



Structuring the execution of OpenMP applications for multicore architectures

François Broquedis, Olivier Aumage, Brice Goglin,
Samuel Thibault, Pierre-Andr Wacrenier, Raymond Namyst

INRIA, LaBRI, Université of Bordeaux

F-33405 Talence – France

Email: {broquedi,aumage,goglin,thibault,wacren,namyst}@labri.fr

Abstract—The now commonplace multi-core chips have introduced, by design, a deep hierarchy of memory and cache banks within parallel computers as a tradeoff between the user friendliness of shared memory on the one side, and memory access scalability and efficiency on the other side. However, to get high performance out of such machines requires a dynamic mapping of application tasks and data onto the underlying architecture. Moreover, depending on the application behavior, this mapping should favor cache affinity, memory bandwidth, computation synchrony, or a combination of these. The great challenge is then to perform this hardware-dependent mapping in a portable, abstract way.

To meet this need, we propose a new, hierarchical approach to the execution of OpenMP threads onto multicore machines. Our ForestGOMP runtime system dynamically generates structured trees out of OpenMP programs. It collects relationship information about threads and data as well. This information is used together with scheduling hints and hardware counter feedback by the scheduler to select the most appropriate threads and data distribution. ForestGOMP features a high-level platform for developing and tuning portable threads schedulers. We present several applications for which we developed specific scheduling policies that achieve excellent speedups on 16-core machines.

I. INTRODUCTION

Typical nowadays computer architectures are based on a non uniform memory access (NUMA) interconnect of processing units based on multicores and simultaneous multithreading technologies. They can be seen as nested sets of resources with varying degrees, tightness and complexity of relationships, from a general point of view. The nature of these relationships being the *sharing* (or not) of some memory bank, cache, or even internal processing circuitry in the case of SMT.

Sharing has potentials for efficiency as a source of rationalization, but the other side of the coin is that sharing also has potentials for inefficiency as well, as a source of contention. Exploiting these machines efficiently is therefore a real challenge and dilemmas come 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.

As a consequence of their hierarchical nature, nowadays

architectures feature numerous degrees of freedom. The layout, the number and the respective size of the nested sets of resources may considerably differ from one machine to another one. Therefore, the efficient programming of multicore and NUMA architectures requires an in-depth knowledge of both the target computer architecture and the application behavior like data sharing, synchronizations, memory access patterns, affinity and inter-thread relationship. As a consequence, the programmer should be given means to hint and guide the execution of his application.

Nevertheless, even if experienced programmers are able to develop efficient applications for a specific hardware configuration, most of them rely on the operating system to get performance out of their parallel programs. Thus, the application programmer should be able to decide how much effort he wants to invest in his application parallelization process, and how much control he is willing to leave to the operating system. However, while the core scheduler of operating systems can often be influenced, it misses the ability to get accurate information about application behavior: for instance, adaptively-refined meshes entail very irregular and unpredictable behavior. This is why we claim that we can only reach the portability of performance by letting the runtime system be in charge of controlling thread scheduling and data placement layouts with the help of the application programmer. The runtime system can in turn cooperate with the operating system and gather hardware counters in the name of the application programmer, thus adapting the programmer requests to the current system state and the available resources.

The runtime system plays central role here. Its job is to adapt the execution of the application to the resources allocated by the operating system. Now, while the scheduling decisions should be left to the runtime, the whole software stack, from the application to the operating system, should be involved in the parallelizing and locality adaptation by providing useful information to the other components. In particular, it is important for the application to expose a structured view of its parallelism so that the runtime can later map it onto the allocated resources. For instance, in OpenMP frameworks, the information extracted by the compiler about memory affinity and adherence to the same parallel section can benefit in the guidance of task/thread scheduling. Such

an information may also help in selecting a suitable memory bank location for pieces of data, or moving data to a new location to follow the evolution of the application state and behavior.

Our approach builds on these ideas to provide an OpenMP runtime system tightly integrated with a hardware-aware scheduler for structured execution of applications on nowadays hierarchical shared-memory computers. It combines the principles of a tree-like application parallelism structuring extracted with the help of the OpenMP compiler front-end and a tree-like modeling of the underlying architecture. It then dynamically maps the continuously evolving application parallelism tree onto the architecture tree, migrating threads and/or data when needed so as to maintain a good balancing of threads when events arise (task termination, new parallel sections, etc.).

This paper is organized as follows. Section 2 presents our FORESTGOMP OpenMP runtime system to exploit multicore and NUMA architectures through high level guided parallel execution. FORESTGOMP cooperates with our BUBBLESCHED programmable thread scheduling framework. Section 3 and Section 4 respectively introduce and evaluate the BUBBLESCHED policy we designed to schedule OpenMP programs onto multicore architectures and the complementary policy we designed for the NUMA case. Section 6 discusses relationships with other works from the community and Section 7 concludes the paper.

II. FORESTGOMP: A HIGH LEVEL WAY TO GUIDE PARALLEL EXECUTION

To fully tap into the potential of multicore machines, we have designed a platform for experimenting with and tuning scheduling policies of threads and tasks generated by OpenMP programs.

