearest-neighbor communication, except Barnes and Shpatial that have random and square patterns in the correlation maps besides high correlations seen around

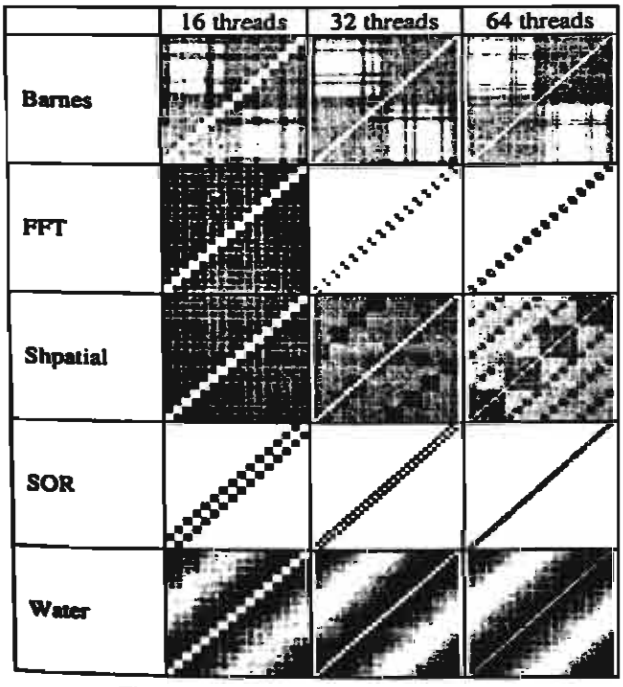

Figure 2: Application Correlation Maps

diagonals like other nearest neighbor applications. Note that we previously tried a group-selection policy that chose next thread group from the highest sum of inter-group correlations. Without surprise, later we found that this policy had no impact to grain-emulation performance because CVM scheduled groups many times less often than threads. So, we prefer a random policy in step 3) of the grain emulation algorithm rather than a fancier one.

3.2 Grain Selection Algorithm

To decide how many threads should be used per group, CVM monitors reduction in iteration elapsed time. This iteration time reflects how efficient CVM overlaps local computation with network latency and how many computing resources threads and memory protocol use for a given grain size. Since this decision is an optimization problem, we decide to use trial-and-error search in which for each trial CVM systematically changes number of threads in groups to emulate various grain sizes. The choice to either decrease or increase grain size depends on CVM's acknowledge of performance changes and current thread configuration compared to previous iterations.

For example, CVM will decrease number of groups or decrease grain size if more computation is needed to hide network latency. Otherwise, if less computation overlapping is needed, it will increase number of groups or increase grain size to optimize overall performance. All of these can be observed from reduction in application iteration time. Moreover, we implement five-percent performance threshold to justify significant change before making a decision to stop the trial-and-error process. We expect that the threshold should help reducing noises and making grain selection process repeatable in our experiments.

To reduce complexity of our study, we attempt to select grain size by *mode number*. That means CVM uses trial-and-error search in power-of-two threads per group like the example in Figure 1. Moving from Figure 1 (d) to 1 (a) results in trials that increase grain size by decreasing mode number from 3 to 0, and vice versa.

If no improvement can be found during the trial-and-error process, the algorithm stops and final grain size is reset to the best-known one. Since CVM uses no look-ahead iteration, this is just to use grain size from the last

Name	Description	Problem Size	Type of Sync.	No. of Shared Pages
Barnes	N-body gravity simulation	8K body	lock, barrier	500
FFT	Fast Fourier Transform	64×64×128	Barrier	3600
Shpatial	Spatial Molecular Dynamic	4096 mols	lock, barrier	340
SOR	Successive Over-Relaxation	1024×2048	Barrier	2050
Water	N^2 Molecular Dynamic	512 mols	lock, barrier	50

Table 1: Applications and Their Problem Sizes

iteration or before iteration elapsed time starts to increase over our threshold. We believe changing emulated grains on all nodes in steps has two benefits. First, we can keep grain emulation in balance among nodes. Second, we can reduce the number of cases studied in our experiments.

4. Experimental Results

We implemented grain emulation and grain selection algorithm in CVM and tested their performances on a cluster of four 733 MHz Intel PIII workstations connected by 100M-bit Ethernet switch. Each workstation had 256-megabyte memory and ran Linux kernel version 2.4.7. Our objective is to study effects of emulated grain to multithreading performance and grain selection benefits. Table 1 shows details of five iterative parallel applications and the problem sizes used in our experiments. Barnes, Shpatial, and Water are from SPLASH benchmark suite [6]. FFT and SOR are well-known scientific computations that implemented Fast Fourier Transform and Successive Over-Relaxation.

