ForestGOMP: an efficient OpenMP environment for NUMA architectures

François Broquedis, Nathalie Furmento, Brice Goglin, Pierre-André Wacrenier, Raymond Namyst

International Journal of Parallel Programming manuscript No.
(will be inserted by the editor)

ForestGOMP: an efficient OpenMP environment for NUMA architectures

François Broquedis · Nathalie Furmento · Brice Goglin · Pierre-André Wacrenier · Raymond Namyst

Abstract Exploiting the full computational power of current hierarchical multiprocessor machines requires a very careful distribution of threads and data among the underlying non-uniform architecture so as to avoid remote memory access penalties. Directive-based programming languages such as OpenMP, can greatly help to perform such a distribution by providing programmers with an easy way to structure the parallelism of their application and to transmit this information to the runtime system.

Our runtime, which is based on a multi-level thread scheduler combined with a NUMA-aware memory manager, converts this information into *scheduling hints* related to thread-memory affinity issues. These hints enable dynamic load distribution guided by application structure and hardware topology, thus helping to achieve performance portability. Several experiments show that mixed solutions (migrating both threads and data) outperform work-stealing based balancing strategies and *next-touch*-based data distribution policies. These techniques provide insights about additional optimizations.

Keywords OpenMP, Memory, NUMA, Hierarchical Thread Scheduling, Multi-Core

1 Introduction

Modern computing architectures are increasingly parallel. While the *High Performance Computing* landscape is still dominated by large clusters, the degree of parallelism within cluster nodes is increasing. This trend is obviously driven by the emergence of multicore processors that dramatically increase the number of cores, at the expense of a poorer memory bandwidth per core. To minimize memory contention, hardware architects have been forced to go back to a hierarchical organization of cores and memory banks or, in other words, to NUMA architectures (*Non-Uniform Memory Access*). Such architectures are now becoming mainstream thanks to the spreading of AMD HYPERTRANSPORT and INTEL QPI technologies.

F. Broquedis · N. Furmento · B. Goglin · P.A. Wacrenier · R. Namyst
LaBRI – INRIA Bordeaux-Sud-Ouest – University of Bordeaux
351 cours de la Libération – F-33405 Talence – France
E-mail: Francois.Broquedis@labri.fr, Nathalie.Furmento@labri.fr, Brice.Goglin@inria.fr, Pierre-Andre.Wacrenier@labri.fr, Raymond.Namyst@labri.fr

Running parallel applications efficiently on previous generation of multiprocessor machines was mainly a matter of careful task scheduling. In this context, parallel runtime systems such as Cilk [16] or TBB [4] have proved to be very effective. In fact, these approaches can still behave well over hierarchical multicore machines with cache-oblivious applications. However, in the general case, successfully running parallel applications on NUMA architectures requires a careful distribution of tasks and data to avoid "NUMA penalties" [8, 28]. Moreover, applications with strong memory bandwidth requirements need data to be physically allocated on the "right" memory banks in order to reduce contention. This means that high-level information about the application behavior, in terms of memory access patterns or affinity between threads and data, must be conveyed to the runtime system.

Several programming approaches provide means to specify task-memory affinities within parallel applications (OpenMP [3], HPF [18], UPC [11]). However, retrieving affinity relations at runtime is difficult; compilers and runtime systems must tightly cooperate to achieve a sound distribution of thread and data that can dynamically evolve according to the application behavior. Our prior work [10] emphasized the importance of establishing a persistent cooperation between an OpenMP compiler and the underlying runtime system on multicore NUMA machines. We designed FORESTGOMP [10] that extends the GNU OpenMP implementation, GOMP, to make use of the BUBBLESCHED flexible scheduling framework [27]. Our approach has proved to be relevant for applications with nested, massive parallelism.

In this paper, we introduce a major extension to our OpenMP runtime system that connects the thread scheduler to a NUMA-aware memory management subsystem. This new runtime can not only use per-bubble memory allocation information when performing thread re-distributions, it can also perform data migration — either immediately or upon *next-touch*— in situations when it is more appropriate. Actually, it can even combine both. We discuss several of these situations, and give insights about the most influential parameters that should be considered on today's hierarchical multicore machines.

The remainder of this paper is organized as follows. We present the background of our work in Section 2. Section 3 explains our objectives and motivations and describes the software we consider in this work. Section 4 presents our extensions to the FORESTGOMP runtime system that enables dynamic placement of threads and memory. In Section 5, we evaluate the relevance of our proposal with several performance-oriented experiments. Before concluding, related work is summarized in Section 6.

2 Background and Motivations

In this section, we briefly introduce modern memory architectures and how they affect application performance. We detail how existing software techniques try to overcome these issues and discuss their intrusiveness.

2.1 Modern Memory Architectures

The emergence of highly parallel architectures with many multicore processors raised the need to rethink the hardware memory subsystem. While the number of cores per machine quickly increases, memory performance unfortunately does not evolve accordingly. Concurrent accesses to memory buses lead to dramatic contention, causing the overall performance to decrease. This led hardware designers to drop the centralized memory model in

favor of distributed and hierarchical architectures, where memory nodes and caches are directly attached to cores. This design has been widely used in high-end servers based on the ITANIUM processor. It now becomes mainstream since AMD HYPERTRANSPORT (see Figure 5) and the recent INTEL QPI memory interconnects dominate the server market. Indeed, these new memory architectures assemble multiple memory nodes into a single distributed cache-coherent system. It has the advantage of being as convenient to program as regular shared-memory SMP processors, while providing a much higher memory bandwidth and much less contention.

However, while being cache-coherent, these distributed architectures have non-constant physical distance between hardware components, causing their communication time to vary. Indeed, a core accesses its local memory faster than the one attached to other cores. A *memory node*, or *NUMA node*, then consists in a set of cores with uniform memory access cost, and accessing memory near a node is faster than accessing the memory of other NUMA nodes. The corresponding ratio is often referred to as the *NUMA factor*. It generally varies from 1.2 up to 3 depending on the architecture and therefore has a strong impact on application performance [8]. Not only does the application run slower when accessing remote data, but contention may also appear on memory links if two processors access each others' memory nodes. Moreover, the presence of shared caches between cores increases the need to take data locality into account while scheduling tasks, so as to prevent cache lines from bouncing between different sets of cores.

Data location	Local	Local + Neighbors
4 threads on node 0	5151 MB/s	5740 MB/s
4 threads per node (16 total)	4×3635 MB/s	4×2257 MB/s

Table 1 Aggregated bandwidth on a quad-quad-core OPTERON host depending on the machine load (4 or 16 threads) and the location of memory buffers, using four parallel STREAM [20] instances.

To illustrate this problem, we ran some experiments on a quad-socket quad-core OPTERON machine. Second row of Table 1 shows that the STREAM benchmark [20] using only few threads on a non-loaded machine achieves best performance when spreading its pages across all memory nodes and keeping all threads together on a single processor. Indeed, distributing the pages aggregates the memory throughput of each NUMA node while keeping threads together lets the application benefit from shared caches.