A. Rationale

Up to now, the OpenMP specification [1] has been built on the assumption that the underlying target architecture is a set of identical, independent computing units with a flat, isochronic shared memory space. For many years, this assumption was reasonable, but it is not anymore. Multicore and NUMA architectures break the assumption and offer new challenges to the application programmer that the OpenMP specification is not yet ready to take up.

Parallel programming approaches on such architectures have to deal with thread and data placement which OpenMP ignores. This is why the OpenMP ARB is currently discussing possible evolutions providing ways to group related threads together, place them according to their affinities, and distribute the data they access. These evolutions are promising but are mostly designed for the experienced programmers. The application programmer still has to make assumptions on parameters he cannot guess that directly depends on the underlying system state, like the amount of free

memory or memory contention. This may lead to developing non-portable applications by defining the number of threads devoted to a parallel section and explicitly pinning threads onto the cores. Explicit binding and extended scheduling controls are desirable for advanced users to experiment with accurate low-level placement and understanding performance, but for the regular scientific application programmer there is a need for a more abstract, higher level approach.

The OpenMP specification was designed keeping in mind that the more accurate an OpenMP application is, the better performance it obtains. The OpenMP language implements this philosophy by providing high level clauses and keywords that help the programmer with parallelizing his application. For example, the user can define the minimal length of a chunk of the iteration space a parallel loop. Additional parameters and keywords helps the programmer defining the loop scheduling behavior or his parallelism grain.

Our approach extends the OpenMP philosophy of an incremental parallelization work from the application programmer. It advocates the idea that the programmer can help in the mapping of his application on the multicore or NUMA underlying architecture by progressively providing the runtime system with the additional information it needs to get a global view of thread and data relationship. The task of the runtime system is then to structure that parallelism to make an efficient use of hierarchical architectures.

B. The ForestGOMP runtime system

We developed the FORESTGOMP runtime to meet all these needs. FORESTGOMP is an extension of the GNU OpenMP library (LIBGOMP) that automatically structures the parallelism of OpenMP applications relying on the BUBBLESCHED platform. BUBBLESCHED provides a way to group related threads together into structures called *Bubbles* [2]. This way, the scheduler has a persistent view of the threads relationship that can be defined by sharing data or synchronizing a lot for example. As bubbles can contain nested bubbles, nested parallelism ends up with generating a tree of threads.

OpenMP parallel regions create threads to parallelize a sequential job. These threads usually share data and often needs to synchronize with each other. Thus the FORESTGOMP runtime groups these threads into the same bubble structure. More generally, a bubble is created every time the application creates parallelism. As nested parallel regions generate nested bubbles, FORESTGOMP automatically builds a tree of OpenMP thread teams out of any OpenMP applications. This gives the FORESTGOMP scheduler a persistent and global view of threads likely to work together.

The runtime then needs to efficiently map this structured parallelism onto any hierarchical architecture. To do so, we believe that the operating system should not completely hide the computer architecture. Therefore, our runtime builds a

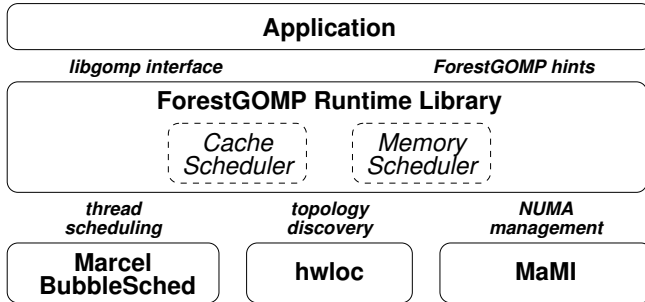


Figure 1. Description of the FORESTGOMP runtime system, its schedulers that are combined to tackle cache and memory affinity, and its interaction with the application and low-level libraries.

generic representation of the architecture with the help of the OS as *hierarchical runqueues*. A runqueue is a list containing threads and bubbles to be executed. We associate a runqueue to any level of the architecture: we define core-level runqueues, processor-level runqueues, NUMA node-level runqueues that define scheduling domains. For instance, scheduling a thread on a node-level runqueue means this thread can only be executed by one of the core this NUMA node contains. This way, one may constrain the execution of OpenMP teams to a specific part of the computer architecture.

To sum it up, FORESTGOMP structures the parallelism by generating a tree of threads and bubbles out of OpenMP applications and models hierarchical architectures with a tree of runqueues. So the problem of scheduling boils down to mapping a tree of threads onto a tree of runqueues. Many ways exist to implement this mapping, depending on the application nature and needs. The FORESTGOMP runtime provides several mapping policies called *bubble schedulers*. One may even design his own specific bubble scheduler thanks to a programming interface that helps with the details of bubbles and runqueues management.

The FORESTGOMP runtime system is implemented as a shared library that is binary compatible with LIBGOMP and extends it by adding a specialized affinity hint interface. As depicted on Figure 1, applications create OpenMP thread teams and let FORESTGOMP transparently schedule them using custom thread schedulers that are for instance invoked when an imbalance or locality issue appears. FORESTGOMP relies on the BUBBLESCHED framework of the MARCEL multithreading library [2] to actually execute these threads according to its scheduling decision.

