View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by INRIA a CCSD electronic archive server

A Flexible Thread Scheduler for Hierarchical Multiprocessor Machines

Samuel Thibault

To cite this version:

Samuel Thibault. A Flexible Thread Scheduler for Hierarchical Multiprocessor Machines. Second International Workshop on Operating Systems, Programming Environments and Management Tools for High-Performance Computing on Clusters (COSET-2), ICS / ACM / IRISA, Jun 2005, Cambridge, United States. inria-00000138

HAL Id: inria-00000138
https://hal.inria.fr/inria-00000138

Submitted on 27 Jun 2005

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Flexible Thread Scheduler for Hierarchical Multiprocessor Machines

Samuel Thibault
LaBRI
Domaine Universitaire, 351 cours de la libération
33405 Talence cedex, France
samuel.thibault@labri.fr

inria-00000138, version 1 - 27 Jun 2005

ABSTRACT

With the current trend of multiprocessor machines towards more and more hierarchical architectures, exploiting the full computational power requires careful distribution of execution threads and data so as to limit expensive remote memory accesses. Existing multi-threaded libraries provide only limited facilities to let applications express distribution indications, so that programmers end up with explicitly distributing tasks according to the underlying architecture, which is difficult and not portable. In this article, we present: (1) a model for dynamically expressing the structure of the computation; (2) a scheduler interpreting this model so as to make judicious hierarchical distribution decisions; (3) an implementation within the Marcel user-level thread library. We experimented our proposal on a scientific application running on a ccNUMA Bull NovaScale with 16 Intel Itanium II processors; results show a 30% gain compared to a classical scheduler, and are similar to what a handmade scheduler achieves in a non-portable way.

1. INTRODUCTION

"Disable HyperThreading!" That is unfortunately the most common pragmatic answer to performance losses noticed on HyperThreading-capable processors such as the Intel Xeon. This is of particular concern since hierarchy depth has increased over the past few years, making current computer architectures more and more complex (Sun WildFire [10], Sgi Origin [13], Bull NovaScale [28] for instance).

Those machines look like Russian dolls: nested technologies allow them to execute several threads at the same time on the same core of one processor (SMT: Simultaneous Multi-Threading), to share cache memory between several cores (multicore chips), and finally to interconnect several multiprocessor boards (SMP) thanks to crossbar networks. The resulting machine is a NUMA (Non-Uniform Memory Architecture) computer, on which the memory access delay depends on the relative positions of processors and memory banks (this is called the "NUMA factor").

The recent integration of SMT and multicore technologies make the structure of NUMA machines even more complex, yet operating systems still have not exploited previous NUMA machines efficiently. Hennessy and Patterson underlined that fact [11] about systems proposed for SGI Origin and Sun Wildfire: *"There is a long history of software lagging behind on massively parallel processors, possibly because the software problems are much harder."* The introduction of new hardware technologies emphasizes the need for software development. Our goal is to provide a *portable* solution to enhance the efficiency of high-performance multithreaded applications on modern computers.

Obtaining optimal performance on such machines is a significant challenge. Indeed, without any information on tasks' affinity, it is difficult to make good decisions about how to group tasks working on a common data set on NUMA nodes. Detecting such affinity is hard, unless the application itself somehow expresses it.

To relieve programmers from the burden of redesigning the whole task scheduling mechanism for each target machine, we propose to establish a communication between the execution environment and the application so as to automatically get an optimized schedule. The application describes the organization of its tasks by grouping those that work on the same data (memory affinity) for instance. The system scheduler can then exploit this information by adapting the task distribution to the hierarchical levels of the machine.

Of course, a universal scheduler that would get good results by using only such a small amount of information remains to be written. In the meantime, we provide facilities for applications to query the system about the topology of the underlying architecture and "drive" the scheduler. As a result, the programmer can easily try and evaluate different gathering strategies. More than a mere scheduling model, we propose a scheduling experimentation platform.

In this article, we first present the main existing approaches that exploit hierarchical machines, then we propose two new models describing application tasks and hierarchical levels of the machine, as well as a scheduler that takes advantage of them. Some implementation details and evaluation results are given before concluding.

2. EXPLOITING HIERARCHICAL MACHINES

Nowadays, multiprocessor machines like NUMAs with multithreaded multicores are increasingly difficult to exploit. Several approaches have been considered.

