Dynamic Task and Data Placement over NUMA Architectures: an OpenMP Runtime Perspective

François Broquedis, Nathalie Furmento, Brice Goglin, Raymond Namyst,

Pierre-André Wacrenier

▶ To cite this version:

HAL Id: inria-00367570
https://hal.inria.fr/inria-00367570

Submitted on 11 Mar 2009

Dynamic Task and Data Placement over NUMA Architectures: an OpenMP Runtime Perspective

François Broquedis[1], Nathalie Furmento[3], Brice Goglin[2],
Raymond Namyst[1], Pierre-André Wacrenier[1]

[1] University of Bordeaux, [2] INRIA, [3] CNRS
LaBRI – 351 cours de la Libération
F-33405 TALENCE – FRANCE
[1,3] `lastname@labri.fr`   –[2] `firstname.lastname@inria.fr`

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 memory access penalties. Directive-based programming languages such as OpenMP provide 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" to solve thread/memory affinity issues. It enables dynamic load distribution guided by application structure and hardware topology, thus helping to achieve performance portability. First experiments show that mixed solutions (migrating threads and data) outperform *next-touch*-based data distribution policies and open possibilities for new 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*). Note that such machines are now becoming mainstream thanks to the spreading of AMD HYPERTRANSPORT and INTEL QUICKPATH technologies.

Running parallel applications efficiently on older multiprocessor machines was essentially a matter of careful task scheduling. In this context, parallel runtime systems such as Cilk [7] or TBB [9] have proved to be very effective. In fact, these approaches can still behave well over hierarchical multicore machines in the case of 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". 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 underlying runtime system.

Several programming approaches provide means to specify task-memory affinities within parallel applications (OpenMP, HPF [10], UPC [3]). 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 [17, 2] emphasized the importance of establishing a persistent cooperation between an OpenMP compiler and the underlying runtime system on multicore NUMA machines. We designed FORESTGOMP [17] that extends the GNU OpenMP implementation, GOMP, to make use of the BUBBLESCHED flexible scheduling framework [18]. Our approach has proved to be relevant for applications dealing with nested, massive parallelism.

We introduce in this paper a major extension of our OpenMP runtime system that connects the thread scheduler to a NUMA-aware memory management subsystem. This new runtime not only can use per-bubble memory allocation information when performing thread re-distributions, but it can also perform data migration — either immediately or upon *next-touch*— in situations when it is more appropriate. 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. We describe our extensions to the FORESTGOMP runtime system in Section 3. In Section 4, we evaluate the relevance of our proposal with several performance-oriented experiments. Before concluding, related work is summarized in Section 5.

2 Background and Motivations

In this section, we briefly introduce modern memory architectures and how they affect application performance. Then we detail how existing software techniques try to overcome these issues and show the difficulty is to be as less intrusive as possible while trying to achieve high performance.

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 remains several orders of magnitude slower. Concurrent accesses to central memory buses lead to dramatic contention, causing the overall performance to decrease. It led hardware designers to drop the centralized memory model in favor of distributed and hierarchical architectures, where memory nodes and caches are attached to some cores and far away from the others. This design has been widely used in high-end servers based on the ITANIUM processor. It now becomes mainstream since AMD HYPERTRANSPORT (see Figure 1) and

the upcoming INTEL QUICKPATH 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 local memory faster than other memory nodes. The 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. Not only does the application run faster if accessing local data, but also contention may appear on memory links if two processors access each others' memory nodes. Moreover, shared caches among cores increase the need to take data locality into account while scheduling tasks.

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-socket quad-core OPTERON machine depending on the machine load (4 or 16 threads) and the location of memory buffers.

To illustrate this problem, we ran some experiments on a quad-socket quad-core OPTERON machine. Second row of Table 1 shows that a custom application using few threads on a non-loaded machine will achieve best performance by distributing its pages across all memory nodes (to maximize the throughput) and keeping all threads together on a single processor (to benefit from a shared cache). However, on a loaded machine, having multiple threads access all memory nodes dramatically increases contention on memory links, thus achieving the best performance when each task only accesses local memory (third row of Table 1). This suggests that achieving high-performance on NUMA architecture takes more than just binding tasks and data based on their affinities. The host load and memory contention should also be taken into account.