The knowledge of the underlying hardware is gathered from the HWLOC and MAMI libraries. HWLOC¹ [3] was initially developed internally in BUBBLESCHED to gather portable information about the underlying hardware at startup. It is now available as a standalone library that builds a hierarchical architecture tree composed of objects

describing the hardware (Node, Socket, Cache, Core, and more) and various attributes such as the cache type and size, or the socket number. It provides MPI and OpenMP libraries with a portable programming interface that abstracts the machine hierarchy, offering both hardware information gathering and process and thread binding facilities. MAMI² is a memory interface that was designed to manage memory allocation and placement with regard to NUMA nodes within our user-level thread library in a portable manner. Aside from usual memory allocation policies such as binding or interleaving, it also offers synchronous and *next-touch* memory migration strategies. FORESTGOMP relies on these features to implement its memory affinity directives.

C. An open scheduling framework

The FORESTGOMP platform allows programmers to easily write, experiment with and tune thread scheduling policies. 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.

Our programming interface is centered around primitives for manipulating bubbles and threads among a hierarchy of runqueues reflecting the underlying topology. 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 memory usage, or the cache miss rates. To handle concurrency, one can use either fine-grain locking functions or coarse-grain functions which lock a whole sub-part of the machine (runqueues and bubbles).

Writing a high-level scheduler actually reduces to writing some hook functions. For instance, one of these hooks 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. Another 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

¹<http://runtime.bordeaux.inria.fr/hwloc>

²<http://runtime.bordeaux.inria.fr/MaMI>

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).

D. Raw performance

As preliminary experiments, we tested the LIBGOMP library that comes with GCC 4.4, the INTEL compiler 11.1 and the FORESTGOMP runtime on the EPCC micro-benchmark and the NAS OpenMP parallel benchmark suite. We executed these benchmarks on a quad-socket quad-core Opteron computer while setting the `OMP_NUM_THREADS` environment variable to 16 to get one thread per core. In that case, and more generally when dealing with flat parallelism, the FORESTGOMP runtime binds one thread per core. Figure II shows the overhead of each runtime system for each OpenMP construct obtained with the EPCC micro-benchmark. Figure I shows the speed-up of the B class of each application from the NAS OpenMP parallel benchmark suite. The calculated speed-ups refer to the sequential execution time of each application compiled with the INTEL compiler. The FORESTGOMP runtime behaves better on both BT and MG applications thanks to a more stable thread distribution. The INTEL runtime obtains better speed-ups on EP and SP thanks to better compiler optimizations. Indeed, the last two lines of table I show that serial execution times compiled with ICC and GCC (without OpenMP directives) differ a lot for these applications. INTEL locks are more efficient than the ones provided by the LIBGOMP and FORESTGOMP runtimes, so the INTEL runtime behaves better on the UA application which generates more than 300000 locks. The performance obtained with the CG, FT and LU applications (exceeding 16 of speed-up thanks to cache effects) is very similar. FORESTGOMP only adds a small overhead induced by the management of the bubble structures.

III. DESIGNING MULTICORE-AWARE SCHEDULING POLICIES

OpenMP developers are used to explicitly drive the parallel behavior of their code but the increasing complexity of computer architectures now renders this process difficult. Multicore architectures expose many processing units often organized in a hierarchical way, sharing multiple levels of cache memory. This explains why parallel languages generating flat parallelism often fail to exploit these architectures at their full potential (benefit from cache memory reuse, fast synchronizations, locality commodities).

Nested parallelism seems to be a natural way to fit the hierarchical structure of parallel computers. However, considering two running nested teams for instance, the first team to finish its work will make a whole set of cores idle, waiting for the other one to complete. Thus, to exploit this paradigm efficiently, it is necessary to create more OpenMP

threads than cores or to dynamically allocate new cores to teams. This comes with a price that needs to be mitigated by the runtime system. For instance, Table III shows how the FORESTGOMP barrier was tailored in this aim.

A. A scheduling strategy to deal with multiple levels of cache memory

We previously presented a scheduler named *Affinity* [4] and showed that FORESTGOMP was able to schedule a very irregular application, called MPU, computing an implicit surface reconstruction which produces more than 30 000 threads per second on a 16-core computer. We present here the *Cache* scheduler, an evolution of *Affinity*, which differs in its initial distribution and by the fact *Cache* is able to take the thread workload into account. The *Cache* bubble scheduler main goal is to maximize cache memory reuse by scheduling related threads together onto cache-sharing cores. In the context of OpenMP, this means to keep the OpenMP teammates together as long as possible. FORESTGOMP structures OpenMP parallelism by grouping threads created from a parallel region into the same bubble. Taking cache affinity into account reduces to preventing the scheduler from breaking these bubbles. The *Cache* distribution algorithm first sorts the teams to schedule depending on their workload. This way, when there are more teams than cores to occupy, a greedy distribution guarantees that the scheduler will not have to burst any bubble. Otherwise, *Cache* decides to burst the bubble containing the greatest number of threads first. The scheduler maintains statistics about where the threads were scheduled the last time the *Cache* algorithm was called, to be able to place them back at their previous location and maximize cache memory reuse. The *Cache* scheduler also implements a work stealing algorithm, inspired from existing programming models like *Cilk* or *TBB*, with the exception that the *Cache* scheduler starts to look for threads to steal locally first, in accordance with cache affinities expressed by bubbles. The way the *Cache* work stealing algorithm browses the architecture topology has a direct impact on the overall application performance.

