A runtime system architecture for ubiquitous support of OpenMP

Giorgos Ch. Philos   Vassilios V. Dimakopoulos   Panagiotis E. Hadjidoukas
University of Ioannina, Ioannina, Greece
{gfilos,dimako,phadjido}@cs.uoi.gr

Abstract

In this work we present the runtime architecture of the OMPi OpenMP compiler. OMPi is a source-to-source C translator featuring a portable, modular and extensible runtime system. It allows for OpenMP threads to map to different execution entities which range from kernel/user-level threads to processes, providing transparent support of OpenMP applications on both SMP machines and clusters of SMPs. When operating within an SMP machine, arbitrary threading libraries can be employed; currently a multitude of such libraries is available, including one which is based on portable user-level threading, for high-performance nested parallelism support. When operating on a cluster, processes are used as the execution entities and different software DSM cores can be utilized under a unified interface; the runtime system uses a hybrid approach whereby its internal bookkeeping is done through explicit message passing, while user-program shared variables are handled by the DSM core.

1 Introduction

OpenMP [22] has become a standard paradigm for shared memory programming, as it offers the advantage of simple and incremental parallel program development, in a high abstraction level. Its usage is continuously increasing as small SMP machines have become the mainstream architecture even in the personal computer market, thanks to the domination of multicore CPUs.

At the same time, computational clusters have emerged as a cost-effective approach to high performance computing. Individual machines unified by a LAN, either using a commodity or a high-performance interconnect, can be viewed as a virtual large-scale machine with a big number of processors and can be programmed as such. They offer an expandable and reliable computational environment which is quite more economic than large massively parallel machines. However, programming for a cluster is rather cumbersome. The most widely used and arguably most efficient tool for cluster programming is the MPI library. MPI though forces the programmer to explicitly distribute data and orchestrate communications by hand, and as a result it has not find its way to mainstream computing.

An alternative to MPI is the use of software DSM (sDSM) libraries that give the illusion of shared memory, simplifying program development. One downside of software DSM libraries is that they usually employ relaxed consistency protocols, which burdens a programmer with inserting synchronization calls in order to make sure that the program executes correctly. Although software DSM systems don't seem to be able to achieve the speedups possible with carefully hand-coded MPI programs, they have nevertheless been proved successful for a number of data-intensive applications [9, 4]. A serious problem with software DSM systems is the complete incompatibility between the various implementations and the esoteric API they usually provide. As a result, it is not always easy to experiment with and compare such systems.

Since the execution model of OpenMP is based on a globally shared memory, combining OpenMP and software DSM systems has been proposed by many researchers as a convenient means of leveraging a cluster, matching the programmer-friendliness of OpenMP with the DSM layer that abstracts away the underlying distributed architecture. Any peculiarities of the DSM layer are completely hidden from the programmer and are left to the compiler and runtime system to handle.

Almost all OpenMP implementations are based on a tight coupling between the compiler and the runtime library. The whole system targets SMP machines or clusters but usually not both. Even in the few cases that support both, there is a fixed, built-in threading library and a software DSM core and the generated code targets them specifically, making it almost impossible to experiment with alternative configurations.

OMPi [6] is an open-source OpenMP system that implements fully OpenMP 2.5 and features a compiler that targets both threading and software DSM libraries. Its runtime system has an open architecture that allows arbitrary threading libraries and arbitrary sDSM cores to be employed, by decoupling the actual execution entities from the rest of the system. However, open-architecture, modularity and portability in no way limit performance. OMPi exhibits quite low overheads and scalable performance even when executing nested parallel regions, comparing favorably even with commercial OpenMP systems.

This paper is organized as follows. An overview of OMPi and its translator is given in Section 2. In Section 3 we present the general runtime architecture of OMPi and give details of the portion that is mostly independent of the actual execution entity employed. Sections 4 and 5 deal with the two execution entities targeted by OMPi, threads and sDSM processes correspondingly. Experimentation with both entities is given in Section 6, while Section 7 summarizes related work. Finally, Section 8 concludes the paper.

2 Overview of OMPi

OMPi consists of a source-to-source compiler for OpenMP/C V2.5 and a runtime system that orchestrates the execution of the produced program. The compiler generates a transformed C program augmented with calls to the runtime system; the system's compiler links it with the runtime library and produces the final executable.