4.1 Performance Effects of Grains

We ran five applications using 4, 8, 16, 32, and 64 threads on four nodes with grain emulation algorithm ***but no grain selection*** and collected elapsed time to compute multithreading speedups. The results are shown in Figure 3 where x-axis presents approximate grain to a specified problem size X (see Table 1) in which we call X *roughly decomposed* (or "|") by 4, 8, 16, 32, and 64 threads. For example, with 64 threads, $X|4$, $X|8$, $X|16$, $X|32$, $X|64$ can be emulated by having 1, 2, 4, 8, and 16 threads *per group* respectively. The y-axis in Figure 3 shows normalized speedups to four-processor single-thread execution for given numbers of actual threads and approximate grains. For readability, we also marked default multithreading speedups with box (□). A series of boxes then represents what would happen if we do not use grain emulation but increase degree of multithreading in each application.

In Figure 3, speedups are curved as applications were affected by different grain sizes. Default multithreading speedups (with no emulation) range from 5% to 12% improvement, except SOR that produced only a-few-percent improvement with 8 and 16 threads. The problem caused by too many threads can be seen in all applications, except only FFT. Barnes, Shpatial and Water ran best at $X|32$, $X|32$ and $X|16$, respectively and dropped down by as much as 60% from the best run in Barnes with 64 threads. Others are slowed down by a few percents but still on positive side. SOR, the simplest application in the application suite, produced multithreading slowdown with 32 and 64 threads because there was so limited amount of network latency that could be efficiently hidden in this small application.

When considering performance of grain emulation, we found strong relationships between performance of actual grain and emulated grain. However, to see that we must include overheads of thread context switches, which is easily controlled in dynamic thread system but not in static one. For example, in Figure 3, the degradation in Barnes with 64 threads could be improved by dividing 64 threads into two groups, resulting in grain size $X|32$ and its performance that was as good as running 32 threads without grain emulation. The same phenomenon with 64 threads also appeared in Water with a few percent

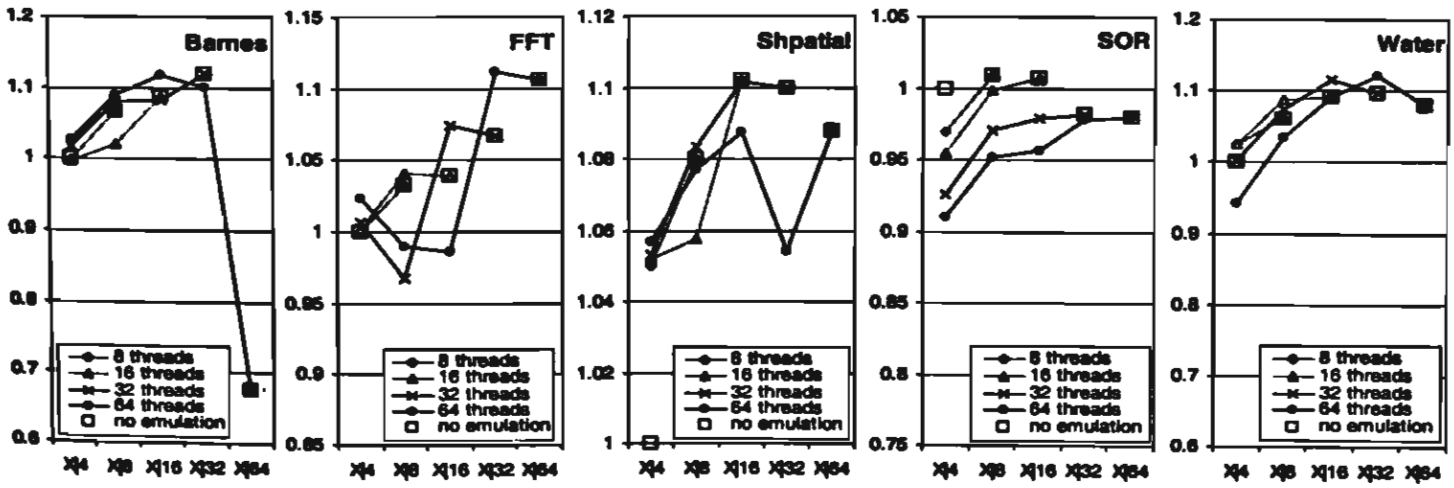

Figure 3: Effects of Actual Grains and Emulated Grains to Multithreading Performance

deviation. In SOR, if we account for thread context switch overhead increased by the number of running static threads, performance effects of emulated grain and actual grain will be close and we believe it could be potentially predictable.