B. Evaluation with the BT-MZ application

We also experimented 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 3.3. 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

Table I
SPEED-UP OF THE NAS OPENMP PARALLEL BENCHMARK SUITE, COMPARED TO THE ICC AND GCC SERIAL TIME, ON A QUAD-SOCKET QUAD-CORE OPTERON COMPUTER.

Runtime	BT	CG	EP	FT	LU	MG	SP	UA
LIBGOMP3 vs. GCC serial	12.4	14.0	15.8	12.2	19.8	5.8	4.3	7.0
FORESTGOMP vs. GCC serial	13.1	14.0	15.6	12.0	19.8	6.4	4.8	3.3
LIBGOMP3 vs. ICC serial	11.5	12.4	9.1	10.1	17.0	5.1	3.2	6.6
FORESTGOMP vs. ICC serial	12.1	12.4	9.0	9.9	17.0	5.6	3.6	3.1
INTEL ICC vs. ICC serial	11.2	12.3	14.9	10.1	17.9	4.7	5.3	4.4
GCC serial (s)	693	369	152	120	913	22.7	655	420
INTEL ICC serial (s)	642	327	87.8	99.1	782	20.0	493	397

Table II
RUNTIME OVERHEAD (US) OBTAINED WITH THE EPCC MICRO-BENCHMARK ON A QUAD-SOCKET QUAD-CORE OPTERON COMPUTER.

Runtime	atomic	barrier	critical	for	lock	parallel	parallel for	reduction	single
LIBGOMP3	0.30	2.91	4.01	2.91	3.95	7.15	8.15	8.74	3.74
INTEL ICC	0.39	5.12	2.20	5.07	2.21	8.15	8.18	15.66	19.50
FORESTGOMP	0.29	3.31	4.03	3.34	3.88	7.02	7.46	7.58	2.56

Table III
EPCC BENCHMARK: OVERHEAD (US) OF THE BARRIER CONSTRUCT DEPENDING ON THE NUMBER OF THREADS ON A QUAD-SOCKET QUAD-CORE OPTERON COMPUTER.

Runtime	16 threads	32 threads	64 threads
LIBGOMP3	2.91	187.9 ($\times 64$)	371.1 ($\times 128$)
INTEL ICC	4.93	17.98 ($\times 3.64$)	63.29 ($\times 12.84$)
FORESTGOMP	3.31	3.87 ($\times 1.17$)	5.15 ($\times 1.56$)

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.* [5] and Jin *et al.* [6], we modified this to use two nested OpenMP static parallel sections instead, using $n_o * n_i$ threads.

This application differs from MPU in the way the application programmer knows the workload associated with each zone. Transmitting this information to the runtime system can help guiding the scheduling to favor load balancing. The *Cache* distribution algorithm takes workload information into account when there are more groups of threads to distribute than cores to occupy. In that case, the scheduler performs a greedy distribution considering the workload of each team.

Table IV shows the speed-ups obtained by the LIBGOMP and INTEL runtime systems, and different versions of the *Cache* scheduler that comes with FORESTGOMP, depending on the number of threads created from outer and inner parallel regions. We tested the C class of the BT-MZ application on a quad-socket quad-core computer. First, the performance confirms the FORESTGOMP runtime was designed for nested parallelism as it behaves better, for any combination of outer and inner number of threads, than the LIBGOMP and INTEL libraries. The *Cache* scheduler behaves

best with 32 teams of 8 threads. Creating more threads than processors in this application offers the *Cache* scheduler ways to steal work when a processor idles, reaching this way a speed-up of 14.48. The column *Cache + load info* shows the performance obtained when the application programmer provides the workload associated to each zone to compute. This way, the *Cache* scheduler distribution algorithm takes the load into account when distributing the teams thus minimizing the need to call the work stealing algorithm. Such a distribution obtains a speed-up of 15.05. Alternatively, we slightly modified the *Cache* distribution algorithm to improve the load balance. In this version, instead of distributing all the teams over the core-level runqueues, we pick the most-loaded runqueue of each NUMA node and put its workload on the NUMA-level runqueue. This way, when a core completes the jobs associated to its runqueue, it starts helping out the most-loaded core associated to its NUMA node without having to call the work stealing algorithm. This strategy leads to a speed-up of 15.25, the best performance we obtained on the BT-MZ application, thanks to the fact that *Cache* distributes the OpenMP teams in a deterministic way that always leads to the same distribution, thus improving locality. This strategy also guarantees that teams with the highest workload will be executed first reducing the ending idle time.