Let us call 'OpenMP threads' the threads that are implied by the OpenMP program. In OMPi, those OpenMP threads can be implemented as portable POSIX threads or machine-specific threads (e.g. Solaris threads) or even heavyweight processes. OMPi, through its compilation process maps OpenMP threads to abstract *execution entities* (EEs). The runtime system provides and controls those execution entities. It has been architected with an internal interface that facilitates the integration of arbitrary EEs. It currently comes with two core libraries that are based on POSIX threads; one is optimized for single-level (non-nested) parallelism, while the other provides limited nested parallelism support through a fixed pool of threads. Two more libraries are available that make use of Solaris instead of POSIX threads. In order to efficiently support unlimited nested parallelism, an additional high-performance library based on portable user-level threads has been developed [7]. Finally, there is one more library that provides heavyweight processes as EEs and interfaces with arbitrary software DSM cores, providing transparent execution on clusters. We have successfully integrated a number of DSM cores, including TreadMarks, JiaJia, Mocha, Mome and Parade [17, 11, 18, 13, 16]. In the next sections we discuss the details of OMPi's runtime architecture and provide experimental results for a multitude of different EEs.

2.1 The compiler

OMPi's source-to-source translator takes as input C source code with OpenMP directives and outputs transformed but equivalent C code augmented with calls to OMPi's runtime system. In its current version, it features a parser capable of understanding programs with C99 syntax and OpenMP V2.5 directives. During parsing, which is the first phase of the compilation process, an abstract syntax tree (AST) is built, which represents the original program. The AST is the input of the second (transformation) phase. The transformer visits the tree nodes and acts whenever a node containing an OpenMP statement is met; it then replaces the whole subtree rooted at that node by a new one which mostly maintains the original block of statements but has additional calls to the runtime system inserted at appropriate places.

While some transformations are relatively intuitive, some others are quite involved. The most crucial one is that of a `parallel` construct. OMPi's translator follows the outlining [3] approach, where the portion of code enclosed in the `parallel` construct is removed and placed within a new function, which will be executed by the threads (or processes) that will be created. The most important problem arising is that of variable visibility, particularly for variables that are to be shared but are non-global. For the following piece of code

```
1  int a;
2  void f() {
3    int b, c, d;
4    #pragma omp parallel private(d)
5      a = b+c+d;
6  }
```

the result of the transformation is given in Fig. 1. The outlined code is moved to function `_thrFunc0_` that is to be executed by all threads. Global variable `a` (line 1) needs no special treatment since global variables are by nature shared among threads. Variable `d` must be private to each thread; this is achieved easily by cloning `d`'s declaration in the produced function. On the other hand, `b` and `c` (line 3) are to be shared but are non-global, lying on the stack segment. Sharing is achieved by creating pointers to those variables and passing them explicitly to the thread function; threads can access them through a call to `ort_get_shared_vars()`. This also necessitates the transformation of the original code (line 5) since in the new function `a` and `b` are now pointers.

The third (final) phase of the compilation process simply traverses the transformed AST and prints out the corresponding C code. The resulting program is compiled by the system's native C compiler and linked with the runtime library producing the final executable.

2.2 Support for processes

OMPi's compiler can also target execution entities that are processes, in addition to threads. The transformations and the produced code are almost identical for both targets, save a couple of differences. In particular,

- Because global variables which are shared among OpenMP threads, are by nature private to each pro-

```
int a;    /* shared global */

static void * _thrFunc0_(void *_arg) {
  struct { int (*b); int (*c); }
     *_shvars = ort_get_shared_vars();
  int (*b) = _shvars->b;  /* shared non-global */
  int (*c) = _shvars->c;  /* shared non-global */
  int d;                  /* private() var */

  a = (*b) + (*c) + d;    /* pointers */
  return (void *) 0;
}

void f() {
  int b, c, d;
  struct { int (*b); int (*c); }
    _shvars = { &b, &c };

  ort_execute_parallel(-1,_thrFunc0_,&_shvars);
}
```

Figure 1: Transformed parallel construct.

cess, they must be explicitly placed in a shared memory area. The compiler produces *constructor* code that allocates space for all global variables during program startup, just before `main()` is executed, through special calls to the runtime library.

- All global variables are transformed to pointers that point to appropriate offsets within the allocated shared area; every occurrence of such a variable is replaced correspondingly all over the program.

Notice that nothing changes for shared non-global variables. In contrast, other approaches use complex and CPU-specific handling of stack frames [15] or involve, for every parallel region, (a) creating a shared memory area, (b) copying these variables there, (c) copying them back to their original area at the end of the parallel region and (d) releasing the shared memory area [12, 23]. This time consuming process is avoided in OMPi, and is resolved entirely in its runtime system which is presented next.

3 The runtime architecture of OMPi

The runtime system of OMPi provides the execution entities that will carry out the work of OpenMP threads and controls their operation and synchronization. It consists of two modules; the first module (ORT) groups EEs, coordinates them and schedules their execution within worksharing regions, but *it does not implement them*. The second module (EELIB) is the one that actually implements the execution entities. A multitude of EELIB libraries are currently available, adhering to a unified interface. ORT's operation is independent of the actual EELIB employed.