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 buffers 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 near the core that accesses it.

However, if the first thread touching a page is not the one that will access it intensively in the future, the page may not be allocated "in the right place". For this reason, some applications manually touch pages during the initialization phase to ensure that they are allocated close to the threads that will actually access them.

Dynamic applications such as adaptive meshes have their task/data affinities varying during the execution, causing the optimal distribution to evolve. One solution consists in migrating pages between memory nodes to move data near the tasks that access them at any time. However, it is expensive and requires actually to detect at runtime that a buffer is not located on the right node. Another solution called *next-touch* is the generalization of the *first-touch* approach: it allows applications to ask the system to allocate or migrate each page near the thread that will first touch it in the future [11, 15, 16]. It is unfortunately hard to implement efficiently and also does not work well in many cases, for instance if two threads are actually touching the same zone.

These features enable memory-aware task and data placement but they remain expensive. Moreover, as illustrated by the above experiment, predicting performance is difficult given that memory performance is also related to the machine load. Irregular applications will thus not only cause the thread scheduler to have to load-balance between idle and busy cores, but will also make the memory constraints vary dynamically, causing heuristics to become even harder to define.

3   Design of 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.

3.1   Objectives

Our objective is to perform thread and memory placement dynamically according to some scheduling hints provided by the programmer, the compiler or 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 and locality may be favored. 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 [2]. 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, shared caches, *etc.*). Our idea consists in using separate scheduling policies at various topology levels of the machine. For instance, low-level work stealing only applies to neighboring

cores while the memory node level scheduler transfers larger entities (multiple threads with their data buffers) without modifying their internal scheduling. Such a transfer has to be decided at runtime after checking the hardware and application statuses. It requires that the runtime system remembers, during the whole execution, which threads are part of the same team and which memory buffers they often access. It should be possible to quantify these affinities as well as dynamically modify them if needed during the execution. To do so, affinity information may be attached at thread or buffer creation or later, either by the application programmer, by the compiler (through static analysis), or by the runtime system through instrumentation. In the end, the problem is to decide which actions have to be performed and when. We have identified the following events:

- When the application allocates or releases a resource (e.g., thread or buffer);
- When a processor becomes idle (blocking thread);
- When hardware counters reveal an issue (multiple accesses to remote nodes);
- When application programmers insert an explicit hint in their code.

To evaluate this model, we developed a proof-of-concept OpenMP extension based on instrumentation of the application. We now briefly present our implementation.

3.2 MAMI, a NUMA-Aware Memory Manager

MAMI is a memory interface implemented within our user-level thread library, MARCEL [18]. It allows developers to manage memory with regard to NUMA nodes thanks to an automatically gathered knowledge of the underlying architecture. The initialization stage preallocates memory heaps on all the NUMA nodes of the target machine, and user-made allocations then pick up memory areas from the preallocated heaps.

MAMI implements two methods to migrate data. The first method is based on the *next-touch* policy, it is implemented as a user-level pager (`mprotect()` and signal handler for `SIGSEGV`). The second migration method is synchronous and allows to move data on a given node. Both move pages using the LINUX system call `move_pages()`.

Migration cost is based on a linear function on the number of pages being migrated[1]. The cost in microseconds for our experimentation platform is $120+11 \times nbpages$ (about 380 MB/s). It is also possible to evaluate reading and writing access costs to remote memory areas. Moreover, MAMI gathers statistics on how much memory is available and left on the different nodes. This information is potentially helpful when deciding whether or not to migrate a memory area. Table 2 shows the main functionalities provided by MAMI.

3.3 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 MARCEL's efficient thread migration mechanism[2] thus offering control on the way OpenMP threads

---

[1] Our experiments first showed a non-linear behavior for the migration cost. It led us to improve the `move_pages()` system call implementation in LINUX kernel 2.6.29 to reduce the overhead when migrating many pages at once [8].