Several OpenMP language extensions have been proposed to control the allocation of work to the participating threads.

Table IV
SPEED-UPS OBTAINED WITH THE BT-MZ (CLASS C) APPLICATION, COMPARED TO THE ICC SERIAL TIME, ON A QUAD-SOCKET QUAD-CORE OPTERON COMPUTER DEPENDING ON THE NUMBER OF THREADS CREATED FROM OUTER AND INNER PARALLEL REGIONS.

Outer × Inner	LIBGOMP3	INTEL	Cache	Cache + load info	Tuned Cache + load info
4×4	9.4	13.8	14.1	14.1	14.1
16×1	14.1	13.9	14.1	14.1	14.1
16×2	11.8	9.2	14.1	14.2	14.3
16×4	11.6	6.1	14.1	14.9	14.9
16×8	11.5	4.0	14.4	15.0	15.2
32×1	12.6	10.3	13.5	13.8	13.8
32×2	11.6	5.9	14.2	14.2	14.3
32×4	11.2	3.4	14.3	14.8	14.8
32×8	10.9	2.8	14.5	14.7	14.7

In order to favor affinities in a portable manner the NANOS compiler [5] allows to associate groups of threads with parallel regions in a static way in order to always execute the same thread on the same core. The OpenUH Compiler [7] proposes a mechanism to accurately select the threads of a subteam to define the thread-core mapping for better data locality, although this proposition does not involve nested parallelism.

Both mechanisms can lead to the performance FORESTGOMP obtains on the BT-MZ benchmark. Indeed, the regular application scheme makes the best thread distribution feasible by explicit thread binding. However, the next section presents experiments in which the thread distribution needs to be reviewed dynamically to achieve the best performance.

IV. DESIGNING MEMORY-AWARE SCHEDULING POLICIES

FORESTGOMP achieves interesting speedups on previous examples thanks to regular memory access patterns. However, in the general case, the application may exhibit irregular access patterns that must be taken into account when load-balancing threads. Thread teams must be able to migrate together with their datasets in memory so as to maintain good locality and load balance.

A. Dealing with NUMA constraints at runtime

To keep designing larger scale configurations, computer architects propose ways to connect multicore chips together relying on technologies like AMD HYPERTRANSPORT or INTEL QPI. The resulting computers burden the application programmer with NUMA penalties and memory contention, making him concerned by memory affinity and thread/memory locality. Some software support already exists to help with limiting the amount of remote memory accesses in parallel applications, like the *first-touch* and *next-touch* allocation policies. *first-touch* makes the operating system allocate the considered pages next to the first thread to access them. It helps with improving thread/data locality, but assumes the memory access patterns will not change during the entire execution. *next-touch* asks the operating system to migrate the considered pages next to

the next thread to access them. It helps with improving the performance of irregular applications, but is not widely available. More generally, both policies suffers from the same issues. First, they assume the first thread to touch the page will be the one that will work on them later. They also ignore the underlying system state while migration may suffer from out-of-memory situations or contention on the bus. These issues vary during the execution, forcing the thread and data distribution to adapt dynamically. That is why our approach is to let the runtime system in charge of threads and data placement. It is easier to achieve a sharper jointed distribution of threads and data with the precious hints the application programmer (or compiler) can provide : programmers (or compilers) may have a global view of threads and data relationship while a runtime can only discover them on the fly.

When it comes to designing applications for NUMA architectures, the application programmer has no other option than relying on the operating system. From our point of view, the runtime system is more likely to communicate efficiently with the operating system. Expressing his needs to the runtime system keeps the application programmer from designing non-portable applications. That is why, in a NUMA context, FORESTGOMP provides a programming interface for the programmer to express memory affinity from his application [8]. He no longer needs to be concerned about the underlying system state. All he has to do is to express his application patterns in a generic way. FORESTGOMP provides two functions to do so: the first one sets the memory affinity for an OpenMP team to be born, and the second one sets the affinity for the caller thread from inside a parallel region. While the first one allows the runtime to perform early optimizations like directly creating the threads on the right runqueues, the second one is also needed when the memory access pattern changes inside a parallel region.

B. The Memory bubble scheduler

Now that the application programmer is able to transmit the memory affinities from his application to the runtime system, we need a scheduling strategy to perform a sound distribution of threads and data regarding the gathered

information. This section introduces the *Memory* bubble scheduler.

Scheduling while preserving memory affinities: Every FORESTGOMP threads and teams now have some attached memory areas the runtime system can see. The *Memory* scheduler main goal is to distribute the threads and migrate the attached data to make every OpenMP team access local memory. To do so, it implements a three-phases distribution algorithm. The first algorithm phase consists in attracting the threads to the NUMA node holding the higher amount of their data. This information can be summarized onto bubbles. A bubble will explode if it contains threads attracted to different runqueues for the scheduler to be able to satisfy every expressed affinity. The phase one can lead to an unbalanced distribution. Indeed, some applications may allocate the accessed data on a small group of nodes, which means some nodes would not have any threads to execute at the end of phase one. To make sure every core will be occupied at the end of the distribution, the second algorithm phase stretches some memory affinities to balance the load on the computer. We move the threads with the smallest amount of attached memory first to do so (including threads with untouched memory). The last phase of the distribution algorithm is a matter of migrating the remotely-located data to the new threads location. As threads were driven to the node holding the bigger amount of their data during phase one, the *Memory* scheduler is guaranteed to migrate as less data as possible. This aspect has a real impact on performance, as migrating data is far more expensive than moving FORESTGOMP threads. When the distribution over the node level runqueues is over, an instance of the *Cache* scheduler is called inside each NUMA node to perform a cache-aware threads distribution.