When called to execute a parallel region (through the `ort_execute_parallel()` call in Fig. 1), ORT enters a negotiation phase with EELIB, asking for a particular number of EEs, depending on what the program requests and whether nested parallelism and the dynamic adjustment of the number of threads are enabled or not. After EELIB confirms the availability of EEs, it gets instructed by ORT to release them in a bunch, as a team. When an EE from the team commences execution, its very first obligation is to call `ort_get_ee_work()`, which supplies all the information for the work the EE is supposed to do. Specifically, among other things, it provides a pointer to the function to be executed (`_thrFunc0_` in Fig. 1). In addition, ORT constructs and manages the EE's *control block* (`eecb`). The control block contains everything ORT needs in order to schedule the EE, including the size of the team, the id of the EE within the team, its parallel level and a pointer to the `eecb` of the team's parent. Through the latter pointers, ORT maintains a dynamic tree of `eecb`s which grows whenever a new team of EEs is unleashed and shrinks whenever a team completes the execution of a parallel region.

Upon startup, the sole EE running is the *initial* EE and operates in level 0. Whenever an EE encounters a parallel region, it becomes the parent of the spawned team; if the parent is in level i, all its children lie in level $i + 1$. There is no prerequisite regarding an EE's level, providing thus full and unlimited support for nested parallelism, as long as EELIB is willing to supply EEs. The parent also becomes a member of the team, with id 0, and is called the *master* EE of the team.

The `eecb` holds additional information in the case the EE becomes the parent of a new team. This includes a barrier structure for synchronizing the team members, a `copyprivate` staging area for `single` constructs that require it and a structure with scheduling information for worksharing regions.

3.1 Workshare region scheduling

OpenMP defines three workshare constructs, `for`, `sections` and `single` whereby the work is divided appropriately among the participating EEs. These code regions are normally blocking, in the sense that they conclude with an implied barrier that synchronizes the EEs before letting them continue their execution. However, when a `nowait` clause is present, there is no implied barrier and the region is non-blocking; such regions present bookkeeping complications. This is because some EEs of the team may advance without notice to subsequent workshare regions, the number of which may not be known statically at compile time. Consequently, there may exist multiple *active* non-blocking regions and different EEs may be in different regions at any given time; in contrast, there can be at most 1 blocking region active.

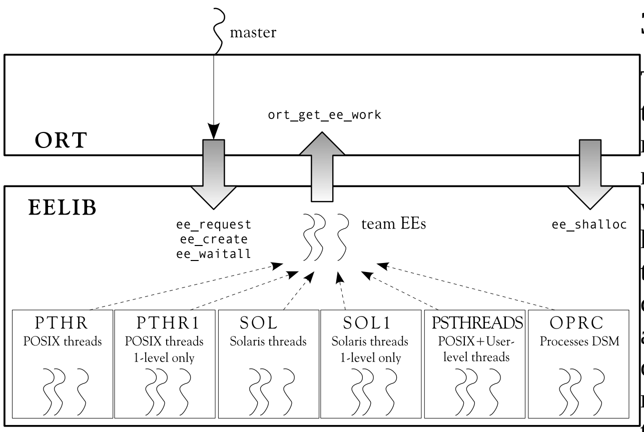


Figure 2: EELIBs and interface with ORT.

Solutions to this problem include bookkeeping using a dynamically allocated list of workshare region structures [2] or avoiding the problem altogether by disallowing more than one non-blocking regions to be simultaneously active, as in the runtime library of the Omni compiler [23]. The approach followed in OMPi is similar to [26]. In the control block of the parent of a team, ORT maintains a pre-allocated workshare queue of fixed size (MAXWS) with bookkeeping information about each active workshare region. Stored information includes construct-specific data (e.g. the number of remaining sections for a `section` construct; the upper bound and the increment for a `for` construct; locks for protecting access to this data by the EEs of the team) plus queue-related data, such as the number of EEs that have entered and the number of EEs that have exited (finished) this region. When the tail and the head of the queue are MAXWS regions apart, i.e. there are MAXWS simultaneously active regions, any EE that tries to activate a new region gets blocked until the tail of the queue advances. This way, we avoid the cost of dynamic adjustment of the capacity of the queue, without introducing the artificial barrier required in [26].