However, on a loaded machine, having multiple threads access all memory nodes dramatically increases contention on memory links. The best performance in this case requires to avoid contention by carefully placing threads and data buffers so as to maximize the amount of local accesses (third row of Table 1). This suggests that achieving high-performance on NUMA architecture requires more than just binding tasks and data according to their affinities. Host load and memory contention must also be involved.

2.2 Software Support for Memory Management

While the memory architecture complexity is increasing, the virtual memory model is slowly being extended to help applications achieving better performance. Applications still manipulate virtual memory regions that are mapped to physical pages that the system allocates

anywhere on the machine. Most modern operating systems actually rely on a lazy allocation: when applications allocate virtual memory, the underlying physical pages are actually allocated upon the first access. While the primary advantage of this strategy is to decrease resource consumption, it brings an interesting feature usually referred to as *first-touch*: each page is allocated in the context of the thread that actually uses it first. The operating system is thus able to allocate physical pages on the memory node attached to the core that made the first access.

However, if the first thread touching a page is not the one that will eventually access it the most intensively, the page may not be allocated "in the right place". This situation actually often occurs since developers tend to prepare data buffers during an initialization phase while the actual computing threads were not launched yet. For this reason, some applications manually touch pages during the initialization phase to ensure that they are allocated on the right NUMA node, that is close to the computing threads that will actually access them later.

However, task/data affinities may change during execution, causing the optimal distribution to evolve dynamically. This is typically the case with dynamic application such as adaptive mesh refinement methods. Even if pages are carefully allocated during the initialization phase, their location is no longer optimal during the following steps. One solution consists in constantly migrating pages between memory nodes to move data near the tasks that access them. However, it is very expensive and it requires to detect at runtime when a memory region is no longer located appropriately. Another solution called *next-touch* is the generalization of the *first-touch* approach. It allows applications to ask the system to allocate or migrate a page near the thread that will perform the next access [19,23,26]. The *next-touch* policy thus can be used to redistribute data buffers to their new best locations between application steps. Unfortunately, this policy is hard to implement efficiently. Moreover, it does not solve situations where two threads are accessing the same memory region.

Actually, predicting performance is difficult because that memory access time is also related to the machine load. Irregular applications will thus not only cause load-imbalance between cores, they will also make the memory constraints vary dynamically, causing heuristics to become even harder to define.

3 Towards a Dynamic Approach to Place Threads and Memory

To tackle the problem of improving the overall application execution time over NUMA architectures, our approach is based on a flexible multi-level scheduling that continuously uses information about thread and data affinities. We present in this section our objectives and motivations and we describe our topology-aware memory manager that helps the FOREST-GOMP runtime system to implement our ideas.

3.1 Objectives and Motivations

Our aim is to perform thread and memory placement dynamically according to some scheduling hints provided by the application programmers, the compiler and even hardware counters. The idea is to map the parallel structure of the program onto the hardware architecture. This approach enables support for multiple strategies:

- At the machine level, the workload and memory load can be spread across NUMA nodes in order to favor locality.

- All threads working on the same buffers may be kept together within the same NUMA node to reduce memory contention.
- At the processor level, threads that share data intensively may also be grouped to improve cache usage and synchronization [10].
- Finally, inside multicore/multithreaded chips, access to independent resources such as computing units or caches may be taken into account. It offers the ability for a memory-intensive thread to run next to a CPU-intensive one without interference.

For irregular applications, all these decisions can only be taken at runtime. It requires an in-depth knowledge of the underlying architecture (memory nodes, multicore processors, shared caches, *etc.*) since both the application structure and the hardware characteristics are the key to high quality decisions.

Our idea consists in using distinct scheduling policies at the multiple topology levels of the machine. For instance, low-level work stealing only applies to neighboring cores so as to maintain data locality with regards to shared caches. At the memory node level, the thread scheduler deals with larger entities (e.g. multiple threads together with their data buffers) and may migrate them as a whole. Such a migration has to be decided at runtime after checking the hardware and application statuses. It requires that the runtime system maintains, during the whole execution, information about threads that belong to the same team and that frequently access some memory regions. Such affinity information can be quantified by the application, and later be refined at run time using hardware counters, for instance by looking at the evolution of cache miss rate before deciding whether a redistribution is needed.

In our model, scheduling actions can be triggered when the following events occur:

- a resources (i.e. thread or memory region) gets allocated/deallocated;
- a processor becomes idle;
- a hardware counter suddenly varies dramatically or exceeds a threshold (cache miss, remote access rate)

The scheduler can also be directly invoked by the application. Typically, a compiler could insert such calls when scheduling directives are encountered in the original source code.

To evaluate the relevance of our approach, we have developed a proof-of-concept OpenMP extension based on instrumentation of the application. We now give a brief overview of our implementation.

3.2 BUBBLESCHED, a Hierarchical Bubble-based Thread Scheduler

Scheduling threads on modern hierarchical architectures with multiple cores, shared-caches and NUMA nodes first requires a precise knowledge of this actual hardware hierarchy. To this end, we use the HWLOC library [1] to perform this topology discovery. It builds a hierarchical architecture tree composed of objects describing the hardware (NUMA nodes, sockets, caches, cores, and more) and various attributes such as the cache type and size, or the socket number (see Figure 1). It provides a portable programming interface that abstracts the machine hierarchy, offering both hardware information gathering and process and thread binding facilities. It also tries to leverage this knowledge through a high-level conceptual interface. HWLOC was initially developed for hierarchical thread scheduling [27], but the emergence of hierarchical multicore and NUMA architectures in clusters lead us to externalize it as a standalone library for generic task placement in high-performance computing. It is now used by both the MARCEL threading library and some MPI implementations such MPICH2 to bind threads and processes according to hardware affinities [9].

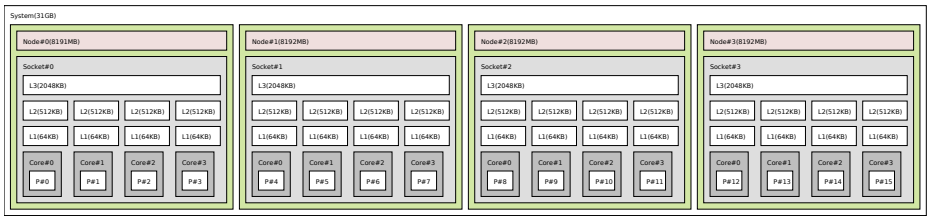


Fig. 1 Graphical view of HWLOC topology discovery on our quad-socket quad-core OPTERON machine.

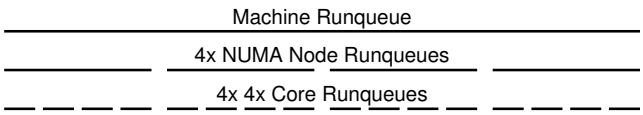


Fig. 2 Hierarchy of runqueues built by BUBBLESCHED on top of the same machine.