[2] The thread migration cost is about 0.06μs per thread and the latency is about 2.5μs.

- void *__mami_malloc__(memory_manager, size);
  Allocates memory with the default policy.
- int __mami_register__(memory_manager, buffer, size);
  Registers a memory area which has not been allocated by MAMI.
- int __mami_attach__(memory_manager, buffer, size, owner);
  Attaches the memory to the specified thread.
- int __mami_migrate_on_next_touch__(memory_manager, buffer);
  Marks the area to be migrated when next touched.
- int __mami_migrate_on_node__(memory_manager, buffer, node);
  Moves the area to the specified node.
- void __mami_cost_for_read_access__(memory_manager, source, dest, size, cost);
  Indicates the cost for read accessing SIZE bits from node SOURCE to node DEST.

Table 2. Application programming interface of MAMI.

are scheduled. FORESTGOMP also automatically generates groups of threads, called *bubbles*, out of OpenMP parallel regions to keep track of teammate threads relations *in a naturally continuous way*. Thanks to MARCEL automatically gathering a deep knowledge of hardware characteristics such as cores, shared caches, processors and NUMA nodes, BUBBLESCHED and MAMI are able to take interesting decisions when placing these bubbles and their associated data. The BUBBLESCHED library provides specific *bubble schedulers* to distribute these groups of threads over the computer cores. The BUBBLESCHED platform also maintains a programming interface for developing new bubble schedulers. For example, the *Cache* bubble scheduler [2] has been developed using this interface. Its main goal is to benefit from a good cache memory usage by scheduling teammate threads as close as possible on the computer and stealing threads from the most local cores when a processor becomes idle. It also keeps track of where the threads were being executed when it comes to perform a new thread and bubble distribution.

The FORESTGOMP platform has been enhanced to deal with memory affinities on NUMA architectures.

A Scheduling Policy Guided by Memory Hints. Even if the *Cache* bubble scheduler offers good results on dynamic cache-oblivious applications, it does not take into account memory affinities, suffering from the lack of information about the data the threads access. 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. That 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. The idea here is to have MAMI attaching *"memory hints"* to the threads by calling the BUBBLESCHED programming interface. These hints describe how much data a thread will use, and where the data is located. This way, the bubble scheduler is able to guide the thread distribution onto the

correct NUMA nodes. Then, the *Cache* bubble scheduler is called inside each node to perform a cache-aware distribution over the cores.

Extending FORESTGOMP to Manage Memory. The FORESTGOMP platform has also been extended to offer the application programmer 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 can express memory affinities by the time a new parallel region is encountered. This allows the FOREST-GOMP 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.

4   Performance Evaluation

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

4.1   Experimentation Platform

The experimention platform is a quad-socket quad-core 1.9 GHz OPTERON 8347HE processor host depicted on Figure 1. Each processor contains a 2MB shared L3 cache and has 8 GB memory attached.
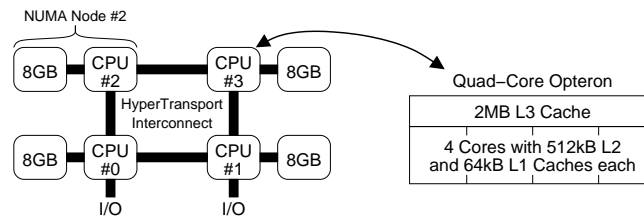

Fig. 1. The experimentation host is composed of 4 quad-core OPTERON (4 NUMA nodes).

Table 3 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

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

Table 3. Memory access latency (uncached) depending on the data being local or remote.

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.

4.2 STREAM

STREAM [12] 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 4 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	Worst-Best
Copy	6 747-8 577	7 851-7 859
Scale	6 662-8 566	7 821-7 828
Add	7 132-8 821	8 335-8 340
Triad	7 183-8 832	8 357-8 361

Table 4. STREAM benchmark bandwidth results on a 16-core machine (1 thread per core), in MB/s.

