Procedia Computer Science

Volume 29, 2014, Pages 2422–2431

ICCS 2014. 14th International Conference on Computational Science

POSH: Paris OpenSHMEM
A High-Performance OpenSHMEM Implementation for Shared Memory Systems

Camille Coti
camille.coti@lipn.univ-paris13.fr [*]

LIPN, CNRS-UMR7030,
Université Paris 13, F-93430 Villetaneuse, France

Abstract
In this paper we present the design and implementation of POSH, an Open-Source implementation of the OpenSHMEM standard. We present a model for its communications, and prove some properties on the memory model defined in the OpenSHMEM specification. We present some performance measurements of the communication library featured by POSH and compare them with an existing one-sided communication library. POSH can be downloaded from http://www.lipn.fr/~coti/POSH.

Keywords:

1 Introduction

The drive toward many-core architectures has been tremendous during the last decade. Along with this trend, the community has been searching, investigating and looking for programming models that provide both control on the data locality and flexibility of the data handling.

SHMEM was introduced by Cray [8] in 1994, followed shortly later by SGI [19]. In an effort to provide a homogeneous, portable standard for the language, the OpenSHMEM consortium released a specification for the application programming interface [16]. The final version of OpenSHMEM 1.0 was released in January 2012.

The OpenSHMEM standard is a programming paradigm for parallel applications that uses single-sided communications. It opens gates for exciting research in distributed computing on this particular communication model.

This paper presents Paris OpenSHMEM (POSH), which is a portable, open-source implementation of OpenSHMEM. It uses a high-performance communication engine on shared

[*]Some experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).

memory based on the Boost library [10], and benefits from the template engine provided by current C++ compilers.

This paper describes the implementation choices and the algorithms that have been used in POSH in order to fit with the memory model and the communication model while obtaining good performance and maintaining portability.

This report is organized as follows; section 2 gives a short overview of the related literature about parallel programming paradigms and distributed algorithms on shared memory and one-sided communication models. Section 3 gives details about the memory model and the communication model which are considered here. Section 4 describes the implementation choices that were made in POSH. Section 5 presents some performance results that were obtained by the current version of POSH. Last, section 6 concludes the report and states some open issues and future works that will be conducted on POSH.

2 Related works

Traditionally, distributed systems are divided into two categories of models for their communications: those that communicate by sending and receiving messages (*i.e.,* message-passing systems) and those that communicate using registers of shared memory where messages are written and read from (*i.e.,* shared memory systems) [21, 11].

Along with the massive adoption of the many-core hardware architecture, researchers and engineers have tried to find the most efficient programming paradigm for such systems. The idea is to take advantage of the fact that processing units (processes or threads) have access to a common memory: those are *shared memory* systems. Unix IPC V5 and posix threads are the most basic programming tools for that. OpenMP [9] provides an easy-to-use programming interface and lets the programmer write programs that look very similar to sequential ones, and the parallelization is made by the compiler. Therefore, the compiler is in charge with data decomposition and accesses. Several data locality policies have been implemented to try to make the best possible guess about where it must be put to be as efficient as possible [22]. OpenMP performs well on regular patterns, where the data locality can be guessed quite accurately by the compiler. Cilk [2] and TBB [18] can also be cited as programming techniques for shared-memory systems.

MPI [12, 14] has imposed itself as the *de facto* programming standard for distributed-memory parallel systems. It is highly portable, and implementations are available for a broad range of platforms. MPICH [15] and Open MPI [13] must be cited among the most widely used open-source implementations. It can be used on top of most local-area communication networks, and of course most MPI implementations provide an implementation on top of shared memory. MPI is often referred to as "the assembly language of parallel computing": the programmer has total control of the data locality, however all the data management must be implemented by hand by the programmer. Moreover, it is highly synchronous: even though specific one-sided communications have been introducted in the MPI2 standard [14], the sender and the receiver must be in matching communication routines for a communication to be performed.

Hence, there exists two opposing trends in parallel programming techniques: programming easiness versus data locality mastering. A third direction exists and is becoming pertinent with many-core architectures: making data locality mastering easier and more flexible for the programmer. UPC can be cited as an example of programming technique that is part of that third category [7]. It provides compiler-assisted automatic loops, automatic data repartition in (potentially distributed) shared memory, and a set of one-sided communications.