Updating the threads and data distribution at the right time: Updating the threads and data distribution is a sensitive mechanism that influences the overall application performance. Calling the bubble scheduler too often will slow down the execution, as the bubble scheduler will browse the architecture topology to figure out the updates to perform on the current distribution. On the other hand, you will not get the best performance if the scheduler do not react soon enough in case of a major access pattern change. FORESTGOMP adopts a two-ways mechanism to decide how often the distribution needs to be updated. First, every time the application programmer updates the memory affinities, the bubble scheduler is called to check the current distribution. This approach may not be sufficient for irregular applications, so FORESTGOMP also provides a more dynamic mechanism based on hardware counters inspecting. The runtime checks the counters on a regular basis and infers the amount of remote memory accesses initiated from the current processor while defining a threshold from which FORESTGOMP will call the scheduler for checking the current distribution. These two approaches are

complementary. Indeed, in some cases updates from the application programmer will not need the scheduler to rethink the current distribution. In other cases the programmer is able to roughly define which part of his application will work on which data, but cannot tell precisely when and how. Hardware counters can help reacting at the right time for these situations.

C. Evaluation

We evaluate our approach on the Twisted-STREAM and LU matrix factorization benchmarks.

Twisted-STREAM: STREAM³ is a synthetic benchmark parallelized using OpenMP that measures sustainable memory bandwidth and the corresponding computation rate for simple vectors. The three input vectors are wide enough to limit the cache memory benefits (20 millions double precision floats) and are initialized in parallel using a *first-touch* allocation policy to get the corresponding memory pages close to the thread that will access them.

To complicate the STREAM memory access pattern, we designed the Twisted-STREAM benchmark in which we use nested OpenMP parallel regions. The benchmark now creates one team per NUMA node of the computer that works on its own set of STREAM vectors initialized in parallel, as in the original version of STREAM. To fit the target computer, a quad-socket quad-core OPTERON computer, the benchmark creates four teams of four threads.

Twisted-STREAM contains two distinct phases. During the second phase, each team works on a different data set than the one it was given in the first phase. We also modified the original benchmark so as to make the threads workload vary. The user can so specify how many STREAM iterations each team will have to compute.

The *first-touch* allocation policy only gives good results for the first phase as shown in table V. FORESTGOMP achieves the best and most stable performance on phase 1. The underlying bubble scheduler distributes the threads by the time the outer parallel region is reached. Each thread is permanently placed on one NUMA node of the computer. Furthermore, FORESTGOMP creates the teammates threads where the master thread of the team is currently located. As the vectors accessed by the teammates have been touched by the master thread, this guarantees the threads and the memory are located on the same NUMA node, and thus explains the good performance we obtain during phase one.

A typical solution to the lack of performance observed during phase two seems to rely on a *next-touch* page migration between the two phases of the application. However we show in the remaining of this section that *next-touch* is not always the best answer to the memory locality problem. We tested two different versions of the Twisted-STREAM benchmark: *Twisted-100*, in which all of the three vectors are

³<http://www.cs.virginia.edu/stream/>

Table V
 AVERAGE RATES PER NODE OBSERVED WITH THE TWISTED-STREAM BENCHMARK USING A *first-touch* ALLOCATION POLICY. DURING PHASE 2, THREADS ACCESS DATA ON A DIFFERENT NUMA NODE.

	LIBGOMP	INTEL	FORESTGOMP
Phase 1	2 ± 1 GB/s	3 ± 0.5 GB/s	3.6 ± 0.2 GB/s
Phase 2	1.3 ± 1 GB/s	2 ± 0.5 GB/s	1.7 ± 0.2 GB/s

remotely-located during phase 2, and *Twisted-66* in which only two of the three vectors are located on a remote NUMA node. It is worth noticing that we obtained performance shown in table V with *Twisted-100*.

Table VI shows the normalized execution times of the Twisted-100 benchmark compared to the performance obtained by the LIBGOMP runtime. In this case, all data is remotely located during phase 2. *next-touch* migrates the data to the location of the threads, which helps improving the performance when the threads workload is big enough. The *Memory* bubble scheduler prefers to move the threads to the location of the data. Indeed, the STREAM benchmark works on three 160MB-vectors. As migrating 480MB of memory is far more expensive than moving 16 FORESTGOMP threads, FORESTGOMP obtains the best performance here, whatever threads workload we set.