4.3 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 5 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.

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. 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 |
| | Worst-Best | Worst-Best |
Operation		
Copy	6 900-8 032	8 302-8 631
Scale	6 961-7 930	8 201-8 585
Add	7 231-8 181	8 344-8 881
Triad	7 275-8 123	8 504-9 217

Table 5. Nested-STREAM benchmark bandwidth results in MB/s.

4.4 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, because 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 6.

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 6. Average rates observed with the Triad kernel of 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 FOREST-GOMP API to express which data are used in the second phase of the computation.

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 2 shows the results obtained for both cases.
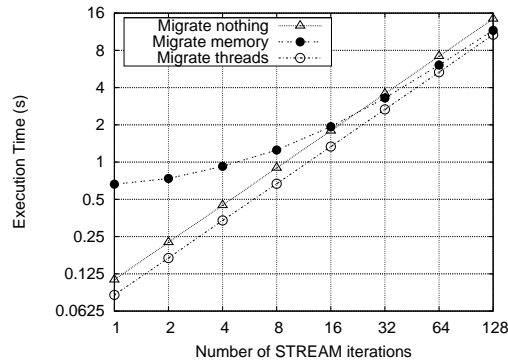

Fig. 2. Execution times of different thread and memory policies on the Twisted-STREAM benchmark, where the whole set of vectors is remotely located.

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 be become lower than the cost for accessing remote data.

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 3(a) shows the obtained performance. As mentioned in Table 3, 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 3(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 3(b).
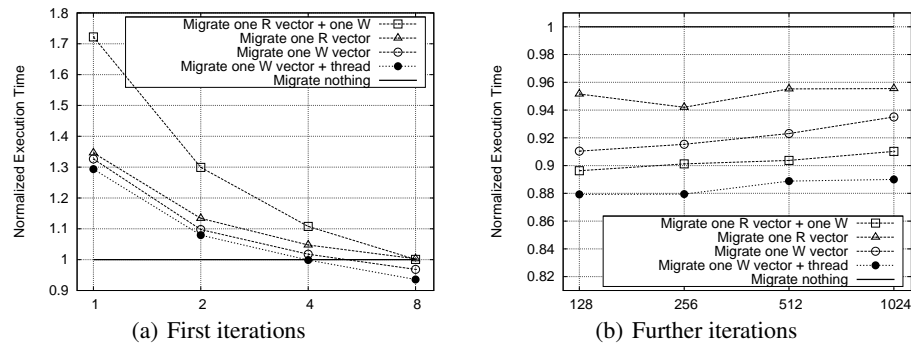

Fig. 3. Execution times of different thread and memory policies on the Twisted-STREAM benchmark, where only two of the three vectors are remotely located.

We also tested all our 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   Related Work

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 [1] or by enriching OpenMP with data distribution directives [4] 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.* [13] 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 be even more efficient when the page migration engine and the operating system scheduler [14] 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.

To tackle irregular algorithms, [11, 15, 16] have studied the promising *next-touch* policy. It allows the experienced programmer to ask for a new data distribution explicitly the next time data is touched. While being mostly similar to the easy-to-use *first-touch* memory placement policy in terms of programming effort, the *next-touch* policy suffers from the lack of cooperation between the allocation library and the thread scheduler. Moreover, this approach does not take the underlying architecture into account and so can hardly achieve most of its performance. FORESTGOMP works around this issues thanks to BUBBLESCHED and MAMI knowledge of the underlying processor and memory architecture and load.

6   Conclusion and Future Work

Exploiting the full computational power of current more and more hierarchical multi-processor 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. Through this information, the scheduler can take appropriate load balancing decisions and either choose to migrate memory, or decide 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. Our early 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 may be more efficient in such situations.

We plan first to enhance the current bubble framework so as to improve our scheduling decision criteria by introducing global redistribution phases at some times. Obviously, hardware counter feedback should also be involved in this process. We therefore need to experiment further with both synthetic and real-life applications.