2.1 Predetermined distribution and scheduling

For very regular problems, it is possible to determine a task schedule and a data distribution that are suited to the target machine and its hierarchical levels. The application just needs to get the system to apply that schedule and that distribution, and excellent (if not optimal) performance can be obtained. The PaStiX[12] large sparse linear systems solver is a good example of this approach. It first launches a simulation of the computation based on models of BLAS operators and communications on the target architecture. Then it can compute a static schedule of block-computations and communications.

So as to enforce these scheduling strategies, many systems (Aix, Linux, Solaris, Windows, ...) allow process threads to be bound to processor sets, and memory allocations to be bound to memory nodes. Provided that the machine is dedicated to the application, the thread scheduling can be fully controlled by binding exactly one thread to each processor. To perform task switching, mere explicit context switches may be used: threads are only used as execution flow holders.

2.2 Opportunist distribution and scheduling

Greedy algorithms (called Self-Scheduling (SS) [27]) are dynamic, flexible and portable solutions for loop parallelization. Whatever the target machine, a Self-Scheduling algorithm takes care of both thread scheduling and data distribution. Operating systems schedulers are based on these algorithms.

They basically use a single list of ready tasks from which the scheduler just picks up the next thread to be scheduled. Hence the workload is automatically distributed between processors. For each task, the last processor on which it was scheduled is recorded, so as to try to reschedule it on the same processor as much as possible to avoid cache misses. These techniques are used in the Linux 2.4 and Windows 2000 [25] operating systems. However, a unique thread list for the whole machine is a bottleneck, particularly when the machine has many processors.

To avoid such contention, Guided Self-Scheduling (GSS) [22] and Trapezoid Self-Scheduling (TSS) [30] algorithms make each processor take a whole part of the total work when they are idle, raising the risk of imbalances. AFfinity Scheduling (AFS) [15] and Locality-based Dynamic Scheduling (LDS) [14] algorithms use a per-processor task list. Whenever idle, a processor will steal work from the least loaded list, for instance. These latter algorithms are used by current operating systems (Linux 2.6 [1], FreeBSD 5.0 [24], Cellular Irix [33]). They also add a few rebalance policies: new processes are charged to the least loaded processor, for instance.

However, contention appears quickly with an increased number of processors, particularly on NUMA machines. Wang *et al.* propose a Clustered AFfinity Scheduling (CAFS) [31] algorithm which groups p processors in groups of $\sqrt{p}$. Whenever idle, rather than looking around the whole machine, processors steal work from the least loaded processor of their group, hence getting better localization of list accesses. Moreover, by aligning groups to NUMA nodes, data distribution is also localized. Finally, the Hierarchical AFfinity Scheduling (HAFS) (Wang *et al.* [32]) algorithm lets any idle group steal work from the most loaded group. This latter approach is being considered for latest NUMA-aware developments of operating systems such as Linux 2.6 and FreeBSD.

2.3 Negotiated distribution and scheduling

There are intermediate solutions between predetermined and opportunist scheduling. Some language extensions such as OpenMP [16], HPF (*High Performance Fortran*) [26] or UPC (*Unified Parallel C*) [5] let one achieve parallel programming by simply annotating the source code. For instance, a `for` loop may be annotated to be automatically parallelized. An HPF matrix may be annotated to be automatically split into rather independent domains that will be processed in parallel.

The distribution and scheduling decisions then belong to the compiler. To do this, it adds code to query the execution environment (the number of processors for instance) and compiles the program in a way generic enough to adapt to the different parallel architectures. In particular, it will have to handle threads for parallelized loops or distributed computing, and even handle data exchange between processors (in the case of distributed matrices of HPF). To date, expressiveness is limited mostly to "`Fork-Join`" parallelism, which means, for instance, that the programmer can not express imbalanced parallelism.

Programmers may also directly write applications that are able to adapt themselves to the target machine at runtime. Modern operating systems provide full information about the architecture of the machine (user-level libraries are available: `lgroup` [29] for Solaris or `numa` [1] for Linux). The application can then not only get the number of processors, but also get the NUMA nodes hierarchy, their respective number of processors and their memory sizes. Those systems also let the application choose the memory allocation policy (specific memory node, *first touch* or *round robin*) and bind threads to CPU sets. Thus, the application controls threads and memory distribution, but it is then in charge of balancing threads between processors.