ORT optimizes the operation of the workshare queue by using lock-free accesses when possible and by employing atomic operations if available, resorting to plain locking only when necessary. A final optimization is the avoidance of full initialization of the queue. Every time a new team of EEs is created, all regions of the queue must be properly initialized by the parent before being put to use. If MAXWS is not small this results in a major overhead. ORT avoids this by initializing only the first region of the queue; the first EE to enter a new non-blocking region is responsible for initializing the next region in the queue. This way, at any given time, the queue has one extra region ready for use.

3.2 The interface with EELIB

The EELIB is responsible for providing all execution entities, except the master EE, and three types of locks: normal, nested and spin locks. The first two types are made available to the OpenMP application programmer while the third type is used internally by ORT. EELIB has no other obligation, as everything else is handled entirely by ORT. Upon initialization, EELIB announces its capabilities to ORT, which include support of nested parallelism, support for dynamic adjustment of the number of EEs, the maximum number of EEs and the maximum number of nested parallelism levels supported. Regarding the EEs, EELIB implements 3 functions that are called by ORT (see Fig. 2): `ee_request()`, `ee_create()` and `ee_waitall()`. The first two are used when creating a new team. The parent asks for a particular number of EEs, and `ee_request()` replies with the actual number it can provide. In EELIBs that do not support nested parallelism `ee_request()` always returns 0 when called from level ≥ 1. If `ee_request()` returns a number smaller than the one requested and the dynamic adjustment of the number of EEs is not enabled/supported, the program terminates. If everything was OK, ORT calls `ee_create()` to instruct EELIB to actually create the requested EEs. Upon completion of the parallel region, the master calls `ee_waitall()` and blocks until all the other EEs in the team have terminated.

4 Threading libraries

A number of EELIBs that provide thread EEs are available for OMPi, including libraries that are based on POSIX threads, Solaris threads and a portable user-level thread package (Fig. 2). The default library of OMPi (PTHR) uses a pool of POSIX threads as its EEs. The pool is formed upon initialization, and the number of pre-created threads is equal to N, which is the maximum of the number available processors and the number specified in the `OMP_NUM_THREADS` environmental variable; after that the pool size cannot change. The threads spin, yielding the processor frequently, waiting for work. Upon a request for a number of EEs, no matter at what parallelism level, the EELIB checks the pool population; it can only supply as many threads as are available at that particular moment in the pool. This means that nested parallelism is supported as long as the total number of threads in all nesting levels is at most N; if the programmer has chosen to disable the dynamic adjustment of the number of EEs, then the library may not be able to supply the requested number of threads. Upon completion of its work, a thread returns to the pool, decrementing first a counter in the parent's `eecb`. The master thread blocks at the `ee_waitall()` call until this counter becomes 0, which means that all threads of the team have finished.

The default EELIB of OMPi, although providing limited support of nested parallelism, is mostly optimized for single-level parallelism. For cases where deep nesting levels are expected, we have developed a high-performance library called PSTHREADS [7]. The library is based on portable user-level threads that are executed by its virtual processors, which never exceed the number of physical processors. In contrast to most OpenMP implementations, which utilize kernel-level threads, OMPi maps OpenMP threads to non-preemptive PSTHREADS. This approach minimizes the OpenMP runtime overheads, especially when nested parallelism is enabled, and manages to exploit fine-grain parallelism. In addition, the internal thread scheduling scheme of the PSTHREADS library favors the execution of threads on a single processor and improves data locality.

Providing the simple interface described in Section 3.2 is enough to make OMPi utilize any arbitrary threading library. For example, we have incorporated Marcel threads [25] unmodified, by providing an EELIB with about 150 lines of code.

5 Processes and software DSM

OMPi comes with one more EELIB (OPRC) which provides processes as its execution entities. The interface with ORT is exactly the same, except that the EELIB offers one more facility, `ee_shalloc()` which explicitly allocates shared memory areas. In the case of processes, ORT initially allocates an appropriately-sized area and then copies all the shared global variables of the program in there (see Section 2.2). Because non-global variables of the initial EE in the user program may be shared among the children it will create, *the stack of the initial EE* is also placed in shared memory. As such, any of its local variables can be accessible by any other process. Since the initial EE's stack is system-specified and cannot be changed, we achieve the desired effect by switching to a new thread, just before the program's `main()` is executed; this thread has its stack explicitly allocated in the shared memory area and will be the one to call `main()` and ultimately execute the program.