One-sided communications are natural on shared memory systems, and more flexible than two-sided communications in a sense that they do not require that both of the processes involved in the communication (origin and destination of the data) must be in matching communication routines. However, they require a careful programming technique to maintain the consistency of the shared memory and avoid race conditions [5].

SHMEM was introduced by Cray [8] as part of its programming toolsuite with the Cray T3 series, and SGI created its own dialecte of SHMEM [19].

Some implementations also exist for high-performance RDMA networks: Portals has been working on a specific support for OpenSHMEM by their communication library [1]. Some other implementations are built on top of MPI implementations over RDMA networks, such as [4] for Quadrics networks or [17] over InfiniBand networks.

In this paper, we propose to use a shared memory communication engine based on the Boost.Interprocess library [10], which is itself using the POSIX `shm` API.

3 Memory model

OpenSHMEM considers a memory model where every process of the parallel application owns a local bank of memory which is split into two parts:

- Its *private memory*, which is accessible by itself only; no other process can access this area of memory.
- Its *public memory*, that can be accessed by any process of the parallel application, in read/write mode.

3.1 Symmetric objects

The public memory of each process is defined as a *symmetric heap*. This notion of symmetry is important because it is a necessary condition for some helpful memory-management properties in OpenSHMEM (see section 4.1.2 and 4.5.2). It means that for any object which is stored in the symmetric heap of a process, there exists an object of the same type and size and the same address, in the symmetric heap of all the other processes of the parallel application.

Dynamically-allocated variables, *i.e.,* variables that are allocated explicitely at run-time, are placed in the symmetric heap. OpenSHMEM provides some functions that allocate and deallocate memory space dynamically in the symmetric heap.

Another kind of data is put into processes' public memory: global, static variables (in C/C++).

All the data that is placed in the processes' symmetric heap and all the global, static variables are remotely accessible by all the other processes. These two kinds of variables are represented in figure 1. The gray areas prepresent the public memory of each process; composed with global static variables and the symmetric heap. The white areas represent the private memory of each process.

3.2 One-sided communications

Point-to-point communications in OpenSHMEM are *one-sided*: a process can reach the memory of another process without the latter knowing it. It is very convenient at first glance, because no synchronization between the two processes is necessary like with two-sided communications. However, such programs must be programmed very carefully in order to maintain memory consistency and avoid potential bugs such as race conditions.

Point-to-point communications in OpenSHMEM are based on two primitives: `put` and `get`.

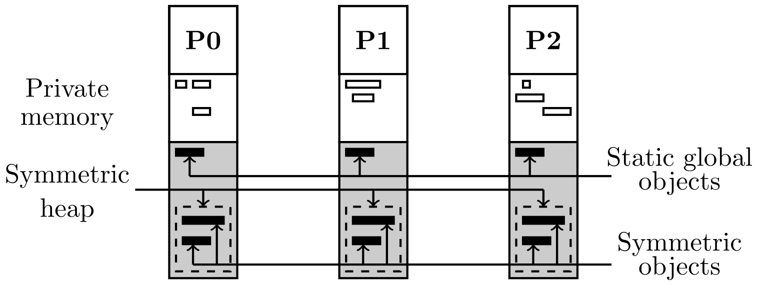

Figure 1: Memory organization with global static objects and data in the symmetric heap. Static objects are remotely accessible, dynamic objects are located in the symmetric heap.

- A `put` operation consists in writing some data at a specific address of remote process's public memory.
- A `get` operation consists in reading some data, or fetching it, from a specific address of a remote process's public memory.

Data movements are made between the public memory of the local process and the private memory of the remote process: a process reads a remote value and stores it in its own private memory, and it writes the value of a variable located in its own private memory into the public memory of a remote process. That memory model and one-sided communications performed on this model have been described more thoroughly in [5].

4 Implementation details

4.1 Shared memory communication engine

POSH relies on Boost's library for inter-process communications `Boost.Interprocess`. In particular, it is using the `managed_shared_memory` class.

Basically, each process's shared heap is an instance of `managed_shared_memory`. Data is put into that heap by an allocation method provided by this class followed by a memory copy. Locks and all the manipluation functions related to the shared memory segment are also provided by Boost.