2.4 Discussion

We chose to classify existing approaches into three categories. The *predetermined* category gives excellent performance. But it is portable only if the problem is regular, *i.e.,* its solving time depends on the data structure and not on the data itself. The *opportunist* approach scales well, but does not take task affinities into account, and thus, on average, does not get excellent performance. The *negotiated* approach lets the application adapt itself to the underlying machine, but requires rewriting of some parts of the scheduler in order to be flexible.

Our proposal is a mix between negotiated and opportunist approaches. We will give the programmers means to dynamically describe how their applications behave, and use this information to guide a generic opportunist scheduler.

3. PROPOSAL: AN APPLICATION-GUIDED SCHEDULER

Our proposal is based on a collaboration between the application and its execution environment.

3.1 Bubbles modeling the application structure

The application is asked to model the general layout of its threads in terms of nested sets called **bubbles**[1].

Figure 1 shows such a model: the application groups threads into pairs, along with a communication thread (priorities will be discussed later). The concept of bubbles can be understood as a *coset with respect to a specific affinity relation*, and bubble nesting expresses refinement of a relation by another one. Indeed, several affinity relations can be considered, for instance:

Data sharing It is a good idea to group threads that work on the same data so as to benefit from cache effects, or at least to avoid spreading the data throughout the NUMA nodes thereby incurring the NUMA factor penalty.

Collective operations It can be beneficial to optimize the scheduling of threads which are to perform collective operations such as a synchronization barrier, which ensures that all involved threads have reached the barrier before they can continue executing.

SMT Many attempts were made to address thread scheduling on Simultaneous Multi-Threading (SMT) processors, mostly by detecting affinities between threads at runtime [4, 17]. Indeed, in some cases, pairs of threads may be able to efficiently exploit the SMT technology: they can run in parallel on the logical processors of the same physical processor without interfering. If the programmer knows that some pairs of threads can work in such *symbiosis*, he can express this relation.

Other relations may be possible to express parallelism, sequentiality, preemption, etc. Yet, blindly expressing these relations may also be detrimental: Bulpin and Pratt show performance loss [3] on SMT processors due to frequent cache misses for instance; Antonopoulos *et al.* also show performance loss [2] when not taking the SMP bus bandwidth limit into account. But the programmer may try and test different refinements of the relations and thus experimentally reveal how the threads of an application should be related.

In order to cope with the emerging multiprocessor networks of the 1980's, Ousterhout [21] proposed to group data and threads by affinity into *gangs*. These gangs hold a fixed number of threads which are to be launched at the same time on the same machine of the network: this is called *Gang Scheduling*. However, processors may be left idle because a single machine can only run one gang at a time, even if it is "small". Feitelson *et al.* [8] propose a hierarchical control of the processors so as to execute several gangs on the same machine. Our approach is actually a generalization of this approach.

[1] In a way relatively similar to some communication libraries such as MPI, which ask the application to specify *communicators*: groups of machines which will communicate.

3.2 Task lists modeling the computing power structure

According to Dandamudi and Cheng [6], a hierarchy of task lists generally brings better performance than simple per-processor lists. This is why two-level list schedulers have been developed [9, 20]. Moreover, it makes task binding to processor sets easier. In a manner similar to Nikolopoulos *et al.*'s Nano-Threads list hierarchy [19], we have taken up and generalized this point of view.

Indeed, we model hierarchical machines by a hierarchy of task lists. Each component of each level of the hierarchy of the machine has one and only one task list. Figure 2 shows a hierarchical machine and its model. The whole machine, each NUMA node, each core, each physical SMT processor and each logical SMT processor has a task list.

For a given task, the list on which it is inserted expresses the scheduling area: if the task is on a list associated with a physical chip, it will be allowed to be run by any processor on this chip; if it is placed on the global list, it will be allowed to be run by any processor of the machine.

3.3 Putting both models together: a bubble scheduler

Once the application has created bubbles, threads and bubbles are just "tasks" that the execution environment distributes on the machine.

3.3.1 Bubble evolution

As Figure 3 shows, the goal of a bubble is to hold tasks and bring them to the level where their scheduling will be most efficient. For this, the bubble goes down through lists to the *wanted* hierarchical level. It then "bursts", *i.e.* held threads and bubbles are released and can be executed (or go deeper). The list of held tasks is recorded, for a potential later regeneration (see Section 3.3.3). The main issue is how to specify the right bursting level of a bubble.