Except for shared variables in the user program, ORT maintains its own internal variables that are to be shared. For example, all scheduling information presented in Section 3.1 is stored in the control block of the master EE, and all team members need to have access to it. For performance reasons the implementation of OMPi's runtime system follows a hybrid approach, whereby shared access to ORT's internal variables is not handled by the underlying sDSM core but is done by explicit message passing, using MPI. To make this possible, upon startup OPRC creates a *server* thread at each node of the cluster. Its sole purpose

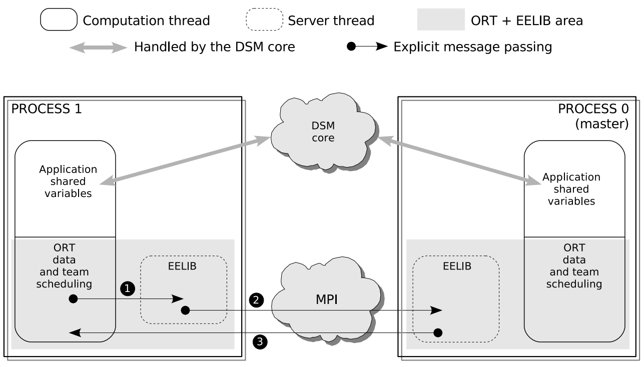
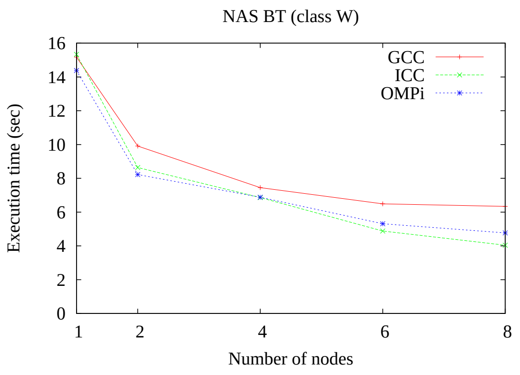


Figure 3: Runtime organization for clusters (hybrid sDSM+MPI).

is to provide asynchronous communication among the local and remote computation threads. The organization is depicted in Fig. 3. Whenever an EE needs access to the team's master `eecb`, a request is made to its local server thread (1). The server thread forwards the request to the master's server thread (2). The latter replies directly to the computation thread with a copy of the control block of the master EE. Local server threads know which the master of the team is because initially it contacts them to distribute the work among the EEs. Recapping, the sole use of the sDSM core is to store, in a consistent manner, the user program's shared variables. Everything else is implemented using MPI.

A number of sDSM cores have been ported to OMPi. To achieve this, OPRC has a unified back-end interface which makes it straightforward to attach arbitrary sDSM libraries. Although most sDSM APIs provide similar functionality, there are some important issues that have to be taken care of. One is the consistency model offered. This directly affects the implementation of the `flush` operation of OpenMP, which is provided by OPRC. For systems offering sequential consistency such as Mome [13] nothing needs to be done, but for others with relaxed consistency models, e.g. Mocha [18], the flush operation is implemented through a full sDSM barrier. One more consideration is the style of shared memory allocation. The implementation of `ee_shalloc()` in OPRC is global, in that all processes ultimately call `ee_shalloc()` in order to complete the operation; this is achieved through appropriate coordination between the server threads and fits exactly the allocation style of many libraries, e.g. JiaJia [11]. In contrast, in TreadMarks [17] allocation is local, done only by one EE, and then followed by an address distribution operation. In this case, `ee_shalloc()` is coded to do nothing when called from any EE but the one that triggered the operation.



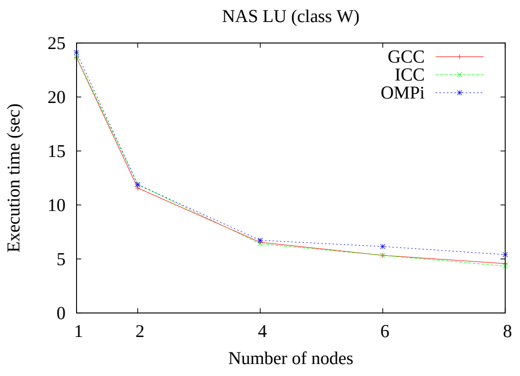


Figure 4: Execution times for BT and LU.

6 Experiments

In this section, we present representative experimental results of our OpenMP environment on both an SMP system and a software DSM cluster. GCC was used as OMPi's back-end compiler.

6.1 Physically shared memory

We conducted our experiments on a server with 4 dual-core Intel Xeon 3.0GHz CPUs running Linux 2.6. We include results for GCC 4.2 and the Intel 10.0 (ICC) compiler.

Figure 4 depicts the execution times for two applications (BT, LU) of the NAS Parallel Benchmarks suite. Since nested parallelism is not exploited in the NAS benchmarks, we present results only for the default EELIB of OMPi. We observe that OMPi is comparable to GCC and ICC in terms of performance and scalability.