4.1.1 Memory management

Allocation and deallocation Memory can be allocated in the symmetric heap with the `shmalloc` function. Internally, that function calls the `allocate` function of the `managed_shared_memory` class on the process's shared heap. That class also provides an `allocate_aligned` routine which is called by the `shmemalign` routine. Memory is freed by `shfree` by using a call to the function `deallocate` provided by `managed_shared_memory`.

Remote vs local address of the data These three SHMEM functions are defined as *symmetric* functions: all the processes must call them at the same time. They are required by the OpenSHMEM standard to perform a global synchronization barrier before returning. As a consequence, if all the memory allocations and deallocations have been made in a symmetric way, a given chunk of data will have the same local address in the memory of all the processes.

Hence, we can access a chunk of data on process A using the address it has on process B. That property is extremely useful for remote data accesses.

4.1.2 Access to another process's shared heap

Fact 1. *If all the processing elements are running on the same architecture, the offset between the beginning of a symmetric heap and a symmetric object which is contained by this heap is the same on each processing element.*

Sketch of Proof: Memory allocations which are performed in the symmetric heaps end by a call to a global synchronization barrier. As a consequence, when some space is allocated in a symmetric heap, all the processing elements *must* allocate space (paragraph 6.4 of the OpenSHMEM standard).

Corollary 1. *As a consequence of fact 1, each processing element can compute the address of a variable located in another processing element's symmetric heap by using the following formula:*

$$addr_{remote} = heap_{remote} + (addr_{local} - heap_{local}) \tag{1}$$

4.2 Symmetric static data

The memory model specifies that global, static variables are made accessible for other processes. In practice, these variables are placed in the BSS segment if they are not initialized at compile-time and in the data segment if they are initialized. Unfortunately, there is no simple way to make these areas of memory accessible for other processes.

Therefore, POSH uses a small trick: we put them into the symmetric heap at the very beginning of the execution of the program, before anything else is done.

A specific program, called the *pre-parser*, parses the source code and searches for global variables that are declared as static. It finds out how they must be allocated (size, etc) and generates the appropriate allocation/deallocation code lines.

When the OpenSHMEM library is initialized (*i.e.,* when the `start_pes` routine is called), it dumps the allocation code into the source code. When the program exits (*i.e.,* when the keyword `return` is found in the main function), the deallocation code lines are inserted before each `return` keyword.

4.3 Datatype-specific routines

OpenSHMEM defines a function for each data type. A large part of this code can be factorized by using an extremely powerful feature of the C++ language: templates. The corresponding code is written only once, and then the template engine instanciates one function for each data type. Hence, only one function needs to be written.

That function is called by each of the OpenSHMEM `shmem_*_...` functions. Each call is actually a call to the compiler-generated function that uses the adequate data type. That function is generated at compile-time, not at run-time: consequently, calling that function is just as fast as if it had been written manually.

4.4 Peer-to-peer communications

Peer-to-peer communications are using memory copies between local and shared buffers. As a consequence, memory copy is a highly critical matter of POSH. Several implementations of `memcpy` are featured by POSH in order to make use of low-level hardware capabilities.One of these implementations is activated by using a compiler directive. In order to minimize the number of conditional branches, selecting one particular implementation is made at compile-time rather than at run-time.

4.5 Collective communications

Collective communications rely on point-to-point communications that perform the actual inter-process data movements. Two options are available for these point-to-point communications:

- *Put-based* communications push the data into the next processes;
- *Get-based* communications pull the data from other processes.

4.5.1 Progress of a collective operation

The communication model used by OpenSHMEM and its point-to-point communication engine is particular in a sense that it is using *one-sided* operations. As a consequence, a process can access in read or write mode another process's memory without the knowledge of the latter process. One consequence of this fact is that a process can be involved in a collective communication without having actually entered the call to the corresponding routine yet.

Hence, if a process A must access the symmetric heap of a process B, the former process must check whether or not the latter has entered the collective communication yet. A boolean variable is included in the collective data structure for this purpose.

If the remote process has not entered the collective communication yet, its collective data structure must be initialized remotely.