In the long run, once we get good heuristics for a bubble scheduler, specifying such a parameter will no longer be necessary. For now, the goal is to provide an experimental platform for developing schedulers, and hence allow this parameter to be tuned by the scheduler developers. They can favor task affinity with the risk of making the load balance difficult (by setting deep bursting levels) or on the contrary favor processor use (by setting high bursting levels).

3.3.2 Priorities

We choose to let the application attach integer priorities to tasks. When a processor looks for a task to be scheduled, it searches through the lists that "cover" this processor, from the most *local* one (*i.e.* on low levels) to the most *global* one, looking for a task with highest priority. It will then schedule that task, even if less prioritized tasks remain on more local lists.

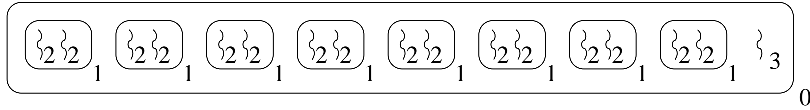
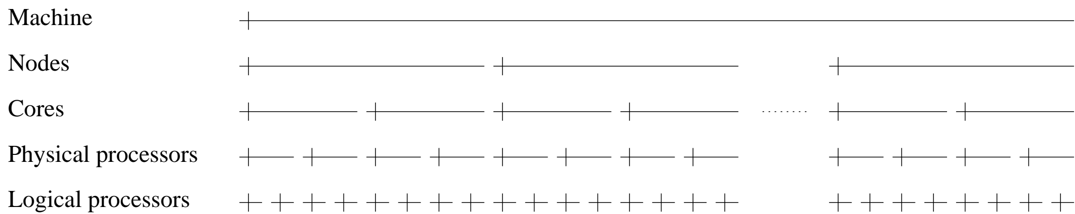


Figure 1: Bubble example, with priorities: thread pairs that have a higher priority than the bubbles holding them, and a highly prioritized thread.

(a) A NUMA of HyperThreaded multicores.



(b) Model with task lists.

Figure 2: A high-depth hierarchical machine and its model.

Figure 1 shows an example using priorities. In this example, bubbles holding computing threads are *less* prioritized than the threads. Consequently, a bubble will burst only if every thread of the previously burst bubbles has terminated, or if there are not enough of them to occupy all the processors. This results in some *Gang scheduling* which automatically occupies all the processors.

3.3.3 Bubble regeneration

Bubbles are automatically distributed by the scheduler over the different levels of task lists of the machine, hence distributing threads on the whole machine while taking affinity into account. However, it is possible that a whole thread group has far less work than others and terminates before them, leaving idle the whole part of the machine that was running it.

To *correct* such imbalance, some bubbles may be regenerated and moved up. Idle processors would then move some of them down on their side and have them re-burst there, getting a new distribution suited to the new workload while keeping affinity intact.

To *prevent* such imbalances, bubbles may periodically be regenerated[2]: each bubble has its own *time slice* after which its threads are preempted and the bubble regenerated.

In the case of Figure 1, the preemption mechanism is extended to *Gang Scheduling*: whenever a bubble is regenerated (because its time slice expired), it is put back at the end of the task list while another bubble is burst to occupy the resulting idle processors.

3.4 Discussion

Bubbles give programmers the opportunity to express the structure of their application and to guide the scheduling of their threads in a *simple*, *portable* and *structured* way. Since the roles of processors and other hierarchical levels are not predetermined, the scheduler still has some degrees of freedom and can hence use an opportunist strategy to distribute tasks over the whole machine. By taking into account any irregularity in the application, this scheduler significantly enhances the underlying machine exploitation. Such pre-

[2]In a way similar to UNIX system thread preemption.