If its EELIB is based on PSTHREADS, OMPi supports multiple levels of parallelism in a very efficient way. To evaluate the OpenMP overheads when nested parallelism is used, we have developed an extension of the EPCC microbenchmarks [5]. Table 1 depicts a sample of the runtime overheads for three of the OpenMP synchronization constructs (parallel, for, single). For OMPi, we provide results for both the default EELIB and the one based on PSTHREADS. In this experiment, we have two nesting levels with 8 threads at the outer level and 4 threads at the inner parallel region. Therefore, there are 32 threads that compete for computational resources. OMPi with PSTHREADS exhibits the lowest runtime overheads mainly because it avoids oversubscribing the 8 processing cores.

Table 1:Nested parallelization overheads (μs)

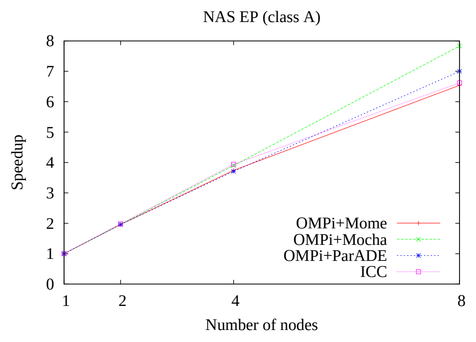
OpenMP compiler	**parallel**	**for**	**single**
GCC 4.2.0	1426.89	246.38	277.23
ICC 10.0	377.82	22.86	15.15
OMPi + PTHR	139.79	155.17	154.31
OMPi + PSTHREADS	6.75	8.22	5.61

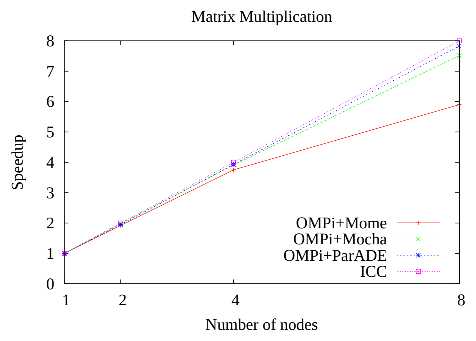
6.2 Software distributed shared memory

Our experiments on sDSM were performed on 8 nodes of a HP XC cluster system. Each node has 2 AMD Opteron 248 processors and 4GB main memory, while the nodes are interconnected with Gigabit Ethernet. We provide results for three page-based software DSM systems that OMPi targets (Mome, Mocha and Parade). Mome supports kernel threads, sequential consistency, and provides several advanced features, like memory mapping, prefetching, and page manager migration. Mocha uses scope consistency similarly to JiaJia but utilizes an improved method that reduces acknowledgment overheads. Finally, Parade is a multithreaded sDSM system that provides home-based lazy release consistency (HLRC) and makes use of MPI. The MPI library used in our experiments for communication and application launching is MPICH2 (1.0.6). We also present results for an evaluation copy of the Intel 10.0 compiler with cluster OpenMP support [10].

For our experiments we use three OpenMP C applications: NAS EP, MM (matrix multiplication) and MD (molecular dynamics). It is important to mention that we have not introduced any modifications to the OpenMP code of these applications in order to run the executables produced by OMPi on the software DSM cluster. For instance, MD is the C version of the sample application available at the official site of OpenMP. In contrast, the Intel compiler requires the user to explicitly annotate both the global and the stack variables that need to be shared.

The speedups of the three OpenMP applications are depicted in Figure 5. We use exactly one OpenMP thread per computing node because Mocha does not support kernel threads. Moreover, we do not run more than one process on a single node because both the sDSM systems and the OpenMP runtime library utilize server threads.





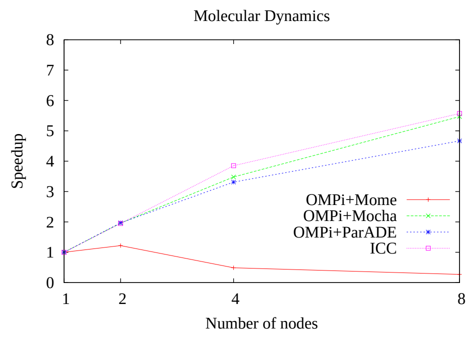


Figure 5: Speedups on the sDSM cluster.

We observe that NAS EP scales efficiently for all OpenMP configurations. This is reasonable because EP is embarrassingly parallel and is not seriously affected by the underlying sDSM protocol. In MM, which multiplies two square matrices of size N=1024, the master thread performs the initialization and then each OpenMP thread (node) computes its statically assigned chunk of iterations. After the parallel region, the master thread accesses the result matrix. Despite the page faults that incur before and after the parallel region, the application scales well for all sDSM systems but Mome. Finally, MD runs for 4096 particles and exhibits lower speedups mainly due to false sharing and the overheads introduced by the implicit movement of data. The performance degradation is significantly higher for Mome due to its sequential consistency protocol.