These results also suggest there is a need to extend OpenMP so as to transmit task/memory affinity relations to the underlying runtime. This evolution could also widen the OpenMP spectrum to hybrid programming [5, 6].

References

1. Siegfried Benkner and Thomas Brandes. Efficient parallel programming on scalable shared memory systems with High Performance Fortran. In *Concurrency: Practice and Experience*, volume 14, pages 789–803. John Wiley & Sons, 2002.

2. François Broquedis, François Diakhaté, Samuel Thibault, Olivier Aumage, Raymond Namyst, and Pierre-André Wacrenier. Scheduling Dynamic OpenMP Applications over Multicore Architectures. In *International Workshop on OpenMP (IWOMP)*, West Lafayette, IN, May 2008.

3. William Carlson, Jesse Draper, David Culler, Kathy Yelick, Eugene Brooks, and Karen Warren. Introduction to UPC and Language Specification. Technical Report CCS-TR-99-157, George Mason University, May 1999.

4. Barbara M. Chapman, Fréderic Bregier, Amit Patil, and Achal Prabhakar. Achieving performance under OpenMP on ccNUMA and software distributed shared memory systems. In *Concurrency: Practice and Experience*, volume 14, pages 713–739. John Wiley & Sons, 2002.

5. Romain Dolbeau, Stéphane Bihan, and François Bodin. HMPP^{TM}: A Hybrid Multi-core Parallel Programming Environment. Technical report, CAPS entreprise, 2007.

6. Alex Duran, Josep M. Perez, Eduard Ayguade, Rosa Badia, and Jesus Labarta. Extending the OpenMP Tasking Model to Allow Dependant Tasks. In *International Workshop on OpenMP (IWOMP)*, West Lafayette, IN, May 2008.

7. Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Montreal, Canada, June 1998.

8. Brice Goglin and Nathalie Furmento. 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*, Rome, Italy, May 2009. IEEE Computer Society Press.

9. Intel. Thread Building Blocks. `http://www.intel.com/software/products/tbb/`   .

10. Charles Koelbel, David Loveman, Robert Schreiber, Guy Steele, and Mary Zosel. *The High Performance Fortran Handbook*, 1994.

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

12. John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

13. Dimitrios S. Nikolopoulos, Theodore S. Papatheodorou, Constantine D. Polychronopoulos, Jesús Labarta, and Eduard Ayguadé. User-Level Dynamic Page Migration for Multiprogrammed Shared-Memory Multiprocessors. In *International Conference on Parallel Processing*, pages 95–103. IEEE Computer Society Press, September 2000.

14. Dimitrios S. Nikolopoulos, Constantine D. Polychronopoulos, Theodore S. Papatheodorou, Jeús Labarta, and Eduard Ayguadé. Scheduler-Activated Dynamic Page Migration for Multiprogrammed DSM Multiprocessors. *Parallel and Distributed Computing*, 62:1069–1103, December 2002.

15. Markus Nordén, Henrik Löf, Jarmo Rantakokko, and Sverker Holmgren. Geographical Locality and Dynamic Data Migration for OpenMP Implementations of Adaptive PDE Solvers. In *Second International Workshop on OpenMP (IWOMP)*, Reims, France, 2006.

16. Christian Terboven, Dieter an Mey, Dirk Schmidl, Henry Jin, and Thomas Reichstein. Data and Thread Affinity in OpenMP Programs. In *MAW '08: Proceedings of the 2008 workshop on Memory access on future processors*, pages 377–384, New York, NY, USA, 2008. ACM.

17. Samuel Thibault, François Broquedis, Brice Goglin, Raymond Namyst, and Pierre-André Wacrenier. An Efficient OpenMP Runtime System for Hierarchical Architectures. In *International Workshop on OpenMP (IWOMP)*, pages 148–159, Beijing, China, June 2007.

18. Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Building Portable Thread Schedulers for Hierarchical Multiprocessors: the BubbleSched Framework. In *European Conference on Parallel Computing (EuroPar)*, Rennes, France, August 2007. ACM.