If some data must be put into a remote process that has yet to initialize its collective data structure, we only copy the pointer to the shared source buffer. The actual memory allocation will be made later. However, only temporary memory allocations are made within collective operations. Buffers that are used as parameters of a collective operation (source, target and work arrays) must be allocated before the call to this operation. Since memory allocations end by a global barrier, no processing element can enter a collective operation if not all of them have finished their symmetric memory allocations.

When a process enters a collective operation, it checks whether the operation is already underway, *i.e.,* whether its collective data structure has already been modified by a remote process. If so, we need to make the actual memory allocation for the local data and copy what has already been put somewhere in another shared memory area.

A process exits the collective communication as soon as its participation to the communication is over. Hence, no other process will access its collective data structure. It can therefore be reset.

4.5.2 Temporary allocations in the shared heap

With some collective communication algorithms, it can be necessary to allocate some temporary space in the shared heap of a given processing element. However, if we allocate some memory in one heap only, we break the important symmetry assumption made in section 4.1.1. Nevertheless, we will see here that actually, they have no impact in the symmetry of the shared heaps outside of the affected collective operation.

Lemma 1. *Non-symmetric, temporary memory allocations in the heap of a subset of the processing elements that are performed during collective operations do not break the symmetry of the heaps outside of the concerned collective operation.*

Sketch of Proof: Semantically, collective operations are symmetric, in a sense that all the concerned processing elements must take part of them. As a consequence, if all the heaps are symmetric before they enter the collective operation and if there is no difference between the state of each heap at the beginning and at the end of the collective operation, hence, the symmetry is not broken.

If, at some point of the progress of a collective operation, a non-symmetric memory allocation is performed in a shared heap, it is only temporary and will be freed by the time the corresponding PE exits the collective operation, receovering the symmetry between the heaps.

4.6 Locks and atomic operations

Boost provides *named mutexes* for locking purpose. These locks are interesting, because they can be specific for a given symmetric heap. Each process uses the same given name for a given chunk of data on a given symmetric heap. Using a mutex that locally has the same name as all the other local mutexes, processes ensure mutual exclusion. Hence, we can make sure that a chunk of data is accessed by one process only.

Boost also provides a function that executes a given function object *atomically* on a managed shared memory segment such as the one that is used here to implement the symmetric heap. Hence, we can perform atomic operations on a (potentially remote) symmetric heap.

5 Performance and experimental results

This section presents some evaluations of the performance achieved by POSH. Time measurements were done using `clock_gettime()` on the `CLOCK_REALTIME` to achieve nanosecond precision. All the programs were compiled using `-Ofast` if available, `-O3` otherwise. Each experiment was repeated 20 times after a warm-up round. We measured the time taken by data movements (put and get operations) for various buffer sizes.

5.1 Memory copy

Since memory copy (`memcpy`-like) is a highly critical function of POSH, we have implemented several versions of this routine and evaluated them in a separate micro-benchmark. The compared performance of these various implementations of `memcpy()` is out of the scope of this paper. A good description of this challenge, the challenges that are faced and the behavior of the various possibilities can be found in [20].

The goal of POSH is to achieve high-performance while being portable. As a consequence, the choice has been made to provide different implementations of `memcpy` and let the user choose one at compile-time, while providing a default one that achieves reasonably good performance across platforms.

We have compared several implementations of `memcpy` on various platforms that feature different CPUs: an Intel Core i7-2600 CPU running at 3.40GHz (Maximum), a Pentium Dual-Core CPU E5300 running at 2.60GHz (Caire), an AMD Athlon 64 X2 Dual Core Processor 5200+ (Jaune), a large NUMA node featuring 4 CPUs with 10 physical cores each (20 logical cores with hyperthreading), Intel Xeon CPU E7-4850 running at 2.00GHz (Magi10) and a NUMA node featuring 2 CPUs with 2 cores each, Dual-Core AMD Opteron Processor 2218 running at 2.60GHz (Pastel[6]). All the platforms are running Linux 3.2, except Jaune (2.6.32) and Maximum (3.9). The code was compiled by gcc 4.8.2 on Maximum, Caire and Jaune, gcc 4.7.2 on Pastel and icc 13.1.2 on Magi10.