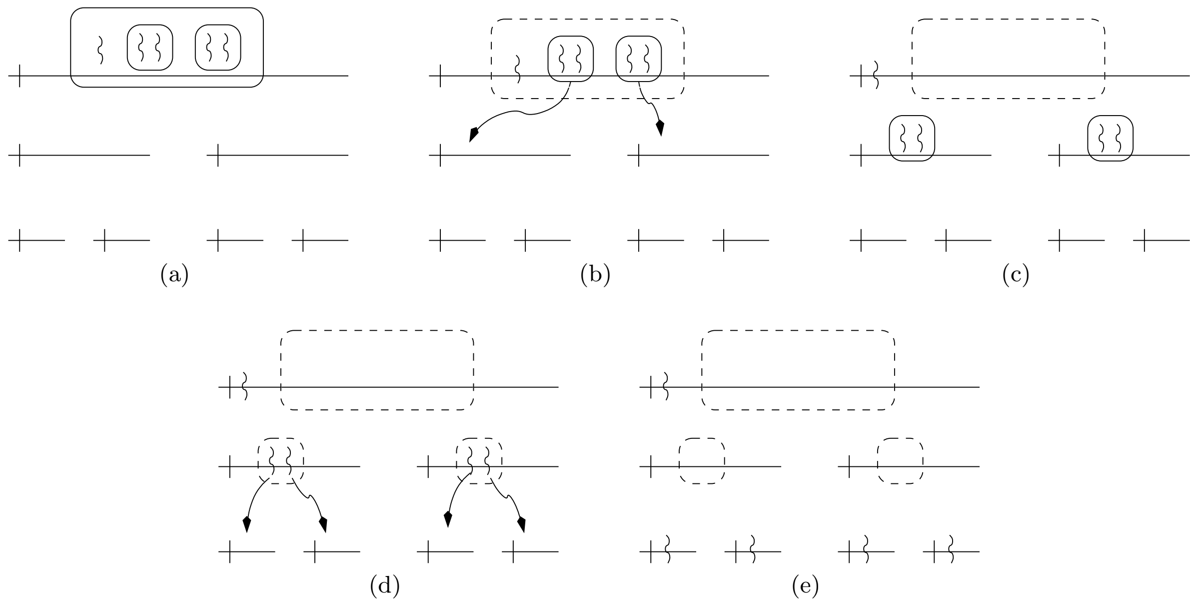


Figure 3: Bubble evolution. (a) The outermost bubble starts on the general list. (b) It bursts, releasing a thread (which can immediately be scheduled on any processor) and two sub-bubbles which can go down through the hierarchy. (c) Going down achieved. (d) Both sub-bubbles burst, releasing two threads each. (e) Threads are distributed appropriately and can start in parallel.

```
marcel_t thread1, thread2;
marcel_bubble_t bubble;

marcel_bubble_init(&bubble);
marcel_create_dontsched(&thread1, NULL, fun1, para1);
marcel_create_dontsched(&thread2, NULL, fun2, para2);
marcel_bubble_inserttask(&bubble, thread1);
marcel_wake_up_bubble(&bubble);
marcel_bubble_inserttask(&bubble, thread2);
```

Figure 4: Bubble creation example: threads are created without being started, then they are inserted in the same bubble.

ventive rebalancing techniques may still have side effects and lead to pathological situations (ping-ponging between tasks, useless bubble migration just before termination, etc.).

4. IMPLEMENTATION DETAILS

Marcel [18, 7] is a two-level thread library: in a way similar to manual scheduling (see section 2.1), it binds one kernel-level thread on each processor and then performs fast user-level context switches between user-level threads, hence getting complete control on threads scheduling[3] in userland without any further help from the kernel. Our proposal was implemented within Marcel's user threads scheduler.

Figure 4 shows an example of using the interface to build and launch a bubble containing two Marcel threads.

The Marcel scheduler already had per-processor thread lists, so that integrating bubbles within the library did not need a thorough rewriting of the data structures. The scheduler code was modified to implement list hierarchy, bubble evolution and to take priorities (described in Section 3.3.2) into account.

So as to avoid contention, there is no global scheduling: processors just call the scheduler code themselves whenever they preempt (or terminate) a thread. The scheduler finds some thread that is ready to be executed by the processor. We added bubble management there: while looking for threads to execute, the scheduler code now also tries to "pull down" bubbles from high list levels and make them burst on a more local level. Getting an efficient implementation is complex, as explained below.

Given a processor, two passes are done to look for the task (thread or bubble) with maximum priority among all the tasks of the lists "covering" that processor. The first pass quickly finds the list containing the task with the highest priority, without the need of a lock. That list and the list holding the currently running task are locked[4]. A second pass is then used to check that the selected list still has a task of this priority, in case some other processor took it in the meantime. If the selected task is a thread, it is scheduled; otherwise it is a bubble that the processor deals with appropriately (going down / bursting). The implementation time-complexity is linear with respect to the number of hierarchical levels of the machine.

Regenerating a bubble is also a difficult operation. Replacing threads in a given bubble requires removing all of them

[3] We suppose that no other application is running, and neglect system daemons wake-ups.

[4] By convention, locking lists is done by locking high-level lists first, and for a given level, according to the level elements identifiers.

	Yield			Switch		
	ns	cycles	%	ns	cycles	%
Marcel (original)	186	495	69	84	223	31
Marcel bubbles	250	665	63	148	395	37
NPTL (Linux 2.6)	672	1790	31	1488	3930	69

Table 1: Cost of the modified Marcel scheduler for searching lists, compared to other schedulers. Yield: list search only, Switch: synchronization and context switch.

from the task lists, except threads being executed. Those threads go back in the bubble by themselves when the processors executing them call the scheduler. Eventually, the last thread closes the bubble and moves it up to the list where it was initially released by the bubble holding it.

5. PERFORMANCE EVALUATION

Our algorithm has some cost, but increases performance thanks to the resulting localization.

5.1 Bubble scheduler cost

We measured the performance impact of our implementation on the Marcel library running on a 2.66 GHz Pentium IV Xeon. Searching through lists has a reasonable cost, and our scheduler execution times are good compared to the Linux thread libraries LinuxThread (2.4 kernel) and NPTL (2.6 kernel), see Table 1.

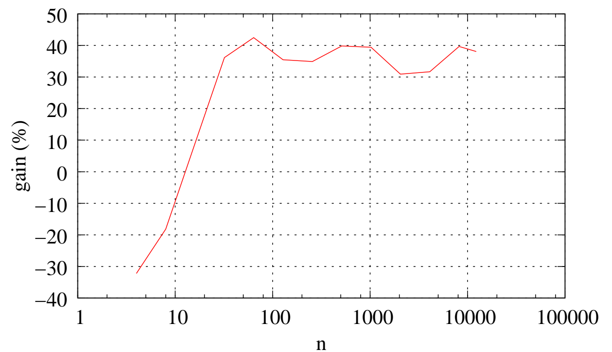
Creation and destruction of a bubble holding a thread does not cost much more than creation and destruction of a simple thread: the cost increases from 3.3μs to 3.7μs.

Test-case examples of recursive creation of threads, such as divide-and-conquer *Fibonacci* show that the cost of systematically adding bubbles that express the natural recursion of threads creations is quickly balanced by the localization that they bring: Figure 5 shows that performance is affected when only a few threads are created, while on a HyperThreaded Bi-Pentium IV Xeon, the performance gain stabilizes at around 30 to 40% with 16 threads; on a NUMA 4×4 Itanium II, the gain is 40% with 32 threads and gets up to 80% with 512 threads.

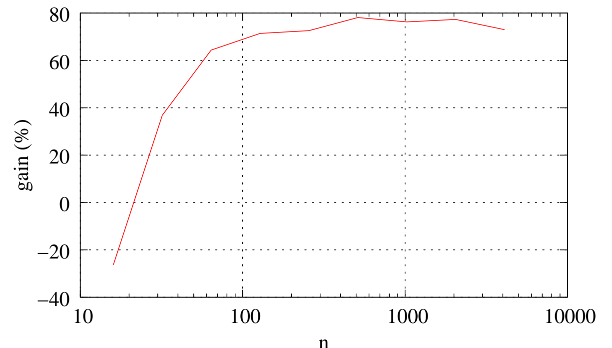
5.2 A real application

Marc Pérache [23] used our scheduler in a comparison of the efficiency of various scheduling strategies for *heat conduction and advection* simulations. Results may be seen in Table 2. The target machine is a ccNUMA Bull NovaScale with 16 Itanium II processors and 64 GB of memory, distributed among 4 NUMA nodes. For a given processor, accessing the memory of its own node is about 3 times faster than accessing the memory of another node. The applications perform cycles of fully parallel computing followed by global hierarchical communication barrier.

In the *simple* version, the mesh is split into as many stripes as the number of processors, and an opportunist schedule is used. The *bound* version binds them to processors in a non-portable way. This gets far better performance: each thread remains on the same node, along with its data. Our proposal lets the application query Marcel about the number of NUMA nodes and processors and then automatically build



(a) 2 HyperThreaded Pentium IV Xeon



(b) 4×4 Itanium II

Figure 5: Performance gain brought by adding bubbles to the *fibonacci* test-case.

	Conduction		Advection	
	Time (s)	Speedup	Time (s)	Speedup
Sequential	250.2		16.13	
Simple	23.65	10.58	1.77	9.11
Bound	15.82	15.82	1.30	12.40
Bubbles	15.84	15.80	1.30	12.40

Table 2: Conduction performance depending on the approach.

bubbles according to the hierarchy of the machine (hence 4 bubbles of 4 threads in this example). It gets performance very similar to those of the *bound* version.

As can be seen, the use of bubbles attained performance close to that which may be achieved with a “handmade” thread distribution, but in a portable way.

These applications are a simple example in which the workload is balanced between stripes. The use of bubbles simply allowed it to automatically fit the architecture of the machine. However, in the future these applications will be modified to benefit from Adaptive Mesh Refinement (AMR) which increases computing precision on interesting areas. This will entail large workload imbalances in the mesh both *at runtime* and *according to the computation results*. It will hence be interesting to compare both development time and execution time of handmade-, opportunist-, and bubble-scheduled versions.

6. CONCLUSION

Multiprocessor machines are getting increasingly hierarchical. This makes task scheduling extremely complex. Moreover, the challenge is to get a scheduler that will perform "good" task scheduling on any multiprocessor machine with an arbitrary hierarchy, only guided by *portable* scheduling hints.

In this paper, we presented a new mechanism making significant progress in that direction: the bubble model lets applications express affinity relations of varying degrees between tasks in a portable way. The scheduler can then use these hints to distribute threads.

Ideally, the scheduler would need no other information to perform this. But practically speaking, writing such a scheduler is difficult and will need many experiments to be tuned. In the meantime, the programmer can use stricter guiding hints (indicating bubble bursting levels, for instance) so as to experiment with several strategies.

Performance observations on several test-cases are promising, far better than what opportunist schedulers can achieve, and close to what predetermined schedulers get. These observations were obtained on several architectures (Intel PC SMP, Itanium II NUMA).

This work opens numerous future prospects. In the short term, our proposal will be included within test-cases of real applications of CEA that run on highly hierarchical machines, hence stressing the bubble mechanism power. It will then be useful to develop analysis tools based on tracing the scheduler at runtime, so as to check and refine scheduling strategies. It will also be useful to let the programmer set other attributes than just priorities, and thus influence the scheduler: "strength" of the bubble (which expresses the amount of affinity that the bubble represents), preemptibility, some notion of amount of work, ...

In the longer term, the goal is to provide a means of expression powerful and portable enough for the application to obtain an automatic schedule that gets close to the "optimal" whatever the underlying architecture. It could also be useful to provide more powerful memory allocation functions, specifying which scope of tasks (a bubble for instance) will use the allocated area.

7. REFERENCES

[1] *Linux Scalability Effort.* http://lse.sourceforge.net/.

[2] C. Antonopoulos, D. Nikolopoulos, and T. Papatheodorou. Scheduling algorithms with bus bandwidth considerations for SMPs. In *2003 International Conference on Parallel Processing*, pages 547–554. IEEE, Oct. 2003.

[3] J. R. Bulpin and I. A. Pratt. Multiprogramming performance of the pentium 4 with hyper-threading. In *Third Annual Workshop on Duplicating, Deconstructing and Debunking (WDDD2004) (at ISCA'04)*, pages 53–62, June 2004.

[4] J. R. Bulpin and I. A. Pratt. Hyper-threading aware process scheduling heuristics. In *2005 USENIX Annual Technical Conference*, pages 103–106, 2005.

[5] W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to upc and language specification. Technical Report CCS-TR-99-157, George Mason University, May 1999. http://upc.gwu.edu/.

[6] S. Dandamudi and S. Cheng. Performance impact of run queue organization and synchronization on large-scale NUMA multiprocessor systems. *Systems Architecture*, 43:491–511, 1997.

[7] V. Danjean. *Contribution à l'élaboration d'ordonnanceurs de processus légers performants et portables pour architectures multiprocesseurs.* PhD thesis, École Normale Supérieure de Lyon, Dec. 2004.

[8] D. G. Feitelson and L. Rudolph. Evaluation of design choices for gang scheduling using distributed hierarchical control. *Parallel and Distributed Computing*, 35:18–34, 1996.

[9] A. Fukuda, R. Fukiji, and H. Kai. Two-level processor scheduling for multiprogrammed NUMA multiprocessors. In *Computer Software and Applications Conferences*, pages 343–351. IEEE, 1993. fukuda@csce.kyushu-u.ac.jp.

[10] E. Hagersten and M. Koster. WildFire: A scalable path for SMPs. In *The Fifth International Symposium on High Performance Computer Architecture*, Jan. 1999. Sun Microsystems, Inc.

[11] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufman, third edition, 2003.

[12] P. Hénon, P. Ramet, and J. Roman. PaStiX: A parallel sparse direct solver based on a static scheduling for mixed 1d/2d block distributions. In *Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, pages 519–527. Springer-Verlag, Jan. 2000.

[13] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA highly scalable server. In *24th International Symposium on Computer Architecture*, pages 241–251, June 1997. Silicon Graphics, Inc.

[14] H. Li, S. Tandri, M. Stumm, and K. C. Sevcik. Locality and loop scheduling on NUMA multiprocessors. In *International Conference on Parallel Processing*, volume II, pages 140–127, Aug. 1993.

[15] E. Markatos and T. Leblanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. *Parallel and Distributed Systems*, 5(4):379–400, Apr. 1994.

[16] T. Mattson and R. EigenMann. OpenMP: An API for writing portable SMP application software. In *SuperComputing 99 Conference*, Nov. 1999.

[17] R. L. McGregor, C. D. Antonopoulos, and D. S. Nikolopoulos. Scheduling algorithms for effective thread pairing on hybrid multiprocessors. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Papers*, page 28.1, Washington, DC, USA, 2005. IEEE Computer Society.

[18] R. Namyst. *PM2: un environnement pour une conception portable et une exécution efficace des applications parallèles irrégulières*. PhD thesis, Univ. de Lille 1, Jan. 1997.

[19] D. S. Nikolopoulos, E. D. Polychronopoulos, and T. S. Papatheodorou. Efficient runtime thread management for the nano-threads programming model. In *Second IEEE IPPS/SPDP Workshop on Runtime Systems for Parallel Programming*, volume 1388, pages 183–194, Orlando, FL, Apr. 1998.

[20] H. Oguma and Y. Nakayama. A scheduling mechanism for lock-free operation of a lightweight process library for smp computers. *Conference on Parallel and Distributed Systems*, pages 235–242, July 2001.

[21] J. K. Ousterhout. Scheduling techniques for concurrent systems. In *Third International Conference on Distributed Computing Systems*, pages 22–30, Oct. 1982.

[22] C. Polychronopoulos and D. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *Transactions on Computers*, 36(12):1425–1439, Dec. 1987.

[23] M. Pérache. Nouveaux mécanismes au sein des ordonnanceurs de threads pour une implantation efficace des opérations collectives sur machines multiprocesseurs. In *Rencontres francophones en Parallélisme, Architecture, Système et Composant (RenPar 16)*, Mar. 2005.

[24] J. Roberson. ULE: A modern scheduler for FreeBSD. Technical report, The FreeBSD Project, `jeff@FreeBSD.org`, 2003.

[25] M. Russinovich. Inside the windows NT scheduler, Part 2. *Windows IT Pro*, 303, July 1997.

[26] R. Schreiber. An introduction to HPF. In *The Data Parallel Programming Model: Foundations, HPF Realization, and Scientific Applications*, pages 27–44. Springer-Verlag, 1996.

[27] P. Tang and P.-C. Yew. Processor self-scheduling for multiple nested parallel loops. In *Proceedings 1986 International Conference on Parallel Processing*, pages 528–535, Aug. 1986.

[28] Bull. *Bull NovaScale servers.* `http://www.bull.com/novascale/`.

[29] Sun microsystems. *Solaris Memory Placement Optimization (MPO).* `http://iforce.sun.com/protected/solaris10/adoptionkit/tech/mpo/mpo_man.html`.

[30] T. Tzen and L. Ni. Trapezoid self-scheduling: A practical scheduling scheme for parallel compilers. *Parallel and Distributed Systems*, 4(1):87–98, Jan. 1993.

[31] Y.-M. Wang, H.-H. Wang, and R.-C. Chang. Clustered affinity scheduling on large-scale NUMA multiprocessors. *Systems Software*, 39:61–70, 1997.

[32] Y.-M. Wang, H.-H. Wang, and R.-C. Chang. Hierarchical loop scheduling for clustered NUMA machines. *Systems and Software*, 55:33–44, 2000.

[33] S. Whitney, J. McCalpin, N. Bitar, J. L. Richardson, and L. Stevens. The SGI origin software environment and application performance. In *COMPCON 97*, pages 165–170, San Jose, California, 1997. IEEE.