The MARCEL library implements high-performance user-level multithreading [27]. It is able to dynamically migrate threads across cores in less than 2.5 μs. On top of MARCEL, the BUBBLESCHED framework implements high level abstractions for developing powerful scheduling policies. By using HWLOC, BUBBLESCHED builds a hierarchy of *Runqueues* as depicted on Figure 2. Depending on whether a thread should be executed by any core in the machine, any core within a specific NUMA node, or a specific core, the scheduler may dynamically place it on the corresponding runqueue. Moreover, threads may be organized as entities called *Bubbles* so as to expose affinities to the scheduler. For instance, threads sharing data or synchronizing often are grouped in a bubble so that the scheduler keeps them together on the machine. In the end, BUBBLESCHED is responsible for scheduling a hierarchy of bubbles and threads over a hierarchy of hardware resources [27].

The BUBBLESCHED platform also provides a programming interface for developing new bubble schedulers. We developed the *Cache* bubble scheduler [10] whose main goal is to benefit from a good cache memory usage by scheduling teammate threads as close as possible and stealing threads from the most local cores when a processor becomes idle. This approach may cause some performance degradation in presence of memory intensive kernels or concurrent accesses because of cache pollution and contention. However it is interesting for cache-oblivious kernels which do not suffer from such issues while they benefit from locality. The *Cache* scheduler also keeps track of where the threads were being executed when it comes to perform a new thread and bubble distribution so as to improve locality during the whole execution.

3.3 MAMI, a NUMA-aware Memory Manager

While BUBBLESCHED manages threads over hierarchical architectures, it does not take care of data buffers. Managing memory buffers with NUMA awareness requires to know how many NUMA nodes the memory is physically split into, which processors are close to them, and their size. Again, thanks to HWLOC discovering the hardware characteristics, our MAMI library (*Marcel Memory Interface* [2]) gathers a deep knowledge of the memory architecture. Aside from usual memory allocation policies such as binding or interleaving, MAMI also offers two memory migration strategies. The first method is synchronous and

allows to move data on a given node on application's demand. The second method is based on a *next-touch* policy whose implementation is described in Section 3.4.

- void ***mami_malloc**(memory_manager, size);
  Allocate memory with the default policy.
- int **mami_register**(memory_manager, buffer, size);
  Register a memory area which has not been allocated by MAMI.
- int **mami_attach**(memory_manager, buffer, size, owner);
  Attach the memory to the specified thread.
- int **mami_migrate_on_next_touch**(memory_manager, buffer);
  Mark the area to be migrated when next touched.
- int **mami_migrate_on_node**(memory_manager, buffer, node);
  Move the area to the specified node.

Table 2 Application programming interface of MAMI.

MAMI also provides the application with hints about the actual cost of reading, writing, or migrating distant memory buffers. Moreover, MAMI gathers statistics about how much free memory is available on each node. It also remembers how much memory was allocated per thread. This information is potentially helpful when deciding whether or not to migrate a memory area or a thread so as to maintain both memory access locality and load balancing. Table 2 shows the main functionalities provided by MAMI. We will detail in Section 4 how FORESTGOMP relies on these features to implement its memory affinity directives.

3.4 Advanced Support for Memory Migration

Although LINUX earned NUMA-awareness in the last decades, its ability to manage NUMA memory is still limited to controlled allocation and static migration. As explained earlier, the need to migrate memory buffers dynamically raises the need for a *next-touch* policy.

One way to implement the *next-touch* policy in user-space consists in having the operating system generate a *Segmentation Fault* event on *next-touch* and letting a user-space library catching the corresponding signal in user-space. We implemented this model thanks to the `mprotect` primitive enabling *fake* segmentation faults on valid areas. If the fault occurs in a registered memory area, the signal handler retrieves the current location of the thread and target buffers from MARCEL and MAMI, migrates the corresponding pages near the thread, and restores the initial protection.

However, previous studies of this idea [26] revealed poor performance. Even if MARCEL and MAMI bring interesting knowledge of the current thread and data locations at runtime, large overheads are implied by the additional return to user-space (to run the signal handler before re-entering the kernel again for migration) and by the TLB flush on every processor during each `mprotect` (while another flush is already involved during page migration). On the other hand, SOLARIS has been offering an optimized kernel based *next-touch* implementation for a while and it is known to help applications significantly [19,23,26]. However, LINUX does not offer such a feature although it has spread to most high-performance computing sites nowadays. We thus propose a LINUX kernel-based *next-touch* strategy that migrates pages within the page-fault handler as described in Figure 3.

Our implementation is inspired by the *Copy-on-write* implementation in LINUX. The application marks pages as *Migrate-on-next-touch* using a new `madvise` parameter. The

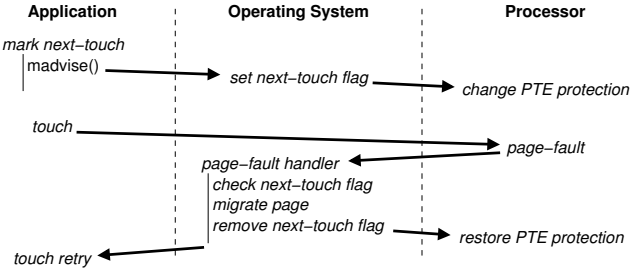


Fig. 3 Implementation of the *next-touch* policy in the LINUX kernel using `madvise` and a dedicated flag in the page-table entry (PTE).

LINUX kernel removes read/write flags from the page-table entries (PTEs) so that the next access causes a fault. When the fault occurs, the page-fault handler checks whether the page has been marked as *Migrate-on-next-touch*. If so, it allocates a new page, copies the data and frees the old one. This implementation enables *next-touch* migration as the new page is allocated on the NUMA node near the current thread by default.

Both user-space and kernel implementations of *next-touch* actually have different semantics. The kernel one is page-based: even if the application touches many pages successively, each of them is migrated individually. The user-space implementation manipulates larger or more complex areas: the library offering the method can obtain from the application the description of the whole memory area (for instance a matrix column) and migrate it entirely as soon as a single page is touched. These different semantics are expected to make the kernel implementation usable for small granularities while the user-space high overhead makes it more suitable for very large granularities. Moreover, as the user-space migration library knows the location of each page after the *next-touch* has occurred, it does not have to query the kernel again for page location. This additional knowledge could enable some optimization for complex migration patterns where multiple migrations are involved.