7 Related work

Apart from commercial/proprietary implementations, the well-known GCC compiler starting from version 4.2 supports OpenMP through its GOMP [21] runtime library. In addition, a number of experimental / research compilers for OpenMP have been developed and used extensively, e.g. [14, 23, 1]. Most implementations target only SMP architectures and are tied to some fixed threading library, which is usually POSIX threads. However, experimentation with other threading libraries is generally required. For example, it has been shown [7] that kernel-level POSIX threads do not always exhibit scalable behavior when utilized in nested parallel levels. In such cases high-performance user-level threads become relevant and OMPi provides an interface for trivial porting. PSTHREADS and Marcel threads [7, 25] have been seamlessly integrated into OMPi runtime. Omni/ST [24] was an experimental version of Omni equipped with the StackThreads/MP library; it provided an efficient but non portable implementation of user-level threading for nested irregular parallelism.

Providing the illusion of shared memory over a cluster of workstations has been pioneered by TreadMarks [17] and has been followed by a multitude of other sDSM libraries, providing similar functionality but potentially higher performance through more relaxed consistency models (e.g. [11, 18, 13]). The idea of utilizing an sDSM library in combination with OpenMP was presented in [20, 12], where the authors implemented a compiler for a subset of OpenMP that targeted TreadMarks specifically. A modified version of TreadMarks is also the target of the Intel compiler for clusters [10]. All other known systems employ custom-designed sDSM libraries as targets to modified research compilers, e.g. [4, 8] based on the NanosCompiler, and [16, 19] based on the Omni compiler. Parade's runtime system [16] follows a hybrid approach, employing MPI for synchronization and scheduling purposes.

8 Conclusion

We presented the runtime architecture of the OMPi compiler for OpenMP/C. OMPi is unique with respect to what it provides. The translator can target execution entities which are either threads or processes. The runtime system is modularized in such a manner that arbitrary thread or sDSM cores can be integrated with minimal effort. OMPi currently supports more that 7 thread libraries and 5 different software DSM implementations, forming an ideal testbed for research and experimentation.

OMPi's performance is highly optimized, especially when targeting threads. We are currently investigating optimization techniques for boosting performance on clusters.

Acknowledgement

Information dissemination of this work was supported by the European Union in the framework of the project “Support of Computer Science Studies in the University of Ioannina” of the “Operational Program for Education and Initial Vocational Training” of the 3rd Community Support Framework of the Hellenic Ministry of Education, funded by national sources and by the European Social Fund (ESF).

References

[1] E. Ayguade, M. Gonzalez, X. Martorell, J. Labarta, N. Navarro, and J. Oliver. NanosCompiler: Supporting Flexible Multilevel Parallelism in OpenMP. *Concurrency: Practice and Experience*, 12(12):1205–1218, Oct. 2000.

[2] C. Brunschen. OdinMP/CCp – A Portable Compiler for C with OpenMP to C with POSIX Threads. Master’s thesis, Dept. of Inform. Techn., Lund University, Sweden, 1999.

[3] J.-H. Chow, L. E. Lyon, and V. Sarkar. Automatic parallelization for symmetric shared-memory multiprocessors. In *Proc. Conf. of the Centre for Advanced Studies on Collaborative Research (CASCON’96)*, Toronto, Canada, Nov. 1996.

[4] J. J. Costa, T. Cortes, X. Martorell, E. Ayguade, and J. Labarta. Running OpenMP Applications Efficiently on an Everything-Shared SDSM. *Journal of Parallel and Distributed Computing*, 66(5):647–658, 2006.

[5] V. V. Dimakopoulos, P. E. Hadjidoukas, and G. C. Philos. A Microbenchmark Study of OpenMP Overheads Under Nested Parallelism. In *Proc. of the Int’l Workshop on OpenMP (IWOMP ’08)*, West Lafayette, USA, May 2008.

[6] V. V. Dimakopoulos, E. Leontiadis, and G. Tzoumas. A Portable C Compiler for OpenMP V.2.0. In *Proc. of the 5th European Workshop on OpenMP (EWOMP ’03)*, Aachen, Germany, Oct. 2003.

[7] P. E. Hadjidoukas and V. V. Dimakopoulos. Nested Parallelism in the OMPi OpenMP C Compiler. In *Proc. of the European Conference on Parallel Computing (EUROPAR ’07)*, Rennes, France, Aug. 2007.