Next, FFT and Shpatial showed improvements that were less consistent. With 64 threads, decreasing FFT grain to *X|32* improved performance higher than running with 32 threads but decreasing grain further gave no speedups at all. Further investigation pointed us to load imbalance mainly on the first processor. Although the total number of remote faults remains the same, global barrier imbalance increased significantly because of additional *diffs* to the first processor from changes in shared-memory access order. The opposite situation can explain a sharp drop in Shpatial with 64 threads. In that case, remote faults at *X|32* were about two times higher than *X|64*. This means scheduling threads in groups can perturb shared memory accesses by changing order of protocol actions.

4.2 Performance of Grain Selection

Next, we ran the same applications with 64 and 32 threads but this time we allowed CVM to adjust application performance using grain selection algorithm described in Section 3.2. We also made sure that the whole selection process had to be done within ten iterations and we excluded the first two iterations to avoid including startup cost and other interferences in our results.

Figure 4 and 5 shows four-processor speedups of the five applications with 64 and 32 threads, respectively. All were measured for the same number of iterations per application: 1) with no grain selection, 2) with grain selection including trail-and-error overhead in *decreasing* grain direction, 3) with grain selection including trail-and-error overhead in *increasing* grain direction, 4) with grain selection *excluding* trail-and-error overhead in decreasing grain direction, 5) with grain selection *excluding* trail-and-error overhead in increasing grain direction, and 6) with the *best* performance for the same number of actual threads from previous experiment.

With 64 threads, Barnes that previously slowed down by 30 percents now achieved 10 percent multithreading speedup or about 60 percent improvement. In this case, CVM decided to emulate grain *X|32* from both decreasing and increasing direction. Next, the similar improvement was seen in Water where 64-thread run caused slight degradation, but after the grain selection algorithm decided *X|32*, its performance was at the best. Results from other applications led to similar conclusions where CVM selected grains that were at the best or almost the best performance achievable by running 64 threads. Also, a quick glance to Figure 5 that tested with 32 threads confirmed the same results in which any attempts to transparently select grain size never caused slowdown.

Next, when we consider application improvements including iterations that CVM used to perform trail-and-error search, the results in Figure 4 and 5 showed 10 to 30 percent slowdown for the specific number of application iterations that we tested. In all cases, CVM stopped the grain selection process within six application iterations and this runtime overhead was consistent with emulated grain effects in Figure 3. All applications except Barnes cost more to decrease grain than to increase grain. Fortunately, if improper grain size becomes performance problem, it is likely that users are running applications with too many threads and so CVM will execute grain selection algorithm in increasing grain direction.

Finally, we noticed that once CVM attempted to emulate grain *X|32* in Shpatial, the speedup drop at *X|32* (see Figure 3) caused the grain selection algorithm to stop and take either *X|16* or *X|64* depending on trial directions. The sub-optimal problem that occurs with 64 threads was directly from trial-and-error technique implemented in our algorithm. Thus, if we applied look-ahead iteration, this problem would be alleviated in some extent. Note that a similar speedup drop also occurred in FFT but not enough to cause the algorithm to stop. Nonetheless, if we did not

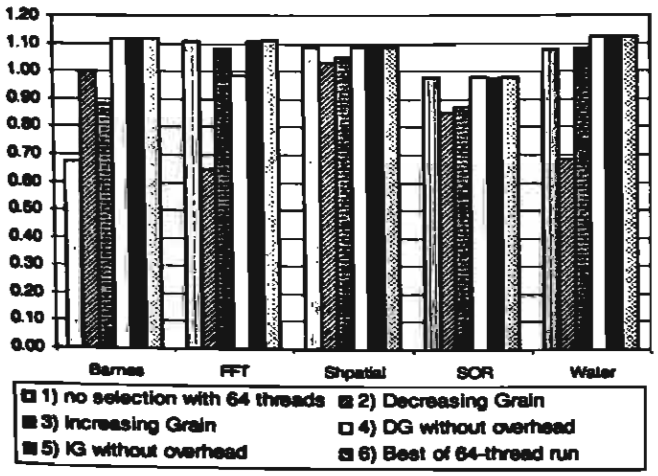

Figure 4: Performance of Grain Selection with 64 threads

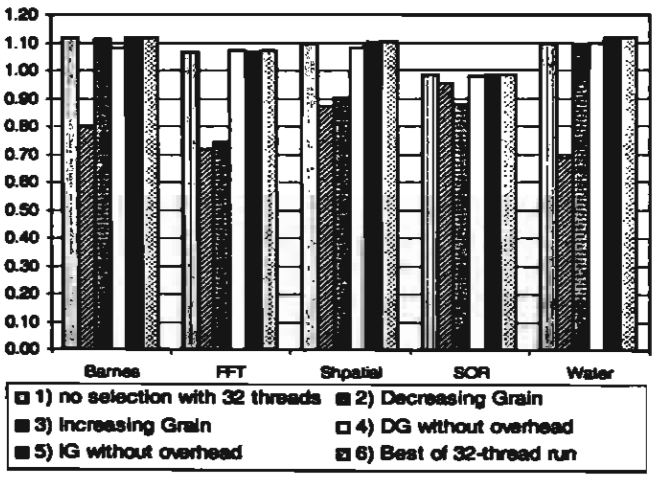

Figure 5: Performance of Grain Selection with 32 Threads

impose five percent threshold, we would have noticed the sub-optimal problem in FFT too.