In the case of the Twisted-66 benchmark, only two of the three STREAM vectors are located on a remote NUMA node during phase 2. This time, the *Memory* bubble scheduler will first attract to threads to the location of the two remote vectors, and then migrate the remaining one. This way, FORESTGOMP only migrates one vector, instead of migrating two of them with a *next-touch* approach. Table VII shows the *Memory* scheduler algorithm offers performance gain sooner, and still behaves better than *next-touch* when the workload increases.

LU Matrix Factorization: We now look at a threaded LU matrix factorization. As usual, the implementation splits the matrix into smaller data blocks that are actually manipulated by a BLAS library (ATLAS). During each step, a new *pivot* block is computed on the diagonal. Then, the values for the corresponding column and row are updated as well as the ones for the remaining bottom-right blocks. This is done using *for* loops that we parallelized thanks to OpenMP parallel for pragmas. Data is initially distributed across all memory nodes in a interleaved manner so as to maximize the memory throughput. We measure a 388.9s factorization time (60.3 Gflop/s) for a 32k-wide matrix divided in 64 blocks per dimension.

This program is actually a good example of application where the developer cannot easily give useful hints about memory access patterns. Indeed, the number of blocks involved in each computation step decreases quadratically, causing threads to work on highly different blocks at each step. Therefore, instead of trying to understand these patterns

and place threads and data jointly, we use a lazy *next-touch* approach. The whole matrix is marked as *next-touch* when the hardware counters overcome the FORESTGOMP remote accesses threshold so that the data is redistributed among the NUMA nodes when needed, depending on OpenMP thread access patterns. The factorization time decreases by 30% down to 298.2s (80.78 Gflop/s) thanks to this strategy that requires a moderate amount of work from the developer. Other matrix sizes show similar behavior but the block size must be carefully chosen so that the *next-touch* strategy is not disturbed by single pages being shared between multiple blocks/threads.

If migrating pages on *next-write* only, another 2.7% improvement is even obtained. We understand this result as the fact that *Writers* should be privileged in this algorithm since placing pages near them improves performance over placing pages near the first reader or writer.

V. RELATED WORK

Memory placement has been studied in the context of OpenMP through some proposals towards data distribution directives *la* HPF [9], [10]. Such directives are useful to organize data the right way to maximize page locality, and, in our research context, a way to transmit affinity information to our runtime system without heavy modifications of the user application. Automatic page migration [11] was also proposed as an innovative way to maintain locality in applications with regular memory access pattern. The convenient *next-touch* policy was studied as a way to tackle irregular algorithms which suffer from the lack of cooperation between the scheduler and the allocation library [12], [13].

Multilevel and nested parallelism has long been emphasized as a promising path toward scalability with OpenMP [14], [15], gradually brought compiler researchers and vendors to put more of their efforts on OpenMP nested parallelism. Extensions to control the binding of OpenMP threads were also proposed [5], [7]. Beyond the specific context of OpenMP, user-level multithreading was also the target of the INTEL Many-Core RunTime [16] environment. 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. It however doesn't provide any framework for developing dedicated schedulers, and thus can only provide a limited range of scheduling abilities.

Table VI
NORMALIZED EXECUTION TIMES OF THE TWISTED-100 BENCHMARK (1.00 IS LIBGOMP) DEPENDING ON THE NUMBER OF STREAM ITERATIONS.

	1	2	4	8	16	32	64	128
<i>next-touch</i>	5.92	3.28	2.06	1.36	1.07	0.99	0.97	0.95
FORESTGOMP	0.75	0.74	0.74	0.74	0.74	0.74	0.74	0.74

Table VII
NORMALIZED EXECUTION TIMES OF THE TWISTED-66 BENCHMARK (1.00 IS LIBGOMP) DEPENDING ON THE NUMBER OF STREAM ITERATIONS.

	1	2	4	8	...	128	256	512	1024
<i>next-touch</i>	1.72	1.3	1.1	1	...	0.96	0.94	0.93	0.93
FORESTGOMP	1.29	1.08	0.99	0.94	...	0.90	0.88	0.86	0.85

Several operating systems provide facilities for distributing kernel threads along the machine by grouping them into sets: liblgroup on SolarisNSG on Tru64 and 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*.

In the context of high productivity computing systems, PGAS languages (Chapel, UPC, X10,...) and Fortress provide programmers with ways to control where particular threads run and how large objects are laid out in memory. For instance, Fortress proposes a tree of *regions* to describe the structure of the machine on which a program is run and provides means to place a thread at a given region. It may be interesting to associate a region with a bubble and have our threading library schedule Fortress threads: the Fortress programmer could therefore express affinities and could benefit from dynamic scheduling mechanisms we propose.

VI. CONCLUSION AND FUTURE WORK

FORESTGOMP is a platform for executing and tuning OpenMP programs over hierarchical multicore architectures. Its main component is an efficient runtime system capable of scheduling trees of threads generated by an OpenMP compiler. These trees both capture application parallelism and serve as a vehicle for carrying thread relationship and memory affinity information. Because this information is maintained by the runtime system during the whole application run, it can be used to perform appropriate redistributions of threads or data whenever it is necessary.

We have shown the benefits of designing and tuning specific scheduling policies, according to the general behavior of the application. We have presented the *Cache* and *Memory* bubble schedulers that achieve very high performance on several applications. These schedulers are written in a portable way, independently from the target architecture, which demonstrates the relevance of our approach.

There are several research directions we intend to address in the near future. We plan to provide the application programmer with tools to mark memory areas that should be attached to a thread upon the next read or write touch. This mechanism would help the runtime system to better infer the memory affinities, especially when the memory access patterns become too complex to be defined *a priori* by the programmer.

Our proposal is in line with the recent efforts of the OpenMP Architecture Review Board which is currently working on the next evolution of the specification towards a satisfying support of hierarchical, multicore architectures. In particular, the next release will feature new directives for specifying affinity between threads and data. Our proposal of a runtime system able to handle this information is complementary.

In the longer run, we plan to explore ways to compose our scheduling strategies with other schedulers and paradigms. For instance, parallel languages like *Cilk* or *TBB* rely on runtime systems able to efficiently schedule fine-grain parallelism on SMP architectures. The idea here is for instance to assign instances of *TBB* schedulers to each NUMA node, while letting our *Memory* scheduler distribute the work across the nodes, thus widening the spectrum of flat parallelism approaches to NUMA computers in a portable way.

REFERENCES

- [1] "OpenMP: The OpenMP API specification for parallel programming," <http://openmp.org>.
- [2] S. Thibault, R. Namyst, and P.-A. Wacrenier, "Building Portable Thread Schedulers for Hierarchical Multiprocessors: the BubbleSched Framework," in *European Conference on Parallel Computing (Euro-Par)*. Rennes, France: ACM, Aug. 2007. [Online]. Available: <http://hal.inria.fr/inria-00154506/>
- [3] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst, "hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications," in *Proceedings of the 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP2010)*, Pisa, Italia, Feb. 2010.

- [4] F. Broquedis, F. Diakhat, S. Thibault, O. Aumage, R. Namyst, and P.-A. Wacrenier, "Scheduling Dynamic OpenMP Applications over Multicore Architectures," in *International Workshop on OpenMP (IWOMP)*, West Lafayette, IN, May 2008. [Online]. Available: <http://hal.inria.fr/inria-00329934>
- [5] 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.
- [6] H. Jin, B. Chapman, L. Huang, D. an Mey, and T. Reichstein, "Performance evaluation of a multi-zone application in different OpenMP approaches," *Int. J. Parallel Program.*, vol. 36, no. 3, pp. 312–325, 2008.
- [7] B. M. Chapman, L. Huang, H. Jin, G. Jost, and B. R. de Supinski, "Extending OpenMP Worksharing Directives for Multithreading," in *European Conference on Parallel Computing (Euro-Par)*, 2006.
- [8] F. Broquedis, N. Furmento, B. Goglin, R. Namyst, and P.-A. Wacrenier, "Dynamic Task and Data Placement over NUMA Architectures: an OpenMP Runtime Perspective," in *Evolving OpenMP in an Age of Extreme Parallelism, 5th International Workshop on OpenMP, IWOMP 2009*, ser. Lecture Notes in Computer Science, vol. 5568. Dresden, Germany: Springer, Jun. 2009, pp. 79–92.
- [9] S. Benkner and T. Brandes, "Efficient parallel programming on scalable shared memory systems with High Performance Fortran," in *Concurrency: Practice and Experience*, vol. 14. John Wiley & Sons, 2002, pp. 789–803.
- [10] B. M. Chapman, F. Bregier, A. Patil, and A. Prabhakar, "Achieving performance under OpenMP on ccNUMA and software distributed shared memory systems," in *Concurrency: Practice and Experience*, vol. 14. John Wiley & Sons, 2002, pp. 713–739.
- [11] D. S. Nikolopoulos, T. S. Papatheodorou, C. D. Polychronopoulos, J. Labarta, and E. Ayguad, "User-Level Dynamic Page Migration for Multiprogrammed Shared-Memory Multiprocessors," in *International Conference on Parallel Processing*. IEEE Computer Society Press, Sep. 2000, pp. 95–103.
- [12] H. Lf 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*, Cambridge, MA, USA, Jun. 2005, pp. 387–392.
- [13] C. Terboven, D. an Mey, D. Schmidl, H. Jin, and T. Reichstein, "Data and Thread Affinity in OpenMP Programs," in *MAW '08: Proceedings of the 2008 workshop on Memory access on future processors*. New York, NY, USA: ACM, 2008, pp. 377–384.
- [14] X. Martorell, E. Ayguad, N. Navarro, J. Corbaln, M. Gonzalez, and J. Labarta, "Thread Fork/Join Techniques for Multi-Level Parallelism Exploitation in NUMA Multiprocessors," in *International Conference on SuperComputing*. ACM, 1999, pp. 294–301.
- [15] 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*, 2000, pp. 100–112. [Online]. Available: citeseer.ist.psu.edu/tanaka00performance.html
- [16] 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," *Intel Technology Journal*, vol. 11, no. 3, Aug. 2007.