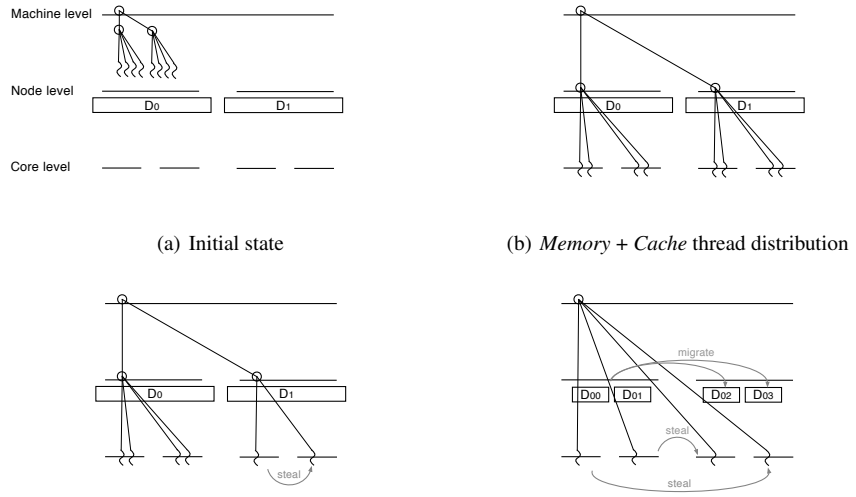
Our current experimentation platforms reveal that the kernel *next-touch* implementation is always faster than the user-space one [17]. However, since the former is only available in modified LINUX kernel, MAMI relies on both strategies to provide FORESTGOMP with efficient *next-touch* migration of data buffers in any case.

4 FORESTGOMP, a MAMI-Aware OpenMP Runtime

FORESTGOMP is an extension to the GNU OpenMP runtime system relying on the MARCEL/BUBBLESCHED user-level thread library. It benefits from advanced multithreading abilities so as to offer control on the way OpenMP threads are scheduled. FORESTGOMP automatically generates groups of threads (i.e. MARCEL *Bubbles*) out of OpenMP parallel regions to keep track of teammate threads relations *in a naturally continuous way* [10]. The FORESTGOMP platform has been enhanced to deal with memory affinities on NUMA architectures. We now detail how FORESTGOMP decides how to place these bubbles and their associated data thanks to BUBBLESCHED and MAMI.

4.1 A Scheduling Policy Guided by Memory Hints

We initially designed the *Cache* bubble scheduler to tackle dynamic cache-oblivious applications [10]. While bringing interesting results in this class of applications, the *Cache* scheduler does not take into account memory affinities, suffering from the lack of information about the data accessed by threads. Indeed, whereas keeping track of the bubble scheduler last distribution to move threads on the same core is not an issue, the BUBBLESCHED library needs feedback from the memory allocation library to be able to draw threads and bubbles to their *"preferred"* NUMA node. This is why we designed the *Memory* bubble scheduler that relies on the MAMI memory library to distribute threads and bubbles over the NUMA nodes regarding their memory affinities. This scheduler contains two main algorithms: the distribution algorithm that performs an initial threads and data distribution and the work-stealing algorithm which steals threads and migrates the corresponding data when one or several cores become idle.



(a) Initial state

(b) *Memory* + *Cache* thread distribution

(c) *Cache* new distribution after core 3 idleness

(d) *Memory* new distribution after cores 2-3 idleness

Fig. 4 Threads and data scheduling computed by the combination of the *Memory* and *Cache* bubble schedulers on an OpenMP application generating 2 teams of 4 threads on a computer made of 2 nodes of 2 cores each.

4.1.1 Distributing Threads and Data Accordingly

The distribution algorithm that comes with the *Memory* scheduler is called anytime the OpenMP application goes parallel. It operates recursively from the machine level to the NUMA node levels of the topology. The idea here is to have MAMI attaching *"memory hints"* to the threads thanks to BUBBLESCHED statistics interface. These hints describe which data regions will be accessed by existing threads and how memory intensive these accesses will be. The MAMI library then dynamically infers the location of the attached data, and summarizes this information on bubbles. This way, FORESTGOMP has all the

information it needs to perform a sound distribution of threads and data. The ultimate goal of *Memory* is to make every thread access local memory. It relies on the information given by MAMI to guide the thread distribution onto the correct NUMA nodes and migrate the attached data when necessary. Even if the distribution algorithm just draws the threads to the location of their data most of the time, a prior data distribution is sometimes needed to avoid memory contention or to prevent a NUMA node from being full. Once the *Memory* scheduler has distributed the threads and the attached data over the NUMA nodes of the machine, the *Cache* bubble scheduler is called inside each node to perform a cache-aware distribution over the cores.
We illustrate the *Memory* distribution algorithm by running an OpenMP application involving two teams of four threads on a computer made of two NUMA nodes of two cores each. The initial data distribution is shown on figure 4(a). *D0* and *D1* represent the data respectively accessed by team *0* and team *1*, and each array is allocated on a different NUMA node as usual for memory throughput reasons. The *Memory* scheduler first draws each team to the node that holds the array they access. Then, the *Cache* scheduler is called to distribute the threads inside each node. Figure 4(b) illustrates the resulting distribution.

4.1.2 Reacting upon core idleness

Each bubble scheduler can provide its own work-stealing algorithm called when a core becomes idle. The algorithm used by the *Cache* scheduler tries to steal threads from the most local cores. Its research scope is limited to the NUMA node of the idle core. The *Memory* scheduler work-stealing algorithm is called when the *Cache* scheduler does not manage to steal any thread inside the current NUMA node. Its main goal is to steal threads from remote nodes and, when appropriate, to migrate the associated memory. The selection of threads to steal is done by browsing the architecture topology from the most local nodes onwards. As migrating memory is an expensive mechanism, *Memory* tries to migrate as less data as possible. To do so, teams of threads with the fewest amount of attached data are chosen first. Threads with untouched memory are the best candidates in this context, as stealing them will not trigger any memory migration. The algorithm also takes the teams workload into account, to avoid stealing threads that will terminate soon. This workload information can be updated from the application using the BUBBLESCHED programming interface.
Figure 4(c) shows how FORESTGOMP reacts the idleness of core #3. The *Cache* scheduler work-stealing algorithm is called first to steal threads inside the current NUMA node. This algorithm picks a thread from core #2 to occupy core #3. This way, the runtime system does not need to migrate memory, as the stolen thread still accesses local data. When dealing with greater imbalance, *Cache* sometimes cannot find anything to steal from the current node. The *Memory* work-stealing algorithm is so called to steal threads from a different node. Figure 4(d) illustrates this behavior. Both cores #2 and #3 were idle, so the *Memory* scheduler had to pick two threads from node 0 to occupy the idle cores. As we steal threads from remote nodes, *Memory* also migrate the data accessed by the stolen threads on next touch. *DOi* represents the chunk of *D0* the *i*-th thread accesses.