5.2 POSH communication performance

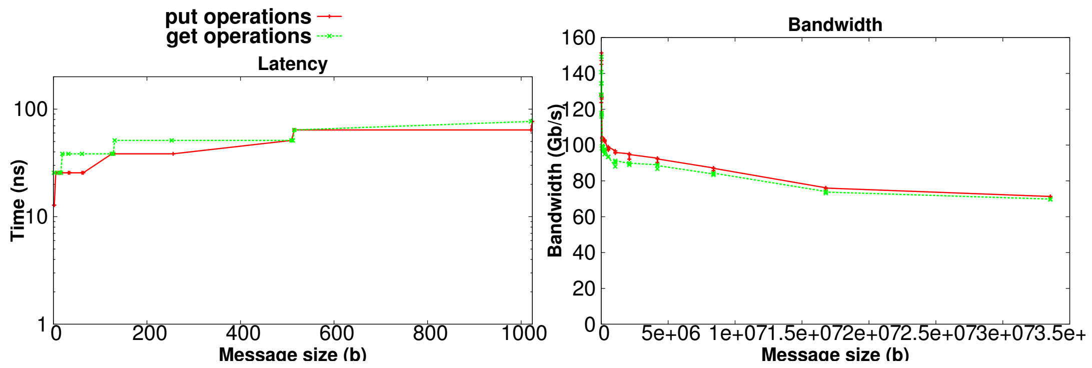

Figure 2: Communication performance of Paris OpenSHMEM on Maximum.

We evaluated the communication performance obtained with POSH. On Caire, Magi10 and Pastel, we used the stock `memcpy` for data movements. On Jaune and Maximum, we used both the SSE-based implementation and the sock `memcpy`. Table 1 present the latency and bandwidth obtained by put and get operations with POSH. Figure 2 plots the latency and bandwidth obtained on Maximum.

On the "fast" machines (Caire, Magi10 and Maximum), the latency is too small to be measured precisely by our measurement method. We can see on table 1 that the latency has the same order of magnitude as the one obtained by a `memcpy` within the memory space of a single process. However, measuring the latency on regular communication patterns gives an indication on the overall latency of the communication engine, but may be different from what would be obtained on more irregular patterns, where the segment of shared memory is not in the same cache memory as the process that performs the data movement. In the latter case, the kernel's scheduling performance is highly critical.

Similarly, the bandwidth obtained by POSH has little overhead compared with the one obtained by a `memcpy` within the memory space of a single process. We can conclude here that our peer-to-peer communication engine adds little overhead, no to say a negligible one, and inter-process communications are almost as fast as local memory copy operations.

Table 1: Comparison of the performance observed by put and get operations with POSH

	SHMEM latency (ns)			
	Best copy		`memcpy`	
	get	put	get	put
Caire	38.40	38.40	38.40	38.40
Jaune	1741.85	1665.90	1667.90	1663.90
Magi10	38.40	38.40	38.40	38.40
Maximum	38.40	38.40	38.40	38.40
Pastel	1830.40	1689.60	1830.40	1689.60

	SHMEM bandwidth (Gb/s)			
	Best copy		`memcpy`	
	get	put	get	put
Caire	18.36	18.38	18.36	18.38
Jaune	17.62	17.55	10.52	10.59
Magi10	20.46	20.16	20.46	20.16
Maximum	74.09	76.15	68.51	69.28
Pastel	26.07	25.50	26.07	25.50

5.3 Comparison with another communication library

We used a similar benchmark to evaluate the communication performance of Berkeley UPC, whose communication engine, GASNet [3], uses `memcpy` to move data. As a consequence, the results obtained here must be compared to those obtained in the previous sections with the stock `memcpy`. Here again, we can see that BUPC inter-process data movement operations have little overhead compared to a memory copy that would be performed within the memory space of a single process.

Table 2: Comparison of the performance observed by put and get operations with UPC

	UPC latency (ns)	
	get	put
Caire	39.40	37.55
Jaune	1623.90	1623.90
Magi10	73.80	54.90
Maximum	26.75	25.00
Pastel	2025.10	1689.95

	UPC bandwidth (Gb/s)	
	get	put
Caire	18.03	18.45
Jaune	9.95	10.63
Magi10	18.64	16.33
Maximum	67.45	68.86
Pastel	23.52	25.06

We can see here that both POSH and another one-sided communication library (Berkeley UPC) have performance that are close to a memory copy within the address space of a single process. Besides, we have seