Designing Efficient Runtime Systems for the Multicore Era

Projet ANR-05-CICG-001-03

Livrable 8.2 (T0 + 36)

Raymond Namyst
Équipe-Projet Runtime
INRIA Bordeaux - Sud-Ouest — LaBRI — Université Bordeaux1
351 cours de la libération — 33405 Talence cedex, France

January 30, 2009

Contents

1 Motivations **3**

2 Building Portable Thread Schedulers for Hierarchical Multiprocessors: the BubbleSched Framework **5**

2.1 Introduction 5
2.2 On the Design of Thread Schedulers 6
2.2.1 What Input Can a Scheduler Expect? 7
2.2.2 On the Importance of Scheduling Guidance 7
2.2.3 Towards a Toolbox for Developing Thread Schedulers . . 8
2.3 BubbleSched: A Framework for Building Portable Schedulers . . 9
2.3.1 An API for Writing Scheduling Strategies 9
2.3.2 Implementation Examples of Scheduling Algorithms . . . 10
2.4 ForestGomp: an Implementation of GNU OpenMP for Hierarchical Machines 12
2.4.1 Scheduling OpenMP Applications Featuring Nested, Irregular Parallelism 12
2.4.2 Generating Bubbles Out of OpenMP Parallel Sections . . 14
2.4.3 A Scheduling Strategy Suited to OpenMP Nested Parallelism 14
2.4.4 A scheduling policy guided by affinity hints 16
2.5 Implementation 18
2.5.1 Implementation over a two level thread library 18
2.5.2 Micro-Benchmarks 19
2.6 Evaluation 20
2.6.1 LU Factorization 20
2.6.2 BT-MZ 22
2.6.3 MPU: An higly irregular application 25
2.7 Related Work 27
2.8 Conclusion 29

3 StarPU : a unified platform for task scheduling on heterogeneous multicore architectures **31**

3.1 Introduction 31
3.2 A unified runtime system 32

3.2.1 A library unifying data manipulation 32
3.2.2 A unified execution model for heterogeneous multicore architectures 34
3.3 A uniform queue-based interface to design scheduling policies . . 34
3.3.1 Integrating scheduling into our uniform execution model . 35
3.3.2 Enriching our uniform execution model with optional scheduling hints 36
3.4 Experimental validation 37
3.4.1 Experimental environment 37
3.4.2 Benchmarks 37
3.4.3 Scheduling strategies implemented using our interface . . 38
3.4.4 Impact of the design of the queues 39
3.4.5 Policies based on performance models 40
3.4.6 Taking advantage of heterogeneity 41
3.5 Related works 42
3.6 Conclusion and future works 43

4 Software Availability **44**

Chapter 1

Motivations

With the beginning of the new century, computer makers have initiated a long term move of integrating more and more processing units, as an answer to the frequency wall hit by the technology. This integration cannot be made in a basic, planar scheme beyond a couple of processing units for scalability reasons. Instead, vendors have to resort to organize those processing units following some hierarchical structuration scheme. A level in the hierarchy is then materialized by small groups of units sharing some common local cache or memory bank. Memory accesses outside the locality of the group are still possible thanks to bus-level consistency mechanisms but are significantly more expensive than local accesses, which, by definition, characterizes NUMA architectures.

Thus, the task scheduler must feed an increasing number of processing units with work to execute and data to process while keeping the rate of penalized memory accesses as low as possible. False sharing, ping-pong effects, data vs task locality mismatches, and even task vs task locality mismatches between tightly synchronizing activities are examples of the numerous sources of overhead that may arise if threads and data are not distributed properly by the scheduler. To avoid these pitfalls, the scheduler therefore needs accurate information both about the computing platform layout it is running on and about the structure and activities relationships of the application it is scheduling.

We believe it is important to expose domain-specific knowledge semantics to the various software components in order to organize computation according to the application and architecture: the whole software stack, from the application to the scheduler, should be involved in the parallelizing, scheduling and locality adaptation decisions by providing useful information to the other components. Unfortunately, most operating systems only provide a poor scheduling API that does not allow applications to transmit valuable *hints* to the system.

This is why we investigate new approaches in the design of thread schedulers, focusing on high-level abstractions to both model hierarchical architectures and describe the structure of applications' parallelism. In particular, we have introduced the *bubble* scheduling concept [44] that helps to structure relations between threads

in a way that can be efficiently exploited by the underlying thread scheduler. *Bubbles* express the inherent parallel structure of multithreaded applications: they are abstractions for grouping threads which "work together" in a recursive way.

Aside from greedily invading all these new cores, demanding HPC applications now throw excited glances at the appealling computing power left unharvested inside the graphical processing units. A strong demand is arising from the application programmers to be given means to access this power without bearing an unaffordable burden on the portability side. Efforts have already be made by the community in this respect but the tools provided still are rather close to the hardware, if not to the metal. Hence, we decided to launch some investigations on addressing this issue. In particular, we have designed a programming environment named StarPU that enables the programmer to offload tasks onto such heterogeneous processing units and gives that programmer tools to fit tasks to processing units capability, tools to efficiently manage data moves to and from the offloading hardware and handles the scheduling of such tasks all in an abstracted, portable manner.

Chapter 2

Building Portable Thread Schedulers for Hierarchical Multiprocessors: the BubbleSched Framework

2.1 Introduction

Nowadays mid-range and large scale shared-memory computers are based on a non uniform memory access (NUMA) interconnect of processing units based on chip multiprocessing (CMP) and simultaneous multithreading (SMT) technologies. Exploiting these machines efficiently is a real challenge, and a thread scheduler is faced with dilemmas when trying to take into account the memory hierarchy and the CPU utilization simultaneously. On NUMA machines for instance, threads should generally be scheduled as close to their data as possible, but bandwidth-consuming threads should rather be distributed over different chips. The core scheduler of operating systems can often be influenced, but it misses the precise application behavior: for instance, adaptively-refined meshes entail very irregular and unpredictable behavior.

As quoted by Gao *et al.* [22], it is important to expose domain-specific knowledge semantics to the various software components in order to organize computation according to the application and architecture. Indeed, the whole software stack, from the application to the scheduler, should be involved in the parallelizing, scheduling and locality adaptation decisions by providing useful information to the other components. For instance, in OpenMP frameworks, the information extracted by the compiler (about memory affinity and adherence to the same parallel section) can be very useful for the guidance of task/thread scheduling. On the other hand, it is very important to rely on architecture specific constraints when making these scheduling decisions. A tight interaction between the OpenMP stack and the underlying hardware-aware scheduler is thus required. The most delicate point,

in particular when dealing with irregular applications, is to exploit this knowledge at runtime (during the whole execution time) so as to maintain a good balancing of threads when events arise (task termination, creation of new embedded parallel sections, blocking synchronization, etc.).

In this aim we have designed and implemented a hierarchical threading library able to follow/obey scheduling directives and advices in a very powerful manner [43]. Scheduling information (affinity, group membership) is attached to bubbles, which are abstractions that can recursively group threads or bubbles sharing common properties. The *bubble* scheduling concept helps to structure relations between threads in a way that can be efficiently exploited by the underlying thread scheduler. *Bubbles* express the inherent parallel structure of multithreaded applications: they are abstractions for grouping threads which "work together" in a recursive way. On the hardware side, hierarchical machines are modelled with a hierarchy of runqueues. Each component of each hierarchical level of the machine is represented by one runqueue: one per logical processor, one per core, one per chip, one per NUMA node, and one for the whole machine. The first *proof-of-concept* implementation of our bubble scheduler was featuring a generic hard-coded scheduler based on a hierarchical *Self-Scheduling* algorithm. However, applications may have different scheduling requirements and thus may attach different semantics to *bubbles*, enforcing memory affinity or emphasizing a high frequency of global synchronization operations for instance. Obviously, no generic scheduler can meet all these needs. Therefore we have designed the *BubbleSched* framework to ease the development and the evaluation of customized, high-level thread schedulers.

The rest of this chapter is organized as follows. Section 2.2 discusses the main issues of designing a scheduler for hierarchical multiprocessor machines. The need for a framework to help the scheduler designer to easily implement and observe a scheduling algorithm is argued in Section 2.2.3. Our *BubbleSched* framework is then presented in Section 2.3, and some implementation details and some micro-benchmarks are given in Section 2.5. Then we show how we have designed *Forest-GOMP* , an extension to the GNU OpenMP runtime system [1] that relies on our framework. In Section 2.6, we give some performance figures that show the relevance of our approach. Before concluding, we discuss related work in Section 2.7.

2.2 On the Design of Thread Schedulers

Designing a thread scheduler for hierarchical machines is a complex job because a compromise must be found between a lot of constraints: favoring affinities between threads and memory, taking advantage of all computational power, reducing synchronization cost, etc. Furthermore, the best trade-off often depends on the target application.

2.2.1 What Input Can a Scheduler Expect?

To make appropriate decisions at execution time, a thread scheduler can combine a number of parameters to evaluate the goodness of each potential scheduling action. These parameters can be collected from several places, at different times.

At runtime, some useful knowledge about the target machine can be discovered. The scheduler can not only get the number of processors but also the architecture hierarchy: how processors and memory banks are connected, how cache levels are shared between processors, etc. Moreover, indication about how well the threads are using the underlying processors can be fetched from performance counters. Some NUMA chips can also report the ratio of remote memory accesses. The scheduler can hence check whether threads and data are properly localized, or enable automatic migration policies [30, 27].

The compiler can also provide information about the application behavior. Data access patterns may be analyzed [39] and used to choose runtime allocation policies. An OpenMP compiler can sometimes accurately estimate the amount of data shared by the threads involved in the same parallel section.

Last but not least, *programmers* can also provide relevant information about the application behavior: how threads will mostly access data, what threads are I/O-bounded, etc.) Such information can help the scheduler to find a good trade-off regarding the co-location of threads and data, and more generally can help to determine the combination of scheduling policies that will perform best.

To sum it up, a lot of information is at the scheduler's disposal or can be collected at run time. Programmers can sometimes even provide additional scheduling hints. All we need is a reasonable way to pass this information from the programmer to the scheduler.

2.2.2 On the Importance of Scheduling Guidance

Opportunistic policies based on Self-Scheduling algorithms [18] are the most natural approaches regarding thread scheduling. These algorithms use a centralized list of ready threads (FREEBSD 4, LINUX 2.4, WINDOWS 2K) or a distributed one (FREEBSD 5, IRIX, LINUX 2.6) associated to load balancing mechanisms. They basically use lists of ready threads, from which the scheduler just picks up the next thread to be scheduled, so that the workload is automatically distributed among processors. They are even heading for using a hierarchy of thread lists [45]. Such an approach scales and adapts easily to new workloads, but it does not use affinity information from the compiler or the programmer, and thus can not actually achieve excellent performance.

For very regular problems, a *predetermined approach* can be used. The idea is to compute *a priori* a good distribution of threads and data that will be enforced during execution. The machine being dedicated to the application, thread scheduling can be fully controlled by binding exactly one kernel thread to each processor. This approach, used in the PASTIX solver for sparse linear systems [24], gives

Figure 2.1: Expressing thread relationships: graphical and tree-based representations

excellent performance for regular problems, but as soon as the solving time depends on the data or intermediate results themselves, behavior prediction fails and performance may actually get worse than by using an opportunistic approach.

An intermediate approach between opportunistic and predetermined scheduling is based on *negotiation*. Some language extensions such as OPENMP, High Performance Fortran (HPF) or Unified Parallel C (UPC) let programmers write parallel applications by simply annotating the source code. The distribution and scheduling decisions then belong to the compiler. Such extensions get good performance by exploiting information from programmers, but the expressiveness is limited to "`Fork-Join`" decomposition schemes, and programmers can not express unbalanced parallelism for instance.

Negotiation yet appears to be a promising approach, because programmers just have to give suggestions and clues, sometimes indirectly, about the behavior of the threads. However, the runtime system has only little control over the operating system's thread scheduler. We believe that a good approach is to extend the scheduler interface so as to allow applications to transmit scheduling hints that will persist inside the scheduler during the lifetime of the threads. The challenge is to find a good trade-off between the expressiveness of the proposed mechanism and the efficiency of the additional treatments involved at run time.

2.2.3 Towards a Toolbox for Developing Thread Schedulers

In [43], we proposed a model that allows programmers to model the relationships between threads using nested sets called **bubbles**. Figure 2.1 illustrates this concept: four threads are grouped as pairs in bubbles (*e.g.* they work on the same data), which are themselves grouped along another thread in a larger bubble (*e.g.* they share less information). This lets express relations like data sharing, collective operations, or more generally a particular scheduling policy need (serialization, gang scheduling, etc.). We also automatically model hierarchical machines with a hierarchy of runqueues. Each component of each hierarchical level of the machine is represented by one runqueue: one per logical processor, one per core, one per chip, one per NUMA node, and one for the whole machine.

Our ground scheduler then uses a hierarchical Self-Scheduling algorithm. Whenever idle, a processor scans all runqueues that span it, and executes the first thread that is found, from bottom to top. For instance, if the thread is on a runqueue that

represents a chip, it may be run by any processor of this chip.

2.3 BubbleSched: A Framework for Building Portable Schedulers

To tackle the delicate issue of scheduling an application on hierarchical machines in an efficient, flexible and portable way, we propose a new platform for easily writing customized schedulers, based on our high-level *bubble* abstractions.

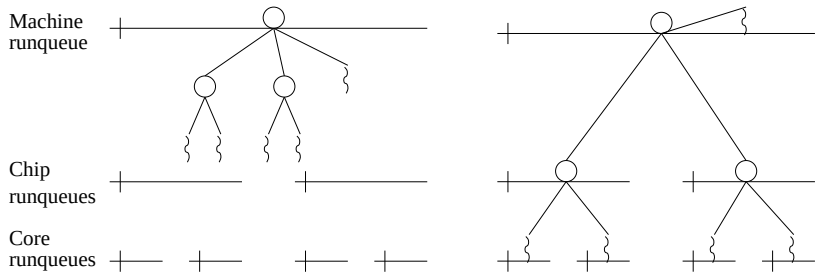
Our platform allows programmers of specialized scientific libraries or parallel programming environments to easily write thread schedulers for some given application classes on hierarchical multi-processor machines. It provides a high-level API for writing powerful and portable schedulers that manipulate threads grouped into bubbles, as well as tracing tools to help analyzing the dynamic behavior of these schedulers. Programmers can hence focus on algorithmic issues rather than on nasty technical details.

This not only permits easy deployment, but also allows a very tight cooperation with the application: the latter may even provide with some statistical or decision functions.

2.3.1 An API for Writing Scheduling Strategies

By default, bubbles (and hence the contained threads) will be placed on the machine runqueue upon start-up. In the previous example of Figure 2.1, running on a dual-dual-core machine, this leads to Figure 2.2(a): the bubble hierarchy is kept at the top of the hierarchy of runqueues. Such a distribution permits the use of all processors of the machine, since all of them have the opportunity to run any ready thread whenever they get idle. This does not however take affinities between threads and processors into account, since no relation between them is used. On the contrary, Figure 2.2(b) shows how threads can be spatially distributed in a very affinity-aware way, since here threads are just correctly distributed over processors. However, if some threads sleep, the corresponding processors become idle, hence resulting to a partial CPU usage.

We designed a whole programming interface [42] for easily manipulating bubbles and threads among the runqueues so as to achieve compromises between distributions of Figures 2.2(a) and 2.2(b). Threads and bubbles are equally considered as **entities**, while bubbles and runqueues are equally considered as **scheduling holders**, so that we end up with entities (threads or bubbles) that we can schedule on holders (bubbles or runqueues). Primitives are then provided for manipulating entities in holders. Runqueues can be accessed through vectors, and can be walked thanks to “father” and “son” pointers. Some functions permit to gather statistics about bubbles so as to take appropriate decisions. This includes for instance the total number of threads and the number of running threads, but also various information such as the accumulated expected and current CPU computation time or



(a) Good CPU usage, poor affinity care. (b) Distributed CPU usage, good affinity care.

Figure 2.2: Possible distributions of threads and bubbles among the machine.

memory usage, or the cache miss rates. To handle concurrency, one can use some fine-grain locking functions, but since bubble scheduling decisions are generally rather medium-term oriented, one can also just use a coarse-grain function which locks a whole sub-part of the machine (runqueues and bubbles). This permits to elaborate complex manipulations without having to care about locking. Once locked, one can eventually enumerate entities held by bubbles and runqueues, and redistribute them at will.

Writing a high-level scheduler actually reduces to writing some hook functions. The `bubble_schedule()` hook is called when the ground Self-Scheduler encounters a bubble during its search for the next thread to execute. The default implementation just looks for a thread in the bubble (or one of its sub-bubbles) and switches to it. The `bubble_tick()` hook is called when some time-slice for a bubble expires, and hence permits periodic operations on bubbles with a per-bubble notion of time. Of course, mere "daemon" threads can also be started for performing background operations. As a result, programmers may manipulate threads with a high level of abstraction by deciding the placement of bubbles on runqueues, or even temporarily putting some bubbles aside (by defining their own runqueues that the basic Self-Scheduler will not look at).

2.3.2 Implementation Examples of Scheduling Algorithms

Our programming interface, though quite simple, permits to develop a wide range of powerful "bubble schedulers". Let us give a few examples that illustrate different ways to exploit the functionalities provided by our platform.

Burst Scheduling

A first example of possible schedulers is our previously-described "burst algorithm" [43]: a Self-Scheduling algorithm "pulls" towards processors the bubbles, which "burst" (*i.e.* release their content on a runqueue) in an opportunistic way when they reach hierarchy levels given by programmers. For implementing this, when the `bubble_schedule()` hook is called on some processor, the bubble

is pulled down a bit towards that processor, and its content is released if needed. Within a few iterations, bubbles and threads get distributed as on Figure 2.2(b). Moreover, bubbles are periodically "regenerated" through the `bubble_tick()` hook: their original content is put back into them, and they are put back on the machine runqueue for a new distribution. This permits to dynamically adapt the distribution to new workloads while keeping affinities in mind. This quite simple algorithm was tested with *heat conduction and advection* simulations. The results in the early paper [43] show that this permits to get the same performance benefit as manual thread binding, but in a portable way.

Gang Scheduling

In the 1980's, OUSTERHOUT [33] proposed to group data and threads by affinity into *gangs*, and then not schedule threads on processors, but gangs on machines. For realizing such *gang scheduler*, we express gangs thanks to bubbles, and a "daemon thread" performs the scheduling, see the dozen lines of code on Figure 2.3. The daemon uses its own `nosched_rq` where it puts all bubbles (i.e. gangs) aside, except one, which is left on the main runqueue for some time, during which the basic Self-Scheduler can schedule the threads of the bubble. The result is as expected: time slices are equitably distributed between gangs, and then threads within gangs share their execution time within these time slices. Programmers can easily tinker with such program (change the periodicity, etc.) without a lot of technical knowledge. Locking is of course required, but since it is rarely executed, a simple coarse locking is efficient enough.

```
runqueue_t nosched_rq;
while(1) {
  /* Wait for next time slice */
  delay(timeslice);
  holder_lock(&main_rq);
  holder_lock(&nosched_rq);
  /* Put all entities aside */
  runqueue_for_each_entry(&main_rq,
&e) {
    get_entity(e);
    put_entity(e, &nosched_rq);
  }
  /* Put one entity on main
runqueue */
  if
(!runqueue_empty(&nosched_rq)) {
    e =
runqueue_entry(&nosched_rq);
    get_entity(e);
    put_entity(e, &main_rq);
  }
  holder_unlock(&main_rq);
  holder_unlock(&nosched_rq);
}
```

(a) Source code.

Figure 2.3: A gang scheduler

An interesting use of this gang scheduler is to emulate a network of virtual machines in a fair way on a single physical machine: each virtual machine is represented by a bubble which contains its thread. By running the gang scheduler in the background, we get fair execution: each machine gets time slices (the `timeslice` variable of Figure 2.3) during which its threads can run on the machine. The result are shown on Figure 2.3, where three busy-looping gangs are sharing a four-processor machine (gang 0 has 5 threads `[0-4]`, etc.) As expected, the machine is shared in a very fair way[1].

In the original gang scheduling model, processors may be left idle because only one gang is run at a time, even if it is "small". FEITELSON *et al.* [20] propose a hierarchical control of the processors so as to execute several gangs on the same machine. Such an approach can be easily implemented by running one gang scheduler thread for each node of the hierarchy of the machine.

Work-Stealing Scheduling

One of our currently in-progress algorithms is based on work stealing: the hierarchy of bubbles is first settled on the list of processor 0. Then, when `bubble_schedule()` is called on an idle processor, it will use our helper functions to look for work to steal locally (on the runqueue of the other processor of the same chip for instance), then more globally, until finding work to steal. The actual work "steal" is a non-trivial algorithmic problem: only a part of the bubble hierarchy should be pulled toward the idle processor, and the structure of the hierarchy should be taken into account as much as possible, something like a "local draw of the tree of bubbles" along the machine. All attributes and statistics attached to bubbles should be carefully taken into account in heuristics, so as to get a distribution suited to the application. Indeed, according to the application, one should either distribute the memory occupation, or the needed bandwidth ; a compromise has to be found. These are only purely algorithmic issues though: no technical problems remain.

2.4 ForestGomp: an Implementation of GNU OpenMP for Hierarchical Machines

2.4.1 Scheduling OpenMP Applications Featuring Nested, Irregular Parallelism

Achieving the best possible performance when programming OpenMP applications requires developers to expose the parallelism and to explicitly design their code to drive its parallel behavior. Therefore, it is quite common nowadays to define per-thread specific data structures (in order to avoid false-sharing) and use a

[1] over the total $4 \times 100\%$ CPU time, each gang as a whole gets roughly one third ($133\% \simeq 5 \times 26\% \simeq 6 \times 22\% \simeq 7 \times 19\%$)

static, possibly pre-calculated, distribution of the workload to get good data locality [27]. Indeed, this model suits very well regular applications with coarse-grain parallelism.

However, this approach is hardly usable when dealing with irregular applications that rather need a dynamic load balancing mechanism. The use of complex synchronization schemes, or even blocking systems calls, may also be responsible for introducing irregularities regarding the computing load on the available processors. Using OpenMP dynamic scheduling directives can sometimes improve performance. In some cases, however, it may penalize data locality or even introduce false sharing effects, which can severely impact performance on hierarchical architectures.

Another approach is to increase the number of potential parallel tasks using nested parallelism, so that threads can be dynamically (re)allocated according to the workload disparity. The performance of such a dynamic thread management, when supported[2], heavily relies on the underlying runtime implementation, but also on the underlying operating system's scheduler. This explains why OpenMP users have been experiencing poor performance with the nested capabilities of some OpenMP compilers, and have ended up performing explicit thread programming on top of OpenMP [8, 23] or explicitely binding thread groups to processors [48].

Nevertheless, there exists some very good implementations of OpenMP nested parallelism, such as Omni/ST [41] for instance. Such implementations are typically based on a fine-grain thread management system that uses a fixed number of threads to execute an arbitrary number of *filaments*, as done in the Cilk multithreaded system [21]. The performance obtained over symmetrical multiprocessors is often very good, mostly because many tasks can be executed sequentially with almost no overhead when all processors are busy. However, since these systems provide no support for attaching high level information such as memory affinity to the generated tasks, many applications will actually achieve poor performance on hierarchical, NUMA multiprocessors.

One could probably enhance these OpenMP implementations to use affinity information extracted by the compiler so as to better distribute tasks or threads over the underlying processors. However, since only the underlying thread scheduler has complete control over scheduling events such as processor idleness, blocking syscall or even thread preemption, this information could only be used to influence task allocation at the beginning of each parallel section.

We believe that a better solution would be to transmit information extracted by the compiler to the underlying thread scheduler *in a persistent manner*, and that only a tight integration of application-provided meta-data and architecture description can let the underlying scheduler take appropriate decisions during the whole application run time. In other words, one can see this configurable scheduler framework as a domain-specific language enabling scientists to transfer their knowledge to the runtime system [22].

[2]Nested parallelism is currently an optional feature in OpenMP.

To evaluate the potential gain of providing a thread scheduler with persistent information extracted by an OpenMP compiler, we have extended the GNU OpenMP runtime system (i.e. the `libgomp` library) so as to rely on the BubbleSched Framework.

2.4.2 Generating Bubbles Out of OpenMP Parallel Sections

The GNU OpenMP compiler[1], GOMP, is based on an extension of the GCC 4.2 compiler that converts OpenMP pragmas into threading calls. The creation of threads and teams is actually delegated to a shared library, `libgomp`, which contains an abstraction layer to map OpenMP threads onto various thread implementations. This way, any application previously compiled by GOMP may be relinked against an implementation of `libgomp` on another thread type and transparently work the same.

We used this flexible design to develop ForestGOMP, a port of GOMP on top of the MARCEL threading library in which BubbleSched is implemented. To do so, a MARCEL adaptation of `libgomp` threads has been added to the existing abstraction layer. We rely on MARCEL's fully POSIX compatible interface to guarantee that ForestGOMP will behave as well as GOMP on pthreads. Then, it becomes possible to run any existing OpenMP application on top of BubbleSched by simply relinking it.

Once MARCEL threads are created they basically behave by default as native pthreads without any notion of team or memory affinity. BubbleSched hooks have been added in the `libgomp` code to provide information about thread teams by creating bubbles accordingly.

Therefore, when a thread encounters a nested parallel region and becomes the master of a new team, it creates a bubble within its currently holding bubble. Then, it moves itself into this new bubble and creates the team's slave threads inside it. Finally, the master dispatches the workload across the team. Once their work is completed, slave threads die while the master destroys the bubble and returns to its original team. As shown on Figure 2.4, only a few lines of code are needed to associate a nested team hierarchy with a bubble hierarchy.

2.4.3 A Scheduling Strategy Suited to OpenMP Nested Parallelism

The challenge of a scheduler for the nested parallelism of OpenMP resides in how to distribute the threads over the machine. This must be done in a way that favors both a good balancing of the computation and, in the case of multi-core and NUMA machines, a good affinity of threads, for better cache effects and avoiding the remote memory access penalty.

To achieve this, we wrote a **bubble spread** scheduler consisting of a mere recursive function that uses the API described in section 2.3.1 to greedily distribute the hierarchy of bubbles and threads over the hierarchy of runqueues. This function takes in an array of "current entities" and an array of "current runqueues".

```
void gomp_team_start (void (*fn) (void *),
                      void *data, unsigned nthreads,
                      struct gomp_work *work_share) {
  struct gomp_team *team;
  team = new_team (nthreads, work_share);
  ... /* Pack 'fn' and 'data' into the 'start_data'
       * structure */

  if (nthreads > 1 && team->prev_ts.team != NULL) {
    /* nested parallelism, insert a marcel bubble */
    marcel_bubble_t *holder =
      marcel_bubble_holding_task (thr->tid);
    marcel_bubble_init (&team->bubble);
    marcel_bubble_insertbubble (holder, &team->bubble);
    marcel_bubble_inserttask (&team->bubble, thr->tid);
    marcel_attr_setinitbubble (&gomp_thread_attr,
                               &team->bubble);
  }

  for(int i=1; i < nbthreads; i++) {
      pthread_create (NULL, &gomp_thread_attr,
                      gomp_thread_start, start_data);
      ...
  }
}
```

Figure 2.4: One-to-One correspondence between MARCEL's bubble and GOMP's team hierarchies.

It first sorts the list of current entities according to their computation load (either explicitly specified by the programmer, or inferred from the number of threads). It then greedily distributes them onto the current runqueues by keeping assigning the biggest entity to the least loaded runqueue[3], and recurse separately into the sub-runqueues of each current runqueue.

It often happens that an entity is much more loaded than others (because it is a very deep hierarchical bubble for instance). In such a case, a recursive call is made with this bubble "exploded": the bubble is removed from the "current entities" and replaced by its content (bubbles and threads). How big a bubble needs to be to get exploded is a parameter that has to be tuned. This may depend on the application itself, since it permits to choose between respecting affinities (by pulling intact bubbles as low as possible) and balancing the computation load (by exploding bubbles in order to have small entities for better distribution).

This way, affinities between threads are taken into account: since they are by construction in the same bubble hierarchy, the threads of the same external loop

[3]This algorithm comes from the greedy algorithm typically used for resolving the bi-partition problem.

iterations are spread together on the same NUMA node or the same multicore chip for instance, thus reducing the NUMA penalty and enhancing cache effects.

Other repartition algorithms are of course possible, we are currently working on a even more affinity-based algorithm that avoids bubble explosions as much as possible.

2.4.4 A scheduling policy guided by affinity hints

The challenge of a scheduler for the nested parallelism resides in how to distribute the threads over the machine. This must be done in a way that favors both a good balancing of the computation and, in the case of multi-core and NUMA machines, a good affinity of threads, for better cache effects and avoiding the remote memory access penalty.

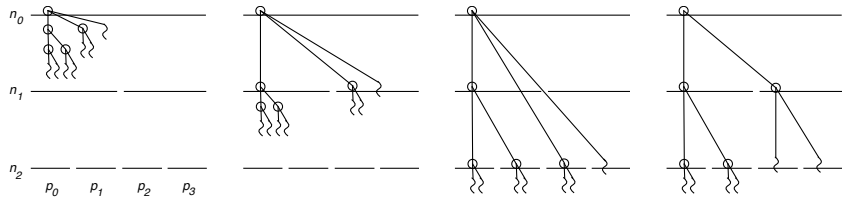
Assumptions

Divide and conquer algorithms generate intensively cooperating groups of threads that run smoother if they are scheduled on the same limited subset of processors. A bad distribution of these collaborating entities results in multiple expensive NUMA accesses over hierarchical architectures, that lowers the general performance of parallel applications. Alternatively, a distribution that considers those affinity relations entails a better use of cache memory, and improves local memory accesses.

The Affinity bubble-scheduler is specifically designed to tackle irregular applications based on a divide and conquer scheme. In this aim, we consider that each bubble contains threads and subbubbles that are heavily related, most of the time through data sharing. We assume that the best thread distribution is obtained by scheduling each entity contained in a bubble on the same processor, sometimes breaking the load balancing scheme, even if a local redistribution is needed once in a while. This scheduler provides two main algorithms, to distribute thread and bubble entities over the different processors initially, and to rebalance work if one of them becomes idle.

Initial Thread Distribution

Entities scheduled in the same bubble should not be torn apart, nevertheless a bubble can be required to extract its contents to increase the number of executable entities in order to occupy every processor of the architecture. This bubble is then said to be exploded. The runqueue level where a bubble explodes during the distribution is crucial to determine whether affinity relations are preserved or not. For instance, if a bubble is exploded on the top level of the topology, its contents can be scheduled on any processor. Therefore the Affinity thread distribution algorithm delays these explosions as much as possible, to maximize locality between the released entities.



(a) Initial state

(b) Greedy distribution after the root bubble explosion

(c) Final distribution

(d) Work rebalancing after processor p3 idleness

Figure 2.5: Threads and bubbles distribution by the *Affinity* scheduler

More precisely, this first scheduling step is based on a mere recursive algorithm to greedily distribute the hierarchy of bubbles and threads over the hierarchy of runqueues. Upon each call, the algorithm counts the entities available to be distributed from the considered runqueues. If there are enough entities to occupy the complete set of processors covered by the runqueues, entities are greedily distributed over the underlying lists. Otherwise, the algorithm analyzes the contents of each available bubble to determine the ones that hold enough threads or subbubbles to occupy a complete subset of processors on their own. If so, bubble explosions are delayed to a further step, thus avoiding early separation of collaborating entities.

Figure 2.5(a) shows the initial state of this recursive algorithm which has been developed to guarantee bubbles and threads distribution from the most general level of the topology, representing the whole computer, to the most specific ones. Figure 2.5(c) shows the resulting distribution, where only the main top-level bubble has exploded. This approach obviously values affinity relations over load balancing, and could not be efficient without a NUMA-aware work stealing algorithm that rearrange the thread distribution when a processor turns to be inactive.

NUMA-aware Work Stealing

The irregular behaviour of some applications prevents estimation of the load of each created thread. This lack of load hints forces the Affinity scheduler to equally consider every entity. As a result, a continuous thread creation and destruction scheme may unbalance the initial thread repartition, and some of the processors may become idle.

The Affinity scheduler implements a dedicated work stealing algorithm to prevent these processors from remaining inactive for too long. This algorithm tracks down lists to steal from, from the most local lists to the most global one if necessary, expanding the search domain as long as no eligible runqueue has been found. Entities are thus stolen as locally as possible. If several entities are usable for work stealing, the Affinity scheduler arbitrarily picks the most loaded one, considering the number of recursively contained threads. If only one bubble is found during the stealing process, its contents are browsed to pick a complete subtree of entities,

as illustrated by figures 2.5(d). When a thread, or a bubble, is finally chosen, the algorithm moves its ancestors to the most internal level of the topology common to the source runqueue and the idle processor, to avoid locking convention issues.

Discussion

A call to Affinity's thread distribution algorithm generally results in assigning a tree of entities to every processor, similar to the Cilk language or OMPi approaches to deal with divide and conquer applications. Statistically, the working load left to an entity located in the upper part of the tree is bigger than the one executed by the leaves-positioned threads. The Affinity scheduler therefore tries to steal from the top of the entity hierarchy, but differs from Cilk implementations by (1) looking for eligible subtrees as close as possible from the idle processor, instead of randomly picking a victim runqueue, and (2) stealing a set of threads that work together rather a lonely thread (like OMPi does). This way of stealing respects the hierarchical nature of both NUMA architectures and the application parallelization scheme.

2.5 Implementation

2.5.1 Implementation over a two level thread library

Our BubbleSched platform is currently implemented as an extension to the MARCEL[4] portable [5] two-level thread library. This library uses operating system functions for detecting the machine architecture, binding one kernel-level thread to each processor and then performs fast user-level context switches between user-level threads. Therefore, assuming that no other application is running, it keeps complete control over thread scheduling on processors in userspace. Operations on bubbles are done at the same level as the application, in user space. "Scheduling daemon threads" are hence just MARCEL threads. So as to avoid the blocking of kernel threads when the application makes blocking system calls, MARCEL uses Scheduler Activations [12] when they are available, or just intercepts blocking calls. Of course, this implementation could also be done in the Linux kernel or other frameworks like SPIN, the ExoKernel, etc. but since our prototype was completely developed in user space, no modification of the underlying operating system is required. This permits an easy deployment on a large variety of test machines without any particular administration privilege.

Our BubbleSched platform also includes a debugger to help understanding the behavior of a scheduler. A lightweight trace of events (thread birth, sleep, wake up, bubble placement) is recorded during the execution of the application [13] and analyzed off-line. This trace can be converted into an animated movie that shows

[4] http://runtime.futurs.inria.fr/marcel/

[5] MARCEL works on a broad range of systems (Linux, AIX, Darwin, Solaris, OSF, etc.) and processors (x86, x86_64, Itanium, PorwerPC, etc.)

	Creation	Execution
MARCEL	0.85	0.60
MARCEL bubbles	0.85	0.70
NPTL	6.4	11.3

Table 2.1: Thread creation time (μs)

	Schedule	Schedule+Switch
MARCEL	0.046	0.270
MARCEL bubbles	0.046	0.310
NPTL	0.33	0.49

Table 2.2: Scheduling cost (μs)

interactively the series of scheduling events and placement decisions that occurred during the execution. Thus, programmers can easily replay the scheduling decisions at will to find out their algorithmic flaws.

2.5.2 Micro-Benchmarks

Thanks to a few microbenchmarks, we could check the overhead of the implementation of bubbles. These test have been conducted on a dual bicore Opteron system running at 1.8 GHz. We have tested the original MARCEL library, MARCEL with bubbles, and as a comparision, the NPTL with the Linux 2.6.22 kernel.

Table 2.1 shows the creation and execution time of threads. We can notice that although bubbles bring a small overhead, creating userlevel threads is still much less costly than creating kernel threads. In the case of bubbles, threads are always automatically inserted to a default bubble, so that the insertion cost is already included here.

Table 2.2 shows the scheduling cost. The Schedule column gives the scheduler time in the case when no actual switch occurs. The Schedule+Switch column gives the scheduler time in the case when a switch occurs. In the case of bubbles, that includes the time to look for threads in the bubble hierarchy. We can notice that that overhead is still low compared to the NPTL case.

Performance of thread migration beetween two cores is a keypoint in the bubble scheduling approach. Table 2.3 shows the cost of such migrations. The Order columns show the time to issue the order itself, and we distinguish the case when the thread is currently active or not. The Latency column shows the time for the migration to actually happen. We can see that performing the order does not cost much, since it is a matter of taking two locks and tinker with lists. The actual migration is performed synchronously, either at next scheduler tick, or (if the scheduler really asks for pay the $\simeq 1\,\mu$s for it) through a preemption Unix signal. In the case of NPTL, the migration order is synchronous (`pthread_setaffinity_np()`), thus the high cost when the thread is already running: the caller has to

	Order (inactive)	Order (active)	Latency
MARCEL	0.047	0.051	2.2
MARCEL bubbles	0.054	0.057	2.5
NPTL	0.8	8	4.8

Table 2.3: Thread migration cost (μs).

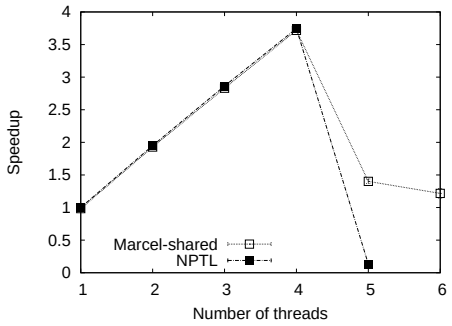


Figure 2.6: Parallel speedup of a single job

wait for it to get preempted.

2.6 Evaluation

2.6.1 LU Factorization

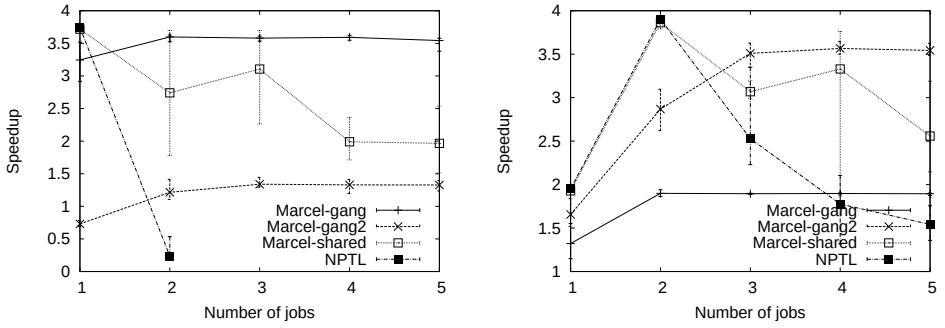
We show the usefulness of a programmable scheduling platform by introducing an example that illustrates well the dilemma between distributing threads over the machine or keeping related threads close together. It was run on a dual-dual-core Opteron system, for which the NUMA factor between the two dual-core nodes is approximately 1.4.

We experimented with SuperLU_MT Version 1.0: this is a thread-parallel solver for large, sparse, non-symmetric systems of linear equations (LU factorization). Figure 2.6 first shows how well SuperLU scales on the target machine. The speedup is quite close to perfect, up to the number of processors [6], but because of cache and synchronization affinities, using a number of threads greater than the number of processors (4) just makes performances become really bad when using generic schedulers like NPTL (LINUX 2.6.17) or our original Marcel library with a mere "shared" policy (single shared runqueue for all processors).

We consider a situation where a multi-scale scientific application needs to perform LU factorizations jobs "on demand". There are many ways to process these jobs, depending on how many threads to run per job and how to schedule them.

[6] On an equivalent 16-way machine, the speedup only gets up to 7.5, however.

Using a mere batch scheduler (running each job 4-way up to completion), or a completely distributed scheduler (running each job 1-way up to completion) may not be wise, in case other parallel parts of the application (running on other machines) need the job result sharply.



(a) 4-way jobs (b) 2-way jobs

Figure 2.7: Parallel speedup of mixed jobs, using generic or gang scheduling policies

Figures 2.7(a) and 2.7(b) show the results obtained using several approaches. On figure 2.7(a), all jobs were run by using four threads, while on figure 2.7(b), all jobs were performed by using two threads. Both figures clearly show that the generic schedulers of NPTL and the original Marcel library get very bad and erratic performances (min-max bars are very tall), because they equally process all threads of all jobs without taking affinities into account. We used our gang scheduler (see section 2.3.2) for running the jobs, with a time slice of 0.2ms (the solving time of one job with only one thread is around 10s) and a single gang scheduler over the whole 4-way machine ("Marcel-gang" curve), but we also have evaluated the execution of two gang schedulers, one on each 2-way NUMA node of the machine ("Marcel-gang2" curve).

The bottom curve of Figure 2.7(b) shows that running only one gang scheduler for 2-way jobs obviously limits the speedup to a value of 2. The bottom curve of Figure 2.7(a) shows that running 4-way jobs on 2-way NUMA nodes also get a limited speedup (because there are more threads than processors). The top curve of Figure 2.7(a) shows that our gang scheduler performs quite well at running jobs on the machine: each job seems to achieve a speedup of approximately 3.5, whatever how many there are. The "Marcel-gang2" curve of Figure 2.7(b) shows that the two gang schedulers achieve this quite well too, provided that there are several jobs, of course. Finally, by carefully comparing these last two curves, one can notice that for this application on this machine, running one gang scheduler for the whole machine is actually a little better than running two separate gang schedulers on each NUMA node. This is probably because we chose, for repartition reasons, to have these two gangs take and put back jobs to a common job pool. Had the application been a little different (less cache- but more memory-bound), we would

have noticed the converse.

It must be noted that this experiment was conducted without modifying the application at all. We just made Marcel build bubbles according to natural thread creation affiliation: since threads that work on the same job are created by the same thread, the resulting bubble hierarchy naturally maps to the job. It was then just a matter of starting gang schedulers with appropriate parameters.

2.6.2 BT-MZ

We validate our approach for OpenMP applications by experimenting with the BT-MZ application. It is one of the 3D Fluid-Dynamics simulation applications of the Multi-Zone OpenMP version of the NAS Parallel Benchmark [14] 3.2. In this version, the mesh is split in the x and y directions into zones. Parallelization is then performed twice: simulation can be performed rather independently on the different zones with periodic face data exchange (coarse grain *outer* parallelization), and simulation itself can be parallelized among the z axis (fine grain *inner* parallelization). As opposed to other Multi-Zone NAS Parallel Benchmarks, the BT-MZ case is interesting because zones have very irregular sizes: the size of the biggest zone can be as big as 25 times the size of the smallest one. In the original SMP source code, outer parallelization is achieved by using Unix processes while the inner parallelization is achieved through an OpenMP static parallel section. Similarly to Ayguade *et al.* [3], we modified this to use two nested OpenMP static parallel sections instead, using $n_o * n_i$ threads.

The target machine holds 8 dual-core AMD Opteron 1.8GHz NUMA chips (hence a total of 16 cores) and 64GB of memory. The measured NUMA factor between chips[7] varies from 1.06 (for neighbor chips) to 1.4 (for most distant chips). We used the class A problem, composed of 16 zones. We tested both the Native POSIX Thread Library of Linux 2.6 (NPTL) and the MARCEL library, before trying the MARCEL library with our *bubble spread* scheduler.

We first tried non-nested approaches by only enabling either outer parallelism or inner parallelism, as shown in Figure 2.8:

Outer parallelism($n_o * 1$): Zones themselves are distributed among the processors. Due to the irregular sizes of zones and the fact that there is only a few of them, the computation is not well balanced, and hence the achieved speedup is limited by the biggest zones.

Inner parallelism($1 * n_i$): Simulation in zones are performed sequentially, but simulations themselves are parallelized among the z axis. The computation balance is excellent, but the nature of the simulation introduces a lot of inter-processor data exchange. Particularly because of the NUMA nature of the machine, the speedup is hence limited to 7.

[7] The NUMA factor is the ratio between remote memory access and local memory access times.

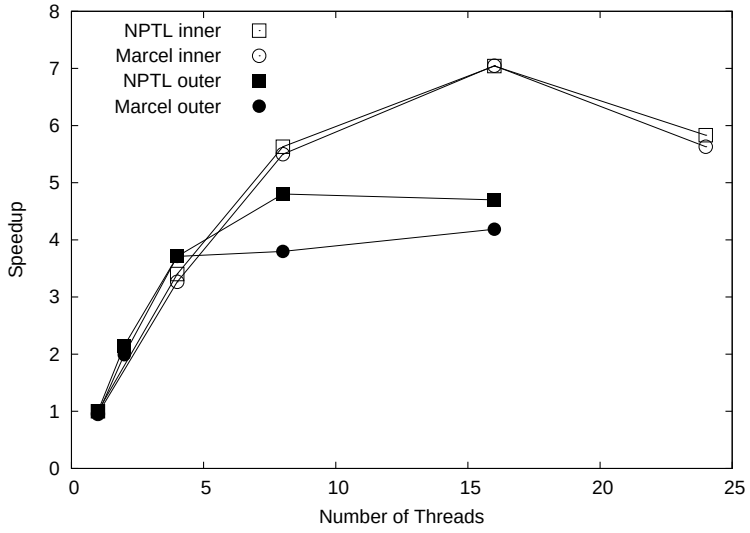


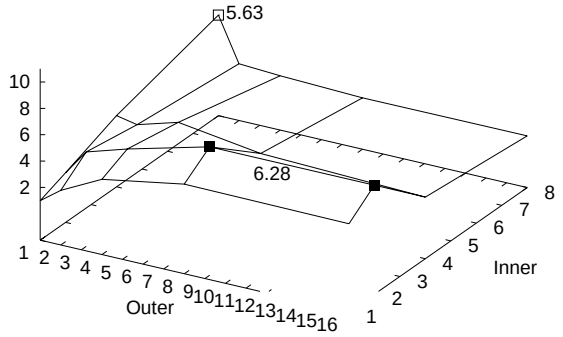
Figure 2.8: Outer parallelism ($n_o * 1$) and inner parallelism ($1 * n_i$).

So as to get the benefits of both approaches (locality and balance), we then tried the nested approach by enabling both parallelisms. As discussed by DURAN *et al.* [16], the achieved speedup depends on the relative number of threads created by the inner and the outer parallelisms, so we tried up to 16 threads for the outer parallelism (i.e. the maximum since there are 16 zones), and up to 8 threads for the inner parallelism. The results are shown on Figure 2.9. The nested speedup achieved by NPTL is very limited (up to 6.28), and is actually worse than what pure inner parallelism can achieve (almost 7, not represented here because the "Inner" axis maximum was truncated to 8 for better readability). MARCEL behaves better (probably because user threads are more lightweight), but it still can not achieve a better speedup than 8.16. This is due to the fact that neither NPTL nor MARCEL takes affinities of threads into account, leading to very frequent remote memory accesses, cache invalidation, etc. We hence used our bubble strategy to distribute the bubble hierarchy corresponding to the nested OpenMP parallelism over the whole machine, and could then achieve better results (up to 10.2 speedup with $16*4$ threads). This improvement is due to the fact that the bubble strategy carefully distribute the computation over the machine (on runqueues) in an affinity-aware way (the bubble hierarchy).

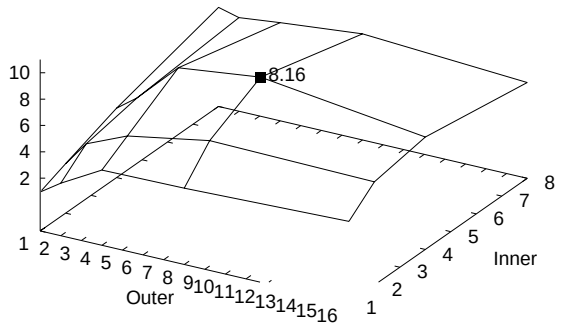
It must be noted that to achieve the latter result, the only addition we had to do to the BT-MZ source code is the following line:

```
call marcel_set_load(int(proc_zone_size(myid+1)))
```

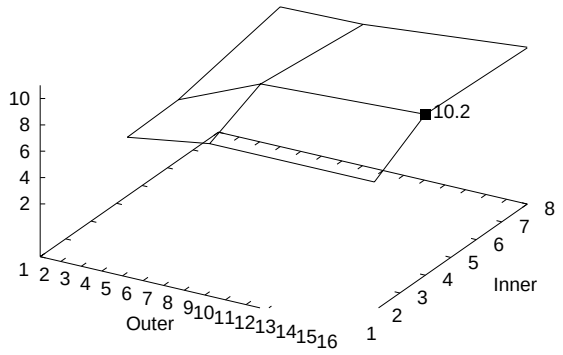
that explicitly tells the bubble spread scheduler the load of each zone, so that they can be properly distributed over the machine. Such a clue (which could even be dynamic) is very precious for permitting the runtime environment to make appropriate decisions, and should probably be added as an extension to the OpenMP



(a) NPTL



(b) Marcel



(c) Bubbles

Figure 2.9: Nested parallelism.

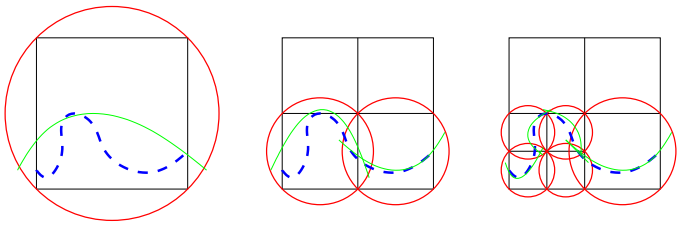


Figure 2.10: Adaptive surface fitting using a recursive subdivision of space which forms a tree hierarchy. Each box is subdivided until the fitted surface is close enough to the points. The resulting surface is a weighted average of each local approximation using partition of unity functions.

standard. Another way to achieve load balancing would be to create more or less threads according to the zone size [3]. This is however a bit more difficult to implement than the mere function call above.

2.6.3 MPU: An higly irregular application

With the Affinity scheduler and several features of MARCEL, it is now possible to parallelize many recursive divide and conquer algorithms, using a naive approach and simple OpenMP constructs, and yet to obtain good speedups. To back our claim, we show that an extremely irregular divide and conquer algorithm, the Multi-Level Partition of Unity algorithm [32], can be parallelized efficiently only by adding a few lines of code to its implementation.

Implicit surface reconstruction application

This surface reconstruction algorithm processes a cloud of points sampling a geometric surface, so as to compute a mathematical representation of this sampled surface. Its main use is related to 3D scanners, that is, devices that are capable of sampling the surface of a physical object by extracting a finite set of 3D points. Reconstructing the whole surface from its samples is required for many applications ranging from rendering to physical simulations.

Thanks to its divide and conquer scheme it is one of the fastest reconstruction algorithms available. Starting from a box containing the whole cloud of points, it tries to fit a simple surface (a quadric) to the points. This surface is implicit, which means that it is defined by a real valued function defined over the entire space and whose value is zero for every point of the implicit surface. If the fitted surface does not approximate the points closely enough, that is, if there are points too far away from the fitted surface, the box is subdivided into 8 subboxes, thus forming an octree. This process is applied recursively to each child box until the error between each approximation and the points of its box is small enough (see Figure 2.10).

This divide and conquer approach is made possible by the use of partition of unity functions. Indeed, using these functions makes it possible to define the global

```
void Node::compute() {
  computeApprox();
  if(_error > _max_error) {
    splitCell();
    for(int i=0; i<8; i++)
      _children[i]->compute();
  }
}
```

Figure 2.11: Sequential MPU code to process a node. An approximation is computed and the node is subdvided if it is not precise enough. This process is then repeated recursively.

```
void Node::compute() {
  computeApprox();
  if(_error > _max_error) {
    splitCell();
#pragma omp parallel for
    for(int i=0; i<8; i++)
      _children[i]->compute();
  }
}
```

Figure 2.12: Parallel MPU code to process a node. A single OpenMP directive has been added to indicate that every node can be processed concurrently.

reconstructed surface as a weighted average of each function defining the local surface approximations. The weight of each local approximation in the weighted average at a given point in space is at its highest at the center of its box, and decreases as the distance to this center increases.

What makes this technique especially attractive is that there is no "stitching" involved between locally computed surfaces. This makes parallelization easier because such a step would require many synchronisations between threads working on neighbour nodes. Therefore this algorithm is well suited to parallelization since every node of the tree can be processed concurrently. The difficulty resides in balancing the work between the processors as the tree is very irregular and there is no simple way to predict where the tree is going to be refined. Ideally the programmer should be able to simply express that the function calls for processing the nodes can be executed concurrently and the runtime would be responsible for balancing this work on the processors.

OpenMP provides constructs that are very well suited to this task, and parallelizing this algorithm using OpenMP is a matter of inserting a few lines of code to indicate that each time a node is subdivided, its 8 children can be processed concurrently (see Fig. 2.11 and 2.12). Running such an application efficiently is challenging for however, because runtime systems need not only to deal with a large number of thread creations/destructions (up to tens of millions for large datasets), but also to schedule them in a way that optimizes memory locality.

Scheduling MPU

We validated our approach by experimenting with the MPU application on a cloud of 437644 points, which leads to the creation of 101185 threads.

The target machine holds 8 bi-core AMD Opteron chips (hence a total of 16 cores) and 64GB of memory. The measured NUMA factor between chips varies from 1.06 (for neighbor chips) to 1.4 (for most distant chips). We tested both

the Native POSIX Thread Library of Linux 2.6 (NPTL) and the MARCEL library, partitioning the set of usable cores in order to execute our tests on respectively 2, 4, 8 and 16 cores. The results can be seen on figure 2.13.

We first tried non-nested approaches to compare the behaviour of these two libraries. Each parallel construct generates a number of threads equal to the number of available cores. The MPU algorithm divides the computed surface in eight different subdomains, every time the refinement primitive is called. Running non-nested tests with a number of threads exceeding 8 is thus not relevant, only the first eight ones will be occupied. The MARCEL thread scheduler operates at user-level, and is less preemptive than the one used by NPTL. MPU thus runs much faster with MARCEL threads.

In the next experiments, we allowed the GOMP compiler to create extra threads when a nested parallel construct is encountered. This approach theoretically suits the MPU application divide and conquer nature. We achieved the best speedups by creating 4 threads at each parallel section. Allowing nested approaches results in creating a great number of threads, and thread creation and management primitives are more expensive in a kernel-level thread library like NPTL. Those used by MARCEL are lighter, which explains why it scales better. On the other hand, neither runtime of those libraries has sufficient information about threads relations to adjust their distribution, so that related threads may be executed by cores located on different NUMA nodes, and the speedup is yet a bit limited. On the contrary, respecting affinity relations by locally scheduling groups of threads results in much better speedups, as can be seen on the *Affinity* curve.

We then evaluated the effectiveness of Affinity's NUMA-aware scheduling algorithm by running two tests. In the first test, the MPU application is *unmodified* but the work stealing algorithm of Affinity is replaced by a *random* work stealing algorithm: the victim is elected from a randomly chosen runqueue. The achieved speedup on 16 cores varies between 8.2 and 11.82.

In the second test, the MPU application is *modified*. A single thread is bound per processor, which is used to schedule tasks in the form of lightweight threads. An idle thread tries to steal tasks from the most local queues when necessary. The structure of the application allows this version to use a lock-free stealing strategy most of the time, in a Cilk-like manner. The result is that the Cilk-like version obtains the best speed-up, 15.05 on 16 cores, *at the cost of portability*, since the MPU application was modified to integrate this algorithm. With a speedup of 14.04 out of 16 cores, the ForestGOMP results almost reach the *cilk-like* results, *without sacrificing* either portability or generality in application-specific optimizations.

2.7 Related Work

Several operating systems and projects propose tools similar to our BubbleSched platform.

The Intel Many-Core RunTime [37] environment relies on user-level thread-

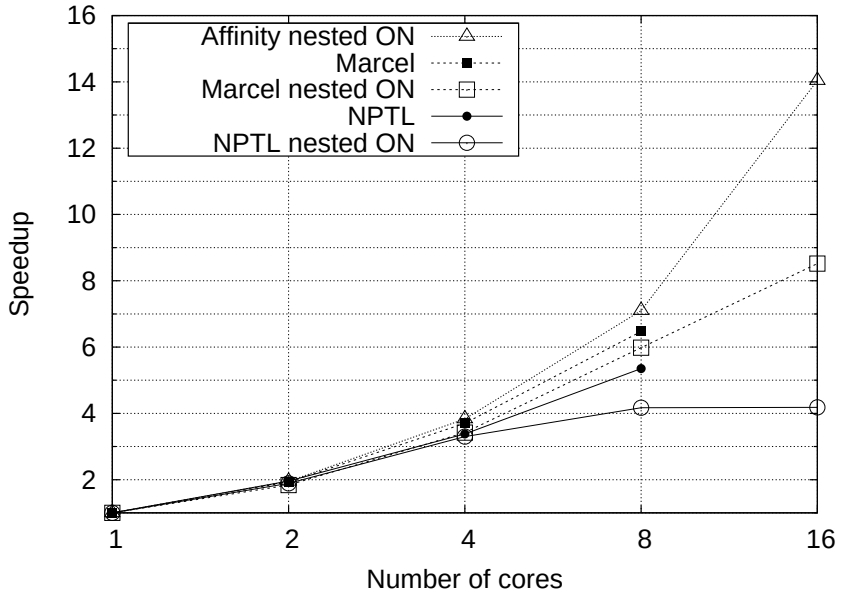


Figure 2.13: Speedup of various MPU implementations.

ing primitives including a scheduler, a transactionnal memory manager, synchronization primitives and provides two user-level threading abstractions: threads and futures (which are more lightweight). From the scheduling point of view, McRT extends the task queue mechanism to support scheduling domains that allows an application to select different hardware units where its different parts to be scheduled on. The application can also specify the scheduling policy either work-sharing or work-stealing policy. It however doesn't provide any framework for developing dedicated schedulers, and thus can only provide a limited range of scheduling possibilities.

Bossa [6] provides scheduling abstractions and a language for developing schedulers. However, its goal is to *prove* the correctness of the scheduler, and as a such the proposed language, though powerful enough to implement the LINUX 2.2 mono-processor scheduler, is quite restrictive and limits programmers a lot. Implementing the Bossa environment in our BubbleSched platform could actually be possible.

The ELITE project [40] provides a very good implementation of a user-scheduling platform that takes into account affinities between threads, cache and data. Mathematical models are even used to calculate probabilities of cache misses. However, interaction with the application is very poor: affinities are detected rather than pro-

vided by programmers.

Fedorova [19] worked on cache-aware schedulers for Operating Systems, and more generally Operating Systems have fairly good schedulers. However, these are targetted towards "blind" multi-application situations and hence can not benefit from knowledge provided by the programmer of a scientific application.

SGI's Process Aggregates [38] are containers for Unix processes. Process sessions and groups can be implemented as process aggregates for instance. However, their current uses are for security and accounting purposes only.

Several operating systems provide facilities for distributing kernel threads along the machine by grouping them into sets: liblgroup on SolarisNSG on Tru64and libnuma on Linux.These look very much like single level bubbles, but no possibility of nested sets is provided, which limits the affinity expressivity. Moreover, none of them provides the degree of control that we provide: with *BubbleSched*, the application has hooks at the very heart of the scheduler to react to events like *thread wake up* or *processor idleness*.

2.8 Conclusion

In this chapter, we present the BubbleSched platform, a tool for designing and prototyping specialized schedulers for specific application domains and libraries (e.g. adaptive mesh refinement algorithms, SPMD codes) for which threads behavior and memory affinities can be predicted to some extent. It provides programmers both with a way to express the application structure, and several high-level scheduling distribution primitives that let scheduling experts write *bubble algorithms* that tightly "drive" the thread scheduler by implementing some hooks. Actually, part of this work was done in collaboration with researchers at the CEA (french Atomic Energy Commission) who have been developing huge HPC applications for a few decades and who are looking for a tool allowing them to transfer their expertise to the underlying runtime system.

We have also discussed the importance of establishing a persistent cooperation between an OpenMP compiler and the underlying runtime system for achieving high performance on nowadays multi-core NUMA machines. We showed how we extended the GNU OpenMP implementation, GOMP, for making use of the flexible MARCEL thread library and its high-level *bubble* abstraction. This permitted us to implement a scheduling strategy that is suited to OpenMP nested parallelism. The results show that it improves the achieved speedup a lot.

This work opens numerous future prospects. We plan to extract the properties of memory affinity at the compiler level, and express them by injecting gathered information into more accurate attributes within the bubble abstraction. These properties may be obtained either thanks to new directives *à la* UPC[8] [10] or be

[8]The UPC `forall` statement adds to the traditional `for` statement a fourth field that describes the affinity under which to execute the loop

computed automatically via static analysis [39]. For instance, this kind of information is helpful for a bubble-spreading scheduler, as we want to determine which bubbles to explode or to decide whether or not it is interesting to apply a *migrate-on-next-touch* mecanism [26] upon a scheduler decision. All these extensions will rely on a memory management library that attaches information to bubbles according to memory affinity, so that, when migrating bubbles, the runtime system can migrate not only threads but also the corresponding data.

On a longer run, a generic tunable scheduler could be developed; it would take into account as much information as possible from the hardware, the compiler and programmers. An integration to the LINUX kernel could even be considered, since a natural bubble hierarchy is already expressed through the notions of threads, processes, sessions and users.

Chapter 3

StarPU : a unified platform for task scheduling on heterogeneous multicore architectures

3.1 Introduction

Multicore processors are now mainstream. To face the ever-increasing demand for more computational power, HPC architectures are not only going to be *massively* multicore, they are going to feature *heterogeneous* technologies such as specialized coprocessors (e.g. Cell/BE SPUs) or data-parallel accelerators (e.g. GPGPUs). As illustrated by the currently TOP500-leading IBM RoadRunner machine, which is composed of a mix of CELLs and OPTERONs, accelerators have indeed gained a significant audience. In fact, heterogeneity not only affects the design of computing units itself (CPU *vs* CELL's SPU), but also memory design (cache and memory banks hierarchy) and even programming paradigms (MIMD *vs* SIMD).

But is HPC, and its usual momentum, ready to face such a revolution ? At the moment, this is clearly not the case, and progress will only come from our ability to harness such architectures while requiring minimal changes to programmers' habits. As none of the main programming standards (*i.e.,* MPI and OPENMP) currently address all the requirements of heterogeneous machines, important standardization efforts are needed. Unless such efforts are made, accelerators *will* remain a niche.

Instead, enhanced versions of computation kernels (BLAS routines, FFT, and other numerical libraries) can be provided to be used as black boxes. Since no performance model that would allow to use a static hardware resource assignment is likely to emerge in the near future, these kernels must be designed to dynamically adapt themselves to the available resources and the application load. Finally, we must find a way to design *scalable* kernels, able to use many cores and multiple GPGPUs at the same time.

As an attempt to provide a runtime system allowing the implementation of

such numerical kernels, we have recently developed a data-management library that seamlessly enforces a coherent view of all hardware memory banks (*e.g.,* main memory, SPU local store, GPU on-board memory, etc.) [2]. This library was successfully used to implement some simple numerical kernels quite easily (using a straightforward scheduling scheme), the next step is now to abstract the concept of task on heterogeneous hardware and to provide expert programmers with scheduling facilities. Integrated within a high-level programming environment such as OPENMP, this would indeed help application developers to concentrate on high-level algorithmic issues, regardless of the underlying scheduling issues.

Here, we propose StarPU, a simple tasking API that provides numerical kernel designers with a convenient way to generate parallel tasks over heterogeneous hardware on the one hand, and easily develop and tune powerful scheduling algorithms on the other hand. StarPU is based on the integration of the data-management facility with a task execution engine. We demonstrate the relevance of our approach by showing how heterogeneous parallel versions of some numerical kernels were developed together with advanced scheduling policies.

The remaining of this chapter is organized as follows. Section 3.2.2 presents the unified model we propose to design in StarPU. In section 3.3, we enrich our model with support for scheduling policies. We study how scheduling improves the performance of our model in section 3.4. We compare our results with similar works in section 3.5, and section 3.6 draws a conclusion and plans for future works.

3.2 A unified runtime system

Each accelerator technology usually has its specific execution model (*e.g.,* CUDA for NVIDIA GPUs), and its proper interface to manipulate data (*e.g.,* DMA on the CELL). Porting an application to a new platform therefore often boils down to rewriting the entire application, which severely impairs productivity. Writing portable code that runs on multiple targets is currently a major issue, especially if the application needs to exploit multiple accelerator technologies, possibly at the same time.

As a result, we design a runtime system with an interface unifying accelerator technologies as well as multicore programming. It allows programmers to develop applications with limited efforts to efficiently exploit different accelerators. Upper layers tools (such as compiler environments and HPC libraries) can rely on StarPU to keep focused on their specific role instead of handling low level technical issues. We now present the main components of StarPU : a high level library that takes care of transparently performing data movements, and a unified execution model.

3.2.1 A library unifying data manipulation

As accelerators and processors usually cannot transparently access the memory of each other, performing computations on such architectures implies explicitly

moving data between the various computational units. Considering that multiple technologies may interact, and that specific knowledge is required to handle the variety of low-level techniques, in previous work [2] we designed a high level library that efficiently automates data transfers throughout heterogeneous machines.

This library has multiple roles in StarPU : on the one hand, it takes care of enforcing data coherency at the software level when there is no such mechanism in the hardware; on the other hand, it implements mechanisms that would hardly be doable by hand (*e.g.,* reordering or inter-accelerators transfers) in order to optimize data transfers which are particularly critical.

Keeping data coherent Given the increase of the number of computational units, reducing the pressure on the memory subsystem becomes crucial. Caching reduces the demand on the memory bus by avoiding numerous data transfers. Data replicates must however be kept consistent with respect to a memory model. Our library implements an MSI protocol that keeps track of the state of each data on every memory nodes at all times. When a coherency miss occurs (if data is not available or the local copy is not up-to-date), it is transparently transferred to the requesting computing resource.

Manipulating hierarchical data While memory reclaiming makes sure that unused cached data will not prevent a task from execution, there is still no guarantee that all the memory requirements of a given task may ever be fulfilled. It is indeed impossible to directly execute a task that computes the sum of two matrices of 1 GB each on a CELL coprocessor, since the latter only has 256 KB of directly addressable memory. It is however possible to use decomposition filters to partition data into smaller pieces. Those can be described in a hierarchical way by composing multiple filters. Contrary to HPF, our filters are not limited to regular patterns for dense matrices, and it is possible to define specifically tailored filters to fit the needs of any application. In addition to a high-level interface that matches the way data are manipulated at the algorithmic level, such a partitioning improves data locality.

Overcoming memory limitations Memory is a scarce resource that must be handled carefully. Since CELL coprocessors only embed 256 KB of local memory, and that GPUs still have relatively limited memory, it is important to make sure our caching technique does not prevent other tasks from getting the amount of memory they need. We ensure that the memory of unused cached data can be reclaimed by possibly discarding data into main memory. As a result, our approach makes it possible to operate on arbitrarily large data-sets instead of being limited to the size of the accelerators' memory.

3.2.2 A unified execution model for heterogeneous multicore architectures

The variety of technologies makes accelerator programming highly dependant on the underlying architecture, so we propose a uniform approach for task and data parallelisms on heterogeneous platforms. We use the *codelet* structure which abstracts a task that can be executed on a core or offloaded onto an accelerator using an asynchronous continuation passing paradigm. After describing the application as a set of interdependent *codelets*, programmers supply an implementation of those *codelets* for each of the architectures that can execute them, using their respective usual programming language. For a given architecture, a *codelet* boils down to a function that complies with a simple interface.

Declaring tasks and data dependencies A *codelet* includes a high level description of the data and the type of access (*i.e.,* read, write or both) that is needed. Since *codelets* are launched asynchronously, this allows StarPU to reorder tasks in case this improves performance. Programmers may then declare dependencies between tasks and let StarPU automatically enforce the actual dependencies between *codelets*.

Designing *codelet* drivers As the aim of *codelets* is to offer a uniform execution model, it is important that it be simple enough to fit to the various architectures that might be targeted. In essence, adding the support of *codelets* for a new architecture means writing a driver that continuously does the following: request a *codelet* from StarPU, fetch its data, execute the appropriate implementation of the *codelet*, perform its callback function and tell StarPU to unlock tasks waiting on this *codelet*. For now, that model has been successfully implemented on top of CUDA, on multicore multiprocessors, and we are porting it on CELL's coprocessors as we have already successfully used a similar approach in previous work in the GORDON runtime system [29].

3.3 A uniform queue-based interface to design scheduling policies

Previous section has shown how tasks can be executed on the various processing units of a heterogeneous machine. However, we did not specify how they should be distributed efficiently, especially with regards to load balancing. It should be noted that nowadays architectures have gone so complex that it is very unlikely that writing portable code which efficiently maps tasks statically is either possible or even productive.

3.3.1 Integrating scheduling into our uniform execution model

As emphasized in section 3.2.1, data transfers have an important impact on performance so that a scheduler favouring locality may increase performance by improving data reuse and thus caching techniques. Also, considering that multiple problems may be solved concurrently, and that machines may not be totally dedicated (*e.g.,* when coupling codes), scheduling tasks dynamically has become a necessity. In the context of heterogeneous platforms, performance vary a lot between architectures (*i.e.,* in terms of raw performance) and depending on the workload (*e.g.,* SIMD code *vs.* irregular memory access). It is therefore crucial to take the specificities of each computing unit into account when assigning work.

Similarly to the problems of data transfers or task offloading, heterogeneity makes the design and the implementation of portable scheduling policies rather challenging to implement. So we propose to extend our uniform execution model with a uniform interface to design *codelet* schedulers. StarPU offers low level scheduling mechanisms (*e.g.,* work stealing) so that scheduler programmers can use them in a high level fashion, regardless of the underlying (possibly heterogeneous) target architecture. Since all scheduling strategies have to implement the same interface, they can be programmed independently from applications, and the user may select the most appropriate strategy at runtime.

In our model, each worker is associated with an *abstract* queue of *codelets*. There are only two operations that can be performed on that queue: task submission (`push`), and request for a task to execute (`pop`). The actual queue may be shared by several workers provided its implementation takes care of protecting it from concurrent accesses, thus making it totally transparent for the *codelet* drivers. All scheduling decisions are typically made within the context of calls to those functions, but there is nothing that prevents a strategy from being called in other circumstances or even periodically.

In essence, defining a scheduling policy consists in creating a set of queues and associating them with the different workers. Various designs may be used to implement the queues (*e.g.,* FIFOs or stacks), and queues may be organized according to different topologies (*e.g.,* a central queue, or per-work queues). Differences between strategies typically result from the way one of the queue is chosen when assigning a new *codelet* after its submission by the means of a `push` operation.

Since they naturally fit our queue-based design, all the strategies that have been written with our interface yet (see Table 3.1) implement a greedy *list scheduling* paradigm: when a ready task (*i.e.,* all its dependencies are fulfilled) is submitted, it is directly inserted in one of the queues, and former scheduling decisions are not reconsidered. Contrary to DAG scheduling policies, we do not schedule tasks that are not yet ready: when the last dependency of a task is executed, StarPU schedules it by the means of a call to the usual `push` function.

3.3.2 Enriching our uniform execution model with optional scheduling hints

To investigate the possible scope of performance improvements thanks to a better scheduling, we let the programmer add some extra optional scheduling hints within the *codelet* structure. One of our objective is to fill the gap between the tremendous amount of work that has been done in the field of scheduling theory and the need to benefit from those theoretical approaches on actual machines.

Declaring priority tasks The first addition is to let the programmer specify the level of priority of a task. Such priorities typically prevent crucial tasks from having their execution delayed too much. While determining which are the priority tasks usually makes sense from an algorithmic point of view, it should also be possible to infer it provided an analysis of the task DAG.

Guiding scheduling policies with performance models Many theoretical studies of scheduling problems often assume to have a *weighted* DAG of the tasks [4]. Whenever it is possible, we thus propose to let the programmer specify a performance model to extend the dependency graph with weights. Scheduling policies may subsequently eliminate the source of load imbalance by distributing work with respect to the amount of computation that has already been attributed to the various processing units.

The use of performance models is actually fairly common in high performance libraries. Various techniques are thus available to allow the programmer to make performance predictions. Some libraries exploit the performance models of computation kernels that have been studied extensively (*e.g.,* BLAS). This for instance makes it possible to select an appropriate granularity [47] or even a better (static) scheduling [36]. It is also possible to use sampling techniques to automatically determine such model costs, provided actual measurements. That may be done either by the means of a pre-calibration run, using the results of previous executions, or even by dynamically adapting the model with respect to the running execution.

Predicting performance may also be made more complex by the heterogeneous nature of the different workers. Scheduling theory literature often assumes that there is a mathematical model for the amount of computation (*i.e.,* in FLOP), and that execution time may be deduced given the relative speed of each processor (*i.e.,* in FLOP/S) [4]. It is however possible that a *codelet* is implemented using different algorithms (with different algorithmic complexities) on the various architectures. As an algorithm may also behave differently depending of the underlying architecture, another solution is to create performance models for each architecture. In the following section, we compare both approaches and analyze the impact of model accuracy on performances.

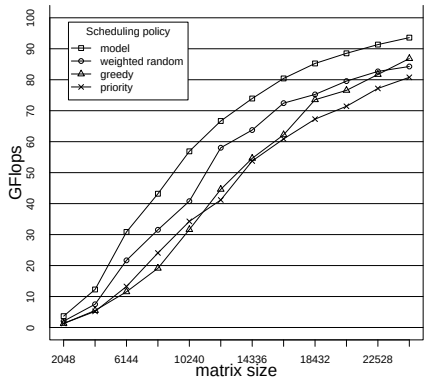


Figure 3.1: LU decomposition

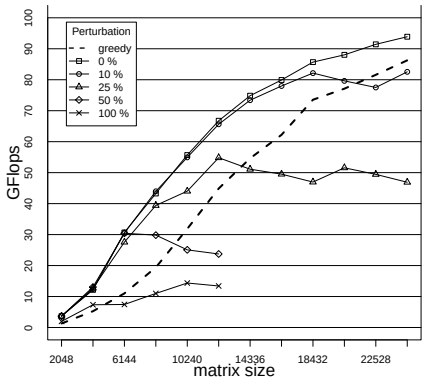


Figure 3.2: Impact of the model accuracy

3.4 Experimental validation

To validate our approach, we present several scheduling policies and experiment them in StarPU on a few applications. To investigate the scope of improvements that scheduling may offer, we gradually increase the quality of the hints given to StarPU by the programmer. We then analyze in more details how StarPU takes advantages of a proper scheduling to exploit heterogeneous machines efficiently.

3.4.1 Experimental environment

Our experiments were performed on an E5410 XEON quad-core running at 2.33 GHz with 4 GB of memory and an NVIDIA QUADRO FX4600 graphic card with 768 MB of embedded memory. This machine runs LINUX 2.6 and CUDA 2.0. We used the ATLAS 3.6 and the CUBLAS 2.0 implementations of the BLAS kernels. All measurements were performed a significant number of times and unless specified otherwise, the standard deviation is never above 1 % of the average value which we show. Given CUDA requirements, one core is dedicated to controlling the GPU efficiently [2] so that we compute on three cores and a GPU at the same time.

3.4.2 Benchmarks

In order to analyze the behaviour of StarPU and the impact that scheduling may have on their performance, we implemented several (single precision) numerical algorithms that use *codelets*. All these algorithms are compute-bound as they mostly involve BLAS3 kernels ($\mathcal{O}(n^3)$ operations against $\mathcal{O}(n^2)$ memory accesses) which are especially adapted to cache sensitive architectures such as multicore and GPGPUs.

Table 3.1: Scheduling policies implemented using our interface

Policy	Category	Queue design	Load balancing
default	greedy	central FIFO	n/a
priority	greedy	central deque or priority FIFO	n/a
work-stealing	greedy	per-worker deque	steal
weighted random	directed	per-worker FIFO	model
cost model	directed	per-worker FIFO	model

- **A blocked matrix multiplication** which will help us demonstrate that greedy policies are not always effective, even on such simple algorithms.
- **A blocked Cholesky decomposition** (without pivoting) which emphasizes the need for priority-based scheduling.
- **A blocked LU decomposition** (without pivoting) which is similar to Cholesky but performs twice as much computation, and thus parallelism. This demonstrates how our system tackles load balancing issues while actually taking advantage of a heterogeneous platform.

Performance figures are shown in synthetic GFLOP/S as this gives an evaluation of the efficiency of the computation since speedups would not be relevant for such heterogeneous platforms.

3.4.3 Scheduling strategies implemented using our interface

We currently implemented a set of common queue designs (stack, FIFO, priority FIFO, deques) that can be directly manipulated within the different methods in a high level fashion. The policy may also decide to create different queue topologies, for instance a central queue or per-worker queues. The `push` and the `pop` methods are then responsible for implementing the load balancing strategy.

Defining a policy with our model only consists in defining very few methods:

- A method that is called during the initialization of StarPU.
- An initialization method that is called by all the workers that will execute *codelets*.
- The `push` and the `pop` methods that implement the interaction with the abstract queue.

Table 3.1 shows a list of scheduling policies that were designed usually in less than 100 lines of C code which shows the conciseness of our approach.

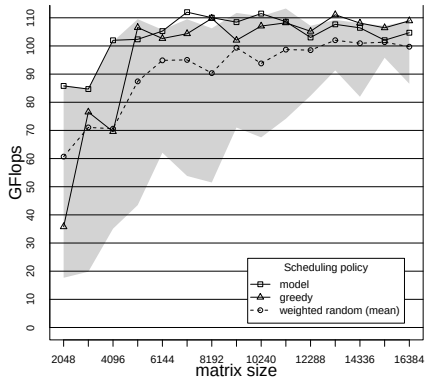


Figure 3.3: Blocked Matrix Multiplication

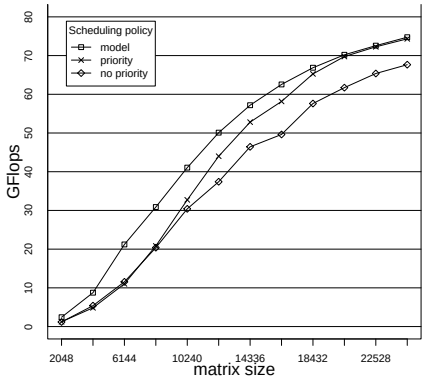


Figure 3.4: Cholesky decomposition

3.4.4 Impact of the design of the queues

The choice of the design and the organization of the queues that compose a strategy is important when writing a scheduling strategy. The choice between a FIFO and a stack may also be important: a stack may for instance help to improve locality and thus data reuse, especially with divide-and-conquer algorithms, but implementing priority is easier with a FIFO. As all our benchmarks naturally tend to have a FIFO task ordering, measurements are not performed on stack-based strategies since that is irrelevant.

Even if it may require some load balancing mechanisms, decentralising queues helps to reduce contention and makes it possible to handle each worker specifically. This is also interesting when accessing a global shared queue is expensive (*e.g.,* on the CELL which needs expensive DMA transfers).

Scheduling policies that include support for priority tasks mostly address algorithms with dependencies that suffer insufficient parallelism, thus affecting load balance. Figure 3.4 shows that the Cholesky algorithm benefits from priorities by up to $10\,GFlops$ on large problems. However, it does not affect LU decomposition a lot on Figure 3.1 as the FIFO ordering naturally fits the natural priorities of the algorithm. On Figure 3.1, priority tasks are either appended at the end of a global FIFO (*greedy* policy) or put into dedicated ones (*priority* FIFO). Small problems benefit from a strict FIFO ordering, but large ones perform better with the greedy algorithm. This confirms that choosing the perfect scheduling policy is not obvious, and that having different ones allows to choose the one that performs best.

3.4.5 Policies based on performance models

Figure 3.3 demonstrates that the *greedy* policy delivers around 105 GFLOPS on medium-sized problems, which is about 90 % of the sum of the results achieved on a single GPU (91.3 GFLOPS) and 3 cores (25.5 GFLOPS). That relatively low efficiency is explained by the important load imbalance. Intuitively, that issue can be solved if a GPU that is n times faster than a core is given n times more tasks: next section shows how that can be achieved using performance models.

Model the capabilities of the workers

In the *weighted random* strategy, each worker is associated with a ratio that can be interpreted as an *acceleration* factor. Each time a task is submitted, a random number is generated to select one of the workers with a probability proportional to its ratio. That ratio may for instance be set by the programmer once for all on the machine, or be measured thanks to reference benchmarks (*e.g.,* BLAS kernels). The *weighted random* strategy should typically be suited to independent tasks of equal size.

Still, the shaded area on Figure 3.3 shows that this policy produces extremely variable schedules for which the lack of load balancing mechanism explains why the average value is worse than the *greedy* policy. Unexpectedly, this strategy also gives really interesting improvement of an order of 10 GFLOPS on LU and Cholesky decompositions even though the latter is not shown on Figure 3.4 for readability reasons. It is interesting to note that this optimization is effective on the entire spectrum of sizes, especially on medium ones for which it is especially interesting to obtain performance improvements without any effort from programmers.

Model tasks as well

Evenly distributing tasks over workers with regard to their respective speed does not necessarily make sense for tasks that are not equally expensive, or if there are dependencies between them. As mentioned in section 3.3.2, we therefore let programmers specify a *cost model* for each task. This model can either represent the amount of work and be used in conjunction with the relative speed of each workers as in previous section, or it may directly model the execution time for each of the different architectures. We implemented the *Earliest Task First* strategy using those models: each worker is assigned a queue. Given the expected duration of the work in that queue, a new task is assigned to the queue that minimizes its termination.

By severely reducing load balancing issues, we obtained substantial improvements over all our previous strategies on our three benchmarks. We always almost obtain the best execution for matrix multiplication in spite of the limited amount of work and the limited scope for performance improvements since tasks are independent. On the Cholesky decomposition, the performance model outperforms the other optimization on all sizes. This strategy improves the performance of medium

Table 3.2: Superlinear acceleration on LU decomposition (30720×30720)

	Simple performance model			Per-architecture performance model		
Measured speed (GFlops)	3 CPUs + 1 GPU 95.41	3 CPUs 21.24	1 GPU 75.04	3 CPUs + 1 GPU 98.21	3 CPUs 21.68	1 GPU 75.07
Efficiency	$95.41 =$ **99.1 %**	$(21.24 + 75.04)$		$98.21 =$ **101.5 %**	$(21.68 + 75.07)$	

and small problems up to twofold, but we observe equivalent results for large ones because the *priority* strategy does not suffer too much load imbalance on large inputs. The *performance modeling* strategy is also handicapped by a limited support for priority tasks, which is especially useful on the Cholesky benchmark. Likewise, the LU decomposition obtains even more important improvements as we achieve up to 25 GFLOPS improvement, thus reducing execution time by a factor of 2 on medium size problems. It also increased asymptotic speed from 80 GFLOPS to more than 95 GFLOPS.

On Figure 3.2, we applied a random variation on the performance prediction to study how the model accuracy affects the scheduling. Even with large miss-predictions (*e.g.,* $25\,\%$), the *performance modeling* strategy still outperforms other ones for medium size problems. The importance of accuracy seems to depend on the size of the problem that needs to be solved. But since we observed that the execution time is usually within less than $1\,\%$ of the prediction, we see that it may be worth paying a little attention to an accurate modeling, but it needs not give exact prediction to be useful.

3.4.6 Taking advantage of heterogeneity

Table 3.2 shows the efficiency of our parallel code on an heterogeneous machine. While such heterogeneity could put a serious limitation in terms of programmability, our implementation of the LU decomposition is not affected by the possible use of various architectures at the same time: the performance measured with three CPUs and a GPU is equivalent to $99.1\,\%$ of the sum of the performance measured with three cores on the one hand, and with one GPU on the other hand. This result is contrasted by the need to dedicate one core to the accelerator, but it demonstrates that the overhead is rather low, mostly caused by parallelization rather than heterogeneity itself.

Table 3.2 also shows an interesting *superlinear* efficiency of $101.5\,\%$ when using a per-architecture performance model instead of a mere *accelerating factor* with a single performance model common to all architectures. This illustrates the impossibility to model the actual capabilities of the various compute units by the means of a mere *ratio*, even though this model is fairly common in theoretical scheduling literature [4]. Some tasks indeed suit GPUs while others are relatively more efficient on CPUs: matrix multiplication may be ten times faster on a GPU than on a core while a CPU may be only five times slower on matrix additions. The rationale behind this *superlinear* efficiency is that it is better to execute the tasks

you are good at, and to let others perform those for which you are not the best.

3.5 Related works

If accelerators have received a lot of attention in the last years, most people are programming them directly on top of constructors' API at a low level, with little attention for code portability. While GPGPUs were historically programmed using standard graphic APIs [34], AMD's FIRESTREAM SDKand especially NVIDIA's CUDA are by far the most common way to program GPUs nowadays. Likewise, CELL is still usually programmed directly on top of the low level LIBSPE interface even if the ALF attempts to target both CELL and multicore processors. FPGA, CLEARSPEED and all other accelerating boards still need specific vendor interfaces. Most efforts however tend to be around writing fast computation kernels rather than designing a generic programming model.

In contrast, multicore (and SMP) programming is getting more mature and *standards* such as OPENMP, which has gained substantial audience in spite of MPI which still remains the most commonly used standard in HPC. Consequently, a lot of projects try to implement the MPI standard on the CELL [31, 35] while it does not seem to be adapted for GPUs even if it becomes common to use hybrid models (*e.g.,* CUDA with MPI processes). DURAN *et al.* propose to enrich OPENMP with directives to declare data dependencies [17], which is particularly useful for all accelerator technologies. OPENMP therefore seems to be a promising programming interface for both CELL [7] and GPUs provided compiling environments are offered sufficient support. StarPU could thus be used as a back-end for CELLSS or for the HMPP [15] which resp. generate *codelets* for the CELL and for GPUs.

A lot of efforts have also been devoted to design or to extend languages with a proper support for data and task parallelism, but most of them actually re-implement a streaming paradigm [28] which does not necessarily capture all applications that may exploit accelerators. Various projects intend to implement libraries with computation kernels that are actually offloaded [9], but StarPU avoids for instance the limitation of the size of problems (solved by BARRACHINA *et al.* [5]) while preserving the benefits of their work at the algorithmic level.

Some runtime systems were designed to address multicore and accelerators architectures [11, 29, 46]. Most approaches adopt an interface similar to CHARM++'s asynchronous OFFLOAD API. The well established CHARM++ runtime system actually offers support for both CELL [25] and GPUs [46] (even though there are no performance evaluation available yet for GPUs to the best of our knowledge). But its rather low level interface only has limited support for data management: offloaded tasks only access blocks of data instead of our high level arbitrary data structures, and they do not benefit from our caching techniques.

3.6 Conclusion and future works

We presented StarPU, a new runtime system that efficiently exploits heterogeneous multicore architectures. It provides a uniform execution model, a high-level framework to design scheduling policies and a library that automates data transfers. We have written several scheduling strategies and observed how they perform on some classical numerical algebra problems.

In addition to improving programmability by the means of a high level uniform approach, we have shown that applying simple scheduling strategies can significantly reduce load balancing issues and improve data locality. Since there exists no ultimate scheduling strategy that addresses all algorithms, programmers who need to hard-code task scheduling within their hand-tuned code may experiment important difficulties to select the most appropriate strategy. Many parameters may indeed influence which policy is best suited for a given input. Empirically selecting the most efficient one, at runtime, makes it possible to benefit from scheduling without putting restrictions or making excessive assumptions. We also demonstrated that given a proper scheduling, it is possible to exploit the specificities of the various computation units of a heterogeneous platform and to obtain a consistent superlinear efficiency.

It is crucial to offer a uniform abstraction of the numerous programming interfaces that result from the advent of accelerator technologies. Unless such a common approach is adopted, it is very unlikely that accelerators will evolve from a *niche* with dispersed efforts to an actual mainstream technique. While the OPENCL standard also does provide task queues and an API to offload tasks, our work shows that such a programming model needs to offer an interface that is simple but also expressive.

We plan to implement our model on additional accelerator architectures, such as the CELL by the means of a driver for the GORDON runtime system [29], or even on top of generic accelerators with an OPENCL driver. In the future, we expect StarPU to offer support for the HMPP compiling environment [15] which could generate our *codelets*. We are also porting real applications such as the PASTIX [36] or the MUMPS solvers. Our interface could offer a high-level platform to implement some of the numerous policies that exist in the scheduling literature. In addition to enriching the set of policies available for programmers, this would reduce the gap between HPC applications and theoretical works in the field of scheduling.

Chapter 4

Software Availability

Marcel and BubbleSched are available for download within the PM2 distribution at `http://runtime.bordeaux.inria.fr/marcel` and StarPU is available for download at `http://runtime.bordeaux.inria.fr/StarPU/` under the GPL license.

Bibliography

[1] GOMP – An OpenMP implementation for GCC. `http://gcc.gnu.org/projects/gomp/`.

[2] C. Augonnet and R. Namyst. A unified runtime system for heterogeneous multicore architectures. Las Palmas de Gran Canaria, Spain, 2008.

[3] E. Ayguade, M. Gonzalez, X. Martorell, and G. Jost. Employing Nested OpenMP for the Parallelization of Multi-Zone Computational Fluid Dynamics Applications. In *18th International Parallel and Distributed Processing Symposium (IPDPS)*, 2004.

[4] C. Banino, O. Beaumont, L. Carter, J. Ferrante, A. Legrand, and Y. Robert. Scheduling strategies for master-slave tasking on heterogeneous processor platforms. *IEEE Trans. Parallel Distrib. Syst.*, 15(4):319–330, 2004.

[5] S. Barrachina, M. Castillo, F. D. Igual, R. Mayo, and E. S. Quintana-Ort. Solving Dense Linear Systems on Graphics Processors. Technical report, Universidad Jaime I, Spain, Feb. 2008.

[6] L. P. Barreto and G. Muller. Bossa: une approche langage à la conception d'ordonnanceurs de processus. In *Rencontres francophones en Parallélisme, Architecture, Système et Composant (RenPar 14)*, Hammamet, Tunisie, Apr. 2002.

[7] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta. Cellss: a programming model for the cell be architecture. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 86, New York, NY, USA, 2006. ACM.

[8] R. Blikberg and T. Sørevik. Load balancing and OpenMP implementation of nested parallelism. *Parallel Computing*, 31(10-12):984–998, Oct. 2005.

[9] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures, 2007.

[10] 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.

[11] C. H. Crawford, P. Henning, M. Kistler, and C. Wright. Accelerating computing with the cell broadband engine processor. In *CF '08*, 2008.

[12] V. Danjean, R. Namyst, and R. Russell. Integrating kernel activations in a multithreaded runtime system on Linux. In *Parallel and Distributed Processing. Proc. 4th Workshop on Runtime Systems for Parallel Programming (RTSPP '00)*, volume 1800 of *Lect. Notes in Comp. Science*, pages 1160–1167, Cancun, Mexico, May 2000. En conjonction avec IPDPS 2000. IEEE TCPP and ACM, Springer-Verlag.

[13] V. Danjean, R. Namyst, and P.-A. Wacrenier. An efficient multi-level trace toolkit for multi-threaded applications. In *EuroPar*, Lisbonne, Portugal, Sept. 2005.

[14] R. F. V. der Wijngaart and H. Jin. NAS Parallel Benchmarks, Multi-Zone Versions. Technical Report NAS-03-010, NASA Advanced Supercomputing (NAS) Division, 2003.

[15] R. Dolbeau, S. Bihan, and F. Bodin. HMPP: A hybrid multi-core parallel programming environment, 2007.

[16] A. Duran, M. Gonzàles, and J. Corbalán. Automatic thread distribution for nested parallelism in OpenMP. In *19th ACM International Conference on Supercomputing*, pages 121–130, Cambridge, MA, USA, June 2005.

[17] A. Duran, J. M. Perez, E. Ayguade, R. Badia, and J. Labarta. Extending the openmp tasking model to allow dependant tasks. In *IWOMP Proceedings*, 2008.

[18] D. Durand, T. Montaut, L. Kervella, and W. Jalby. Impact of memory contention on dynamic scheduling on NUMA multiprocessors. In *Int. Conf. on Parallel and Distributed Systems*, volume 7. IEEE, Nov. 1996.

[19] A. Fedorova. *Operating System Scheduling for Chip Multithreaded Processors*. PhD thesis, Harvard University, Cambridge, Massachusetts, 9 2006.

[20] 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.

[21] M. Frigo, C. E. Leiserson, and K. H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Montreal, Canada, June 1998.

[22] G. R. Gao, T. Sterling, R. Stevens, M. Hereld, and W. Zhu. Hierarchical multithreading: programming model and system software. In *20th International Parallel and Distributed Processing Symposium (IPDPS)*, Apr. 2006.

[23] M. Gonzalez, J. Oliver, X. Martorell, E. Ayguade, J. Labarta, and N. Navarro. OpenMP Extensions for Thread Groups and Their Run-Time Support. In *Languages and Compilers for Parallel Computing*. Springer Verlag, 2000.

[24] 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*, Jan. 2000.

[25] D. Kunzman. Charm++ on the Cell Processor. Master's thesis, Dept. of Computer Science, University of Illinois, 2006. http://charm.cs.uiuc.edu/papers/KunzmanMSThesis06.shtml.

[26] H. Löf and S. Holmgren. affinity-on-next-touch: increasing the performance of an industrial PDE solver on a cc-NUMA system. In *19th ACM International Conference on Supercomputing*, pages 387–392, Cambridge, MA, USA, June 2005.

[27] J. Marathe and F. Mueller. Hardware profile-guided automatic page placement for ccnuma systems. In *Sixth Symposium on Principles and Practice of Parallel Programming*, Mar. 2006.

[28] M. D. McCool. Data-parallel programming on the cell be and the gpu using the rapidmind development platform. 2006.

[29] M. Nijhuis, H. Bos, H. Bal, and C. Augonnet. Mapping and synchronizing streaming applications on cell processors. In *International Conference on High Performance Embedded Architectures & Compilers*, 2009.

[30] D. S. Nikolopoulos, T. S. Papatheodorou, C. D. Polychronopoulos, J. Labarta, and E. Ayguadé. User-level dynamic page migration for multiprogrammed shared-memory multiprocessors. In *International Conference on Parallel Processing*, pages 95–103. IEEE, Sept. 2000.

[31] M. Ohara, H. Inoue, Y. Sohda, H. Komatsu, and T. Nakatani. Mpi microtask for programming the cell broadband enginetm processor. *IBM Syst. J.*, 45(1), 2006.

[32] Y. Ohtake, A. Belyaev, M. Alexa, G. Turk, and H.-P. Seidel. Multi-level partition of unity implicits. *ACM Trans. Graph.*, 22(3):463–470, 2003.

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

[34] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 03 2007.

[35] S. Pakin. Receiver-initiated Message Passing over RDMA Networks. In *IPDPS'08*.

[36] 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 Irregular'2000, Cancun, Mexique*, pages 519–525. Springer Verlag, 2000.

[37] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, V. Menon, T. Shpeisman, M. Rajagopalan, A. Ghuloum, E. Sprangle, A. Rohillah, and D. Carmean. Runtime Environment for Tera-scale Platforms, Aug. 2007.

[38] sgi. *Process Aggregates*. `http://oss.sgi.com/projects/pagg/`.

[39] X. Shen, Y. Gao, C. Ding, and R. Archambault. Lightweight reference affinity analysis. In *19th ACM International Conference on Supercomputing*, pages 131–140, Cambridge, MA, USA, June 2005.

[40] M. Steckermeier and F. Bellosa. Using locality information in user-level scheduling. Technical Report TR-95-14, University of Erlangen-Nürnberg, Dec. 1995. `http://www4.informatik.uni-erlangen.de/Projects/FORTWIHR/ELiTE/`.

[41] Y. Tanaka, K. Taura, M. Sato, and A. Yonezawa. Performance evaluation of openmp applications with nested parallelism. In *Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 100–112, 2000.

[42] S. Thibault. BubbleSched API. `http://pm2.gforge.inria.fr/marcel/doc/html/`.

[43] S. Thibault. A flexible thread scheduler for hierarchical multiprocessor machines. In *Second International Workshop on Operating Systems, Programming Environments and Management Tools for High-Performance Computing on Clusters (COSET-2)*, Cambridge / USA, 06 2005. ICS / ACM / IRISA.

[44] S. Thibault, R. Namyst, and P.-A. Wacrenier. Building Portable Thread Schedulers for Hierarchical Multiprocessors: the BubbleSched Framework. In *EuroPar*, Rennes,France, 8 2007. ACM.

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

[46] L. Wesolowski. An application programming interface for general purpose graphics processing units in an asynchronous runtime system. Master's thesis, 2008.

[47] R. C. Whaley and J. Dongarra. Automatically Tuned Linear Algebra Software. In *Ninth SIAM Conference on Parallel Processing for Scientific Computing*, 1999.

[48] G. Zhang. Extending the OpenMP standard for thread mapping and grouping. In *Second International Workshop on OpenMP (IWOMP 2006)*, Reims, France, 2006.