4.2 Extending FORESTGOMP to Manage Memory

The FORESTGOMP platform has also been extended to offer application programmers a new set of functions to help convey memory-related information to the underlying OpenMP runtime. There are two main ways to update this information. Application programmers

- void **fgomp_malloc**(length);
  Allocate a buffer and attach it to the current thread.
- int **fgomp_set_current_thread_affinity**(buffer, length, shake_mode);
  Attach the given buffer to the current thread, and tell the scheduler whether the thread distribution should be recomputed accordingly.
- int **fgomp_set_next_team_affinity**(buffer, chunk_length, shake_mode);
  Attach one chunk of the given size of the buffer to each thread of the next parallel section.
- int **fgomp_attach_on_next_touch**(buffer, length);
  Attach the given memory buffer to the next thread accessing it.
- int **fgomp_migrate_on_next_touch**(buffer, length);
  Migrate the given memory buffer near the next thread accessing it.

Table 3 The FORESTGOMP interface for managing memory.

can express memory affinities by the time a new parallel region is encountered. This allows the FORESTGOMP runtime to perform early optimizations, like creating the corresponding threads at the right location. Updating memory hints inside a parallel region is also possible. Based on these new hints, the bubble scheduler may decide to redistribute threads. Applications can specify if this has to be done each time the updating function is called, or if the runtime has to wait until all the threads of the current team have reached the updating call. The FORESTGOMP runtime only moves threads if the new per-thread memory information negates the current distribution.

5 Performance Evaluation

We first describe in this section our experimentation platform and we detail the performance improvements brought by FORESTGOMP on increasingly complex applications.

5.1 Experimentation Platform

The experimentation platform is a quad-socket quad-core 1.9 GHz OPTERON 8347HE processor host depicted on Figure 5. Each processor contains a 2MB shared L3 cache and has 8 GB memory attached. The corresponding HWLOC discovery and BUBBLESCHED runqueue hierarchy are depicted by Figures 1 and 2 respectively.

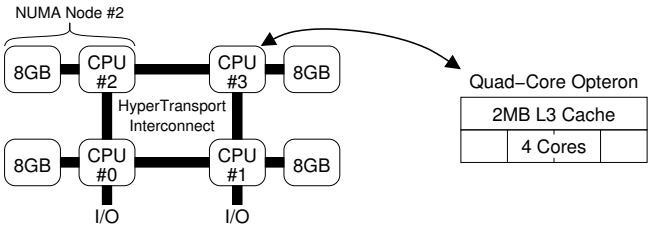


Fig. 5 The experimentation host is composed of 4 quad-core OPTERON (4 NUMA nodes).

Access type	Local access	Neighbor-node access	Opposite-node access
Read	83 ns	98 ns (× 1.18)	117 ns (× 1.41)
Write	142 ns	177 ns (× 1.25)	208 ns (× 1.46)

Table 4 Memory access latency (uncached) depending on the data being local or remote.

Table 4 presents the NUMA latencies on this host. Low-level remote memory accesses are indeed much slower when the distance increases. The base latency and the NUMA factor are higher for write accesses due to more hardware traffic being involved. The observed NUMA factor may then decrease if the application accesses the same cache line again as the remote memory node is not involved synchronously anymore. For a write access, the hardware may update the remote memory bank in the background (*Write-Back Caching*). Therefore, the NUMA factor depends on the application access patterns (for instance their spatial and temporal locality), and the way it lets the cache perform background updates.

5.2 STREAM

STREAM [20] is a synthetic benchmark developed in C, parallelized using OpenMP, that measures sustainable memory bandwidth and the corresponding computation rate for simple vectors. The 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.

Table 5 shows the results obtained by both GCC 4.2 LIBGOMP and FORESTGOMP runtimes running the STREAM benchmark. The LIBGOMP library exhibits varying performance (up to 20%), which can be explained by the fact the underlying kernel thread library does not bind the working threads on the computer cores. Two threads can be preempted at the same time, and switch their locations, inverting the original memory distribution. The FORESTGOMP runtime achieves a very stable rate. Indeed, without any memory information, the *Cache* bubble scheduler deals with the thread distribution, binding them to the cores. This way, the *first-touch* allocation policy is valid during the whole application run.

	LIBGOMP		FORESTGOMP	
Operation	Worst-Best	Average	Worst-Best	Average
Copy	8 504-12 056	10 646	14 200-14 368	14 299
Scale	8 469-11 953	10 619	14 239-14 391	14 326
Add	9 203-12 431	11 057	14 588-14 757	14 677
Triad	9 248-12 459	11 071	14 591-14 753	14 681

Table 5 STREAM benchmark results, in MB/s.

5.3 Nested-STREAM

To study further the impact of thread and data placement on the overall application performance, we modified the STREAM benchmark program to use nested OpenMP parallel regions. The application now creates one team per NUMA node of the computer. Each team

works on its own set of STREAM vectors, that are initialized in parallel, as in the original version of STREAM. To fit our target computer architecture, the application creates four teams of four threads. Table 6 shows the results obtained by both the LIBGOMP and the FORESTGOMP library.

The LIBGOMP runtime system maintains a pool of threads for non-nested parallel regions. New threads are created each time the application reaches a nested parallel region, and destroyed upon work completion. These threads can be executed by any core of the computer, and not necessarily where the master thread of the team is located. This explains why the results show a large deviation.

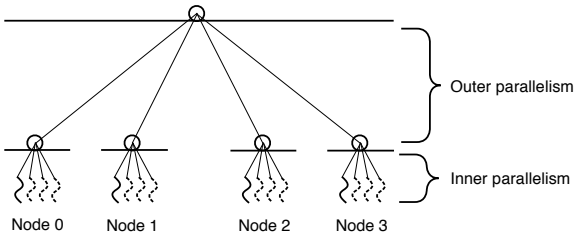


Fig. 6 Nested-STREAM OpenMP threads distribution by the FORESTGOMP runtime. Plain line threads were created by the outer parallel region and dashed line threads by the inner ones.

The FORESTGOMP runtime behaves better on this kind of application. 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, the FORESTGOMP library creates the teammates threads where the master thread of the team is currently located (see Figure 6). 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.

	LIBGOMP		FORESTGOMP	
Operation	Worst-Best	Average	Worst-Best	Average
Copy	1 555-1 983	1 788	3 606-3 626	3 615
Scale	1 614-2 024	1 814	3 599-3 621	3 613
Add	1 672-2 137	1 937	3 708-3 730	3 722
Triad	1 509-2 169	1 886	3 710-3 732	3 723

Table 6 Nested-STREAM benchmark results in MB/s, per 4-thread team.

5.4 Twisted-STREAM

To complicate the STREAM memory access pattern, we designed the Twisted-STREAM benchmark application, which contains two distinct phases. The first one behaves exactly as Nested-STREAM, except we only run the Triad kernel here, as it is the only one to involve the three vectors. During the second phase, each team works on a different data set than the

one it was given in the first phase. The *first-touch* allocation policy only gives good results for the first phase as shown in Table 7.

A typical solution to this lack of performance seems to rely on a *next-touch* page migration between the two phases of the application. However this functionality is not always available. And we show in the remaining of this section that the *next-touch* policy is not always the best answer to the memory locality problem.