[8] P. E. Hadjidoukas, E. D. Polychronopoulos, and T. S. Papatheodorou. OpenMP Runtime Support for Clusters of Multiprocessors. In *Proc. of the Int’l Workshop on OpenMP Applications and Tools (WOMPAT ’03)*, Toronto, Canada, 2003.

[9] M. Hess, G. Jost, M. Müller, and R. Rühle. Experiences Using OpenMP Based on Compiler Directed Software DSM on a PC Cluster. In *3rd Workshop on OpenMP Applications and Tools (WOMPAT 2002)*, Aug. 2002.

[10] J. P. Hoeflinger. Extending OpenMP to Clusters, 2006. White Paper, Intel Corporation.

[11] W. Hu, W. Shi, and Z. Tang. JIAJIA: An SVM System Based on A New Cache Coherence Protocol. In *Proc. of the 7th Int’l Conference on High Performance Computing and Networking (HPCN ’99)*, Amsterdam, The Netherlands, Apr. 1999.

[12] Y. C. Hu, H. L. amd A. L. Cox, and W. Zwaenepoel. OpenMP for Networks of SMPs. *Journal of Parallel and Distributed Computing*, 60:1512–1530, 2000.

[13] Y. Jéegou. Implementation of Page Management in Mome, a User-Level DSM. In *Proc. of the 3rd IEEE International Symposium on Cluster Computing and the Grid (CCGRID ’03)*, Tokyo, Japan, May 2003.

[14] S. Karlsson. A Portable and Efficient Thread Library for OpenMP. In *Proc. 6th European Workshop on OpenMP (EWOMP ’04)*, Stockholm, Sweden, Oct. 2004.

[15] S. Karlsson. An Introduction to Balder—An OpenMP Run-time Library for Clusters of SMPs. In *Proc. of the 1st Int’l Workshop on OpenMP (IWOMP ’05)*, Eugene, USA, 2005.

[16] Y.-S. Kee, J.-S. Kim, and S. Ha. ParADE: An OpenMP Programming Environment for SMP Cluster Systems. In *Proc. of the 15th Int’l Conference for High Performance Computing, Networking, Storage, and Analysis (SC ’03)*, Phoenix, AZ, USA, Nov. 2003.

[17] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proc. of the Winter 94 USENIX Conference*, San Francisco, CA, USA, Jan. 1994.

[18] K. Kise, T. Katagiri, H. Honda, and T. Yuba. Evaluation of the Acknowledgment Reduction in a Software-DSM System. In *Proc. of the 6th Int’l Conf. on Parallel Processing and Applied Mathematics (PPAM ’05)*, Poznan, Poland, Sept. 2005.

[19] T.-Y. Liang, S.-H. Wang, C.-K. Shieh, C.-M. Huang, and L.-I. Chang. Design and Implementation of the OpenMP Programming Interface on Linux-based SMP Clusters. *Journal of Information Science and Engineering*, 2:785–798, 2006.

[20] H. Lu, Y. C. Hu, and W. Zwaenepoel. OpenMP on Networks of Workstations. In *Proc. ACM/IEEE Conf. on High Perf. Networking and Computing (SC’98)*, Orlando, FL, 1998.

[21] D. Novillo. OpenMP and automatic parallelization in GCC. In *Proc. of the 2006 GCC Summit*, Ottawa, Canada, 2006.

[22] OpenMP Architecture Review Board. *OpenMP C and C++ Application Program Interface, Version 2.5*. May 2005.

[23] M. Sato, S. Satoh, K. Kusano, and Y. Tanaka. Design of OpenMP Compiler for an SMP Cluster. In *Proc. of the 1st European Workshop on OpenMP (EWOMP ’99)*, Lund, Sweden, Sept. 1999.

[24] Y. Tanaka, K. Taura, M. Sato, and A. Yonezawa. Performance Evaluation of OpenMP Applications with Nested Parallelism. In *Proc. of the Fifth Workshop on Languages, Compilers and Run-Time Systems for Scalable Computers (LCR ’00)*, Rochester, NY, USA, May 2000.

[25] S. Thibault. A Flexible Thread Scheduler for Hierarchical Multiprocessor Machines. In *Proc. of the 2nd Int’l Workshop on Operating Systems, Programming Environments and Management Tools for High-Performance Computing on Clusters (COSET-2)*, Cambridge, USA, June 2005.

[26] G. Zhang, R. Silvera, and R. Archambault. Structure and algorithm for implementing OpenMP workshares. In *Proc. of the 5th Workshop on OpenMP Applications and Tools (WOMPAT ’04)*, Houston, TX, USA, 2004.