5. Related Work

Fine-grain parallelisms [7, 8] have long been used to provide performance transparency for scientific parallel applications. Such systems required supports from both compilers and runtime systems and often required specific hardware and software. This technique later expanded to applications and tools for coarser grain like multithreaded environments that exploited parallelisms on local nodes to effectively hide network latency. The comparison studies between coarse-grain and fine-grain have been done in many levels of granularity including shared data access [9]. Our work attempts to emulate grain and control grain size of static thread systems where we consider as another method to better exploit available parallelisms.

Many systems in the past supported multithreading for latency hiding as well as performance transparency like dynamic task creation [10]. Since they implemented dynamic threads, their techniques were not compatible to CVM. Our thread scheduling algorithm proposed here for grain emulation is similar to other optimizing schedulers in that ours also attempts to reduce effects to DSM actions and allows local computation to benefit from data locality existing on globally shared memory. More discussions about grain selection problem on other environments can be found in [11].

6. Conclusions

In the past, multithreaded systems that implemented static threads could not adjust computation grain and control multithreading performance at runtime. In this paper, we present a novel thread-scheduling algorithm that allows static threads to emulate effects of dynamic threads and grain-selection algorithm that implements trial-and-error process to maximize multithreading performance using grain emulation at runtime.

Our experimental results with five iterative parallel applications confirmed that the effects of emulated grain were very consistent with effects of actual grain. Among five applications, Barnes with 64 threads slowed down by 30 percents from its single-thread execution but after we applied grain-selection algorithm, its speedup was at the maximum or about the same as running the application with 32 threads with no grain emulation. That was about 60 percent improvement. More importantly, the grain selection algorithm never decided to emulate grain that performed poorer than its initial, user-specified grain in all applications. Cost of trial-and-error process was between 10 to 30 percent after we included iterations containing grain selection overhead. Nonetheless, this one-time overhead could be amortized easily by lengthening the program execution.

Currently, CVM also uses grain emulation and grain selection to deal with dynamic load balancing that relies on thread migrations. In that case, parallel load may be balanced but numbers of threads can be greatly different among nodes resulting in high multithreading overheads on some nodes. Our future work will be in this direction. That is to better understand software multithreading and load-balancing effects of various DSM memory protocols.

7. Reference

[1] Gupta, A., *et al.*, Comparative Evaluation of Latency Reducing and Tolerating Techniques, in *Proceeding of the 18th Annual International Symposium on Computer Architecture*, May 1991.

[2] Thitikamol, K. and P. Keleher, Per-Node Multithreading and Remote Latency, *IEEE Transactions on Computers*, *47*(4) 1998, p. 414-426.

[3] Mowry, T.C., *et al.* Comparative Evaluation of Latency Tolerance Techniques for Software Distributed Shared Memory, in *Proceedings of the 14th International Symposium on High-Performance Computer Architecture*. February 1998.

[4] Thitikamol, K. and P. Keleher. Active Correlation Tracking, in *The 19th International Conference on Distributed Computing Systems*, June 1999.

[5] Thitikamoi, K. and P. Keleher, Thread Migration and Communication Minimization in DSM Systems, in *The Proceedings of the IEEE*, *87*(3) 1998, p. 487-497.

[6] Woo, S.C., *et al.* The SPLASH-2 Programs: Characterization and Methodological Considerations, in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995.

[7] Lowenthal, D.K., *et al.*, Using Fine-Grain Threads and Run-Time Decision Making in Parallel Computing, *Journal of Parallel and Distributed Computing*, *37*(1) 1996, p. 41-54.

[8] Arvind, *et al.*, Assessing the Benefits of Fine-grain Parallelism in Dataflow Programs, in *Proceedings of the 1988 Conference on Supercomputing*. 1988.

[9] Dwarkadas, S., *et al.* Comparative Evaluation of Fine- and Coarse-Grain Approaches for Software Distributed Shared Memory, in *Proceedings of the 5th High Performance Computer Architecture Conference*, January 1999.

[10] Mohr, E., *et al.* Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs, in *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*. 1990.

[11] Siegell, B.S. and P.A. Steenkiste. Controlling Application Grain-size on a Network of Workstations, in *Proceedings of the 1995 Conference on Supercomputing*, 1995, San Diego, California.