	LIBGOMP	FORESTGOMP
Triad Phase 1	8 144 MB/s	9 108 MB/s
Triad Phase 2	3 560 MB/s	6 008 MB/s

Table 7 Average rates (per 4-thread team) observed with the Twisted-STREAM benchmark using a *first-touch* allocation policy. During phase 2, threads access data on a different NUMA node.

The STREAM benchmark program works on three 160MB-vectors. We experimented with two different data bindings for the second phase of Twisted-STREAM. In the first one, all three vectors are accessed remotely, while in the second one, only two of them are located on a remote node. We instrumented both versions with calls to the FORESTGOMP API to express which data are used in the second phase of the computation.

5.4.1 Remote Data

The underlying runtime system has two main options to deal with remote accesses. It can first decide to migrate the three vectors to the NUMA node hosting the accessing threads. It can also decide to move the threads to the location of the remote vectors. Figure 7 shows the results obtained for both cases.

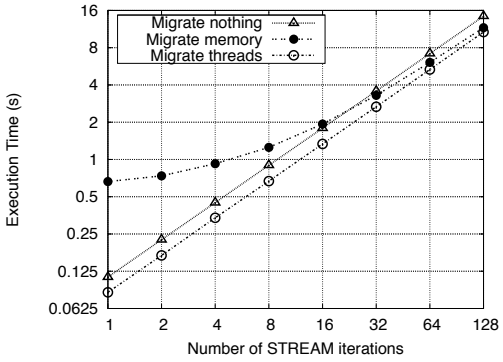


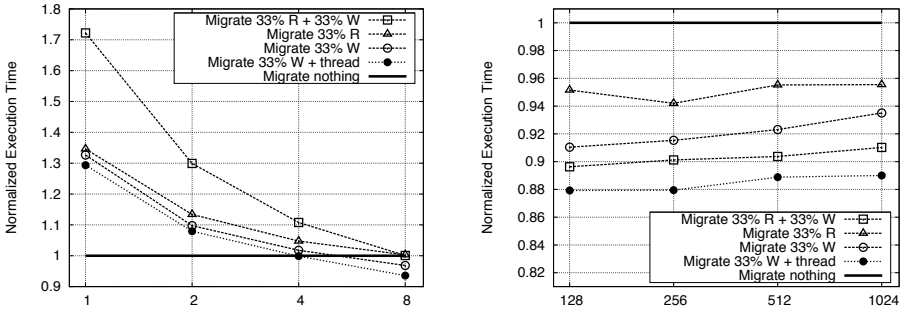
Fig. 7 Execution times of different thread and memory policies on the Twisted-STREAM benchmark, where the whole set of vectors is remotely located.

Moving the threads is definitely the best solution here. Migrating 16 threads is faster than migrating the corresponding vectors, and guarantees that every team only accesses local memory. On the other hand, if the thread workload becomes big enough, the cost for migrating memory may become lower than the cost for accessing remote data.

5.4.2 Mixed Local and Remote Data

For this case, only two of the three STREAM vectors are located on a remote NUMA node. One of them is read, while the other one is written. We first study the impact of the NUMA factor by only migrating one of the two remote vectors. Figure 8(a) shows the obtained performance. As mentioned in Table 4, remote read accesses are cheaper than remote write accesses on the target computer. Thus, migrating the read vector is less critical, which explains our better results when migrating the written vector. The actual performance difference between migrating read and written vectors is due to twice as many low-level memory accesses being required in the latter case.

To obtain a better thread and memory distribution, the underlying runtime can still migrate both remote vectors. Moving only the threads would not discard the remote accesses as all three vectors are not on the same node. That is why we propose a mixed approach in which the FORESTGOMP runtime system migrates both thread and local vector near to the other vector. This way, since migrating threads is cheap, we achieve a distribution where all the teams access their data locally while migrating as few data as possible. Figure 8(a) shows the overhead of this approach is smaller than the *next-touch* policy, for which twice as much data is migrated, while behaving the best when the thread workloads increase, as we can see on Figure 8(b).



(a) First iterations (b) Further iterations

Fig. 8 Execution times of different thread and memory policies on the Twisted-STREAM benchmark, where only two of the three vectors are remotely located.

We also tested these three STREAM benchmark versions on the Intel compiler 11.0, which behaves better than FORESTGOMP on the original STREAM application (10 500 MB/s) due to compiler optimizations. Nevertheless, performance drops significantly on both Nested-STREAM, with an average rate of 7 764 MB/s, and Twisted-STREAM with a second step average rate of 5 488 MB/s, while the FORESTGOMP runtime obtains the best performance.

5.5 Imbalanced-STREAM

Even if the results obtained by FORESTGOMP on the Nested-STREAM and Twisted-STREAM benchmarks are promising, they only demonstrate how efficient the FORESTGOMP threads

and data distribution can be with applications exposing balanced parallelism. When it comes to irregular applications, the runtime system must dynamically react when a processor idles. The Imbalanced-STREAM benchmark, a modified version of the Nested-STREAM benchmark, will help us to illustrate this problem.

The Imbalanced-STREAM benchmark also generates one team of threads per NUMA node of the target computer but we assign twice as many threads per team than in the Nested-STREAM version. Indeed, each core ends up with scheduling two threads in this version, allowing idle cores to steal work from the most loaded ones. We also assign different workloads to these teams, corresponding to the number of STREAM iterations a thread will have to compute. As an example, assigning a workload of 10 to a team means that every thread of the team will compute 10 times what a single thread would have computed in the original STREAM benchmark. To study the impact of work imbalance, we assigned a workload of 15 to the first two teams, a workload of 30 to the third one, and a workload of 1 to the last one. This way, the last team will terminate earlier than the other ones, giving the runtime system the opportunity to perform work-stealing. As before, each team works on its own set of STREAM vectors initialized using the *first-touch* allocation policy. Table 8 shows the execution time of this benchmark without activating FORESTGOMP's work-stealing algorithm. The four teams are run in parallel, and the threads from team *i* are distributed over the cores contained inside node *i*. We can see that Team 3 ends its execution far earlier than the others, and so making a whole set of cores idle.

Execution time (s)	Average	Min	Max
Team 0	1.6595	1.6578	1.6613
Team 1	1.4879	1.4876	1.4881
Team 2	2.7150	2.7113	2.7170
Team 3	0.1336	0.1335	0.1338
Overall time	2.7151	2.7114	2.7171

Table 8 Per-team execution times of the Imbalanced-STREAM benchmark run with FORESTGOMP without activating the work-stealing algorithm.

When some cores are idle, the underlying runtime system can try to dynamically adapt the current thread distribution so as to occupy every core. As detailed in Section 4.1.2, each bubble scheduler that comes with FORESTGOMP provides its own work-stealing algorithm. We first experimented with a modified version of the *Cache* scheduler work-stealing algorithm, which tries to steal threads from the most local cores to maximize cache memory reuse, and do not limit its research scope to the cores composing the current NUMA node. In the Imbalanced-STREAM benchmark, the completion of Team 3 turns the four cores inside Node 3 idle. The *Cache* work-stealing algorithm picks up the eight threads belonging to Team 2, and currently running on Node 2, to distribute them over Nodes 2 and 3. This way, every core inside Nodes 2 and 3 gets one thread to execute, and the load balance problem is solved. We also tried to steal threads running on Nodes 0 and 1. The results we obtained are summarized in Table 9.

Even if stealing threads from Team 0 or from Team 1 appears to be the best solution here, we need to compare these results with the ones presented in Table 8. In fact, the *Cache* work-stealing algorithm obtains poor performance compared to leaving the cores in an idle state. Indeed, distributing the threads of a team over two different NUMA nodes results in generating more traffic on the memory bus. The stolen threads keep accessing data from

Execution time (s)	Steal from Team 0	Steal from Team 1	Steal from Team 2
Team 0	1.9755	1.5228	1.5252
Team 1	1.5226	1.9755	1.5306
Team 2	2.9716	2.9692	3.4574
Team 3	0.1334	0.1333	0.1329
Overall time	**2.9696**	**2.9693**	3.4575

Table 9 Per-team average execution times of the Imbalanced-STREAM benchmark run with FORESTGOMP stealing threads from one of the remaining teams.

their former location and the induced remote memory accesses will increase the contention on the memory bus. As an example, stealing threads from Team 0 actually slows down Team 2, even if the threads from Team 2 keep executing on the same set of cores during the whole application run. This shows memory affinities need to be taken into account when performing work-stealing.

Execution time (s)	Steal from Team 0	Steal from Team 1	Steal from Team 2
Team 0	1.4401	1.7564	1.7374
Team 1	1.7453	1.4285	1.7720
Team 2	2.7178	2.7155	2.2843
Team 3	0.1632	0.1627	0.1629
Overall time	2.7179	2.7156	**2.2844**

Table 10 Per-team average execution times of the Imbalanced-STREAM benchmark run with FORESTGOMP stealing threads from one of the remaining teams **and migrating the corresponding data**.

This is why we experimented the Imbalanced-STREAM benchmark with the *Memory* bubble scheduler NUMA-aware work-stealing algorithm. This algorithm was designed to even the memory load on the computer by migrating the accessed data when stealing some threads. The *Memory* scheduler steals half of a team, just like the *Cache* scheduler, but also migrates the data accessed by the stolen threads. This mechanism can be compared to migrating the data on next touch. However in this specific case, application programmers would not be able to decide when marking data to be migrated. Indeed, due to the irregular nature of this benchmark, it is very unlikely to predict which threads would terminate first, and which data would have to be migrated. However, by expressing memory affinities, application programmers give FORESTGOMP the ability to migrate the accessed data while stealing threads to occupy idle cores. The results we obtain using this technique are presented in Table 10. We can see that stealing threads from Team 2 is the best solution here to minimize the benchmark overall execution time. Indeed, by looking at Table 8, we can see that the threads belonging to Team 2 are the last to finish. Stealing them gives them access to more computing power thus shortening their execution time. The obtained results show that a work-stealing algorithm involving threads and data migration can improve the performance of irregular applications.

These results also exhibits that stealing the team with the highest workload helps with improving the overall performance. We also tried to trigger the work-stealing algorithm anytime a core idles, but the best solution remain the ones which start by stealing from Team 2. Indeed, this algorithm gives better results by taking the load information into account, and thus considering how much work the remaining threads still have to compute, before

deciding to migrate them or not. Application programmers are sometimes able to provide load information about parallel regions. In case of regular applications, the runtime system could also establish a load factor by analyzing statistics about the performance the OpenMP teams obtained during the previous loop iterations.

6 Related Work

The quality of thread scheduling has a strong impact on the overall application performance because of thread and data affinities. While thread schedulers are already able to place threads according to their memory affinities [25], load-balancing also requires to spread threads across all cores, and thus to redistribute data dynamically to match their needs. Indeed, achieving optimal performance has been long known to require careful placement of threads as close as possible to the data they access [8,5].

Many research projects have been carried out to improve data distribution and execution of OpenMP programs on NUMA architectures. This has been done either through HPF directives [7] or by enriching OpenMP with data distribution directives [12] directly inspired by HPF and the SGI Fortran compiler. 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.

Nikolopoulos *et al.* [21] designed a mechanism to migrate memory pages automatically that relies on user-level code instrumentation performing a sampling analysis of the first loop iterations of OpenMP applications to determine thread and memory affinity relations. They have shown their approach can even be more efficient when the page migration engine and the operating system scheduler [22] are able to communicate. This pioneering research only suits OpenMP applications that have a regular memory access pattern while our approach favors many more applications. Hardware counters may also be used to gather affinity information so as to offer placement hints for the next run [24]. FORESTGOMP only gathers affinity knowledge from bubbles that it creates from OpenMP parallel sections, but it is able to dynamically adapt thread and data placement at runtime without relying on a post-mortem analysis. Moreover FORESTGOMP is able to retrieve hardware counters at runtime and compare them to thresholds and look at their evolution so as to for instance decide when to redistribute in case of sudden memory bus contention or cache misses.

Most operating systems acquired some limited NUMA-aware capabilities within their memory management and thread schedulers. To tackle irregular algorithms, [19,23,26] have studied the promising *next-touch* policy. Their approach however suffers from the lack of cooperation between the allocation library and the thread scheduler, and from not mastering the underlying memory architecture constraints. Our runtime consists in a tight integration of the BUBBLESCHED and MAMI knowledge of the application state and of the hardware, which lets FORESTGOMP benefit from our *next-touch* implementation in the LINUX kernel.

In order to favor affinities in a portable manner the NANOS compiler [6] 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 [13] proposes a mechanism to accurately select the threads of a sub-team to define the thread-core mapping for better data locality, although this proposition does not involve nested parallelism. 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*.

7 Conclusion and Future Work

Exploiting the full computational power of current more and more hierarchical multiprocessor machines requires a very careful distribution of threads and data among the underlying non-uniform architecture. Directive-based programming languages provide programmers with a portable way to specify the parallel structure of their application. Using such information, the scheduler can take appropriate load balancing decisions and either choose to migrate memory, or to move threads across the architecture. Indeed, thread/memory affinity does matter mainly because of congestion issues in modern NUMA architectures.

Therefore, we introduce a multi-level thread scheduler combined with a NUMA-aware memory manager. It enables dynamic load distribution in a coherent way based on application requirements and hardware constraints, thus helping to reach performance portability. It also provides NUMA-aware work-stealing algorithms to tackle irregular applications. Our experiments show that mixed solutions (migrating threads and data) improve overall performance. Moreover, traditional *next-touch*-based data distribution approaches are not always optimal since they are not aware of the memory load of the target node. Migrating threads is more efficient in such situations.

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 will 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. Hardware counter feedback should also be involved in this process, as they should warn the runtime system about memory contention and high rates of remote accesses.

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 standard 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 and could also widen the OpenMP spectrum to hybrid programming [14, 15].

In the longer run, we plan to explore ways to compose our scheduling strategies with other schedulers and paradigms. For instance, parallel languages such as *Cilk* or *TBB* rely on runtime systems able to efficiently schedule fine-grain parallelism on SMP architectures. The idea is to let such fine-grain task schedulers run inside NUMA nodes, while using our *Memory* scheduler to limit inter-node remote memory accesses, thus widening the spectrum of flat parallelism approaches to NUMA computers in a portable way.

References

1. hwloc: Portable hardware locality. `http://runtime.bordeaux.inria.fr/hwloc/`
2. Mami: Marcel memory interface. `http://runtime.bordeaux.inria.fr/MaMI/`
3. The OpenMP API specification for parallel programming. `http://www.openmp.org/`
4. Thread Building Blocks. `http://www.intel.com/software/products/tbb/`
5. Antony, J., Janes, P.P., Rendell, A.P.: Exploring Thread and Memory Placement on NUMA Architectures: Solaris and Linux, UltraSPARC/FirePlane and Opteron/HyperTransport. In: Proceedings of the International Conference on High Performance C omputing (HiPC). Bangalore, India (2006)
6. Ayguade, E., Gonzalez, M., Martorell, X., Jost, G.: Employing Nested OpenMP for the Parallelization of Multi-Zone Computational Fluid Dynamics Applications. In: 18th International Parallel and Distributed Processing Symposium (IPDPS) (2004)

7. Benkner, S., Brandes, T.: Efficient parallel programming on scalable shared memory systems with High Performance Fortran. In: Concurrency: Practice and Experience, vol. 14, pp. 789–803. John Wiley & Sons (2002)
8. Brecht, T.: On the Importance of Parallel Application Placement in NUMA Multiprocessors. In: Proceedings of the Fourth Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV). San Diego, CA (1993)
9. Broquedis, F., Clet-Ortega, J., Moreaud, S., Furmento, N., Goglin, B., Mercier, G., Thibault, S., Namyst, R.: 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). IEEE Computer Society Press, Pisa, Italia (2010)
10. Broquedis, F., Diakhaté, F., Thibault, S., Aumage, O., Namyst, R., Wacrenier, P.A.: Scheduling Dynamic OpenMP Applications over Multicore Architectures. In: International Workshop on OpenMP (IWOMP). West Lafayette, IN (2008)
11. Carlson, W., Draper, J., Culler, D., Yelick, K., Brooks, E., Warren, K.: Introduction to UPC and Language Specification. Tech. Rep. CCS-TR-99-157, George Mason University (1999)
12. Chapman, B.M., Bregier, F., Patil, A., Prabhakar, A.: Achieving performance under OpenMP on cc-NUMA and software distributed shared memory systems. In: Concurrency: Practice and Experience, vol. 14, pp. 713–739. John Wiley & Sons (2002)
13. Chapman, B.M., Huang, L., Jin, H., Jost, G., de Supinski, B.R.: Extending openmp worksharing directives for multithreading. In: EuroPar'06 Parallel Processing (2006)
14. Dolbeau, R., Bihan, S., Bodin, F.: HMPP: A hybrid multi-core parallel programming environment (2007)
15. Duran, A., Perez, J.M., Ayguade, E., Badia, R., Labarta, J.: Extending the openmp tasking model to allow dependant tasks. In: IWOMP Proceedings (2008)
16. Frigo, M., Leiserson, C.E., Randall, K.H.: The Implementation of the Cilk-5 Multithreaded Language. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). Montreal, Canada (1998)
17. Goglin, B., Furmento, N.: Enabling High-Performance Memory-Migration in Linux for Multithreaded Applications. In: MTAAP'09: Workshop on Multithreaded Architectures and Applications, held in conjunction with IPDPS 2009. IEEE Computer Society Press, Rome, Italy (2009). DOI 10.1109/IPDPS.2009.5161101
18. Koelbel, C., Loveman, D., Schreiber, R., Steele, G., Zosel, M.: The High Performance Fortran Handbook (1994)
19. Löf, H., Holmgren, S.: affinity-on-next-touch: increasing the performance of an industrial PDE solver on a cc-NUMA system. In: 19th ACM International Conference on Supercomputing, pp. 387–392. Cambridge, MA, USA (2005)
20. McCalpin, J.D.: Memory bandwidth and machine balance in current high performance computers. IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter pp. 19–25 (1995)
21. Nikolopoulos, D.S., Papatheodorou, T.S., Polychronopoulos, C.D., Labarta, J., Ayguadé, E.: User-Level Dynamic Page Migration for Multiprogrammed Shared-Memory Multiprocessors. In: ICPP, pp. 95–103. IEEE (2000)
22. Nikolopoulos, D.S., Polychronopoulos, C.D., Papatheodorou, T.S., Labarta, J., Ayguadé, E.: Scheduler-Activated Dynamic Page Migration for Multiprogrammed DSM Multiprocessors. Parallel and Distributed Computing **62**, 1069–1103 (2002)
23. Nordén, M., Löf, H., Rantakokko, J., Holmgren, S.: Geographical Locality and Dynamic Data Migration for OpenMP Implementations of Adaptive PDE Solvers. In: Second International Workshop on OpenMP (IWOMP 2006). Reims, France (2006)
24. Song, F., Moore, S., Dongarra, J.: Feedback-Directed Thread Scheduling with Memory Considerations. In: Proceedings of the 16th IEEE International Symposium on High-Performance Distributed Computing (HPDC07). Monterey Bay, CA (2007)
25. Steckermeier, M., Bellosa, F.: Using Locality Information in Userlevel Scheduling. Tech. Rep. TR-95-14, University of Erlangen-Nürnberg – Computer Science Department – Operating Systems – IMMD IV, Martensstraße 1, 91058 Erlangen, Germany (1995)
26. Terboven, C., an Mey, D., Schmidl, D., Jin, H., Reichstein, T.: Data and thread affinity in openmp programs. In: MAW '08: Proceedings of the 2008 workshop on Memory access on future processors, pp. 377–384. ACM, New York, NY, USA (2008). DOI http://doi.acm.org/10.1145/1366219.1366222
27. Thibault, S., Namyst, R., Wacrenier, P.A.: Building Portable Thread Schedulers for Hierarchical Multiprocessors: the BubbleSched Framework. In: Euro-Par. ACM, Rennes, France (2007)
28. Yang, R., Antony, J., Janes, P.P., Rendell, A.P.: Memory and Thread Placement Effects as a Function of Cache Usage: A Study of the Gaussian Chemistry Code on the SunFire X4600 M2. In: Proceedings of the International Symposium on Parallel Architectures, Algorithms, and Networks (i-span 2008), pp. 31–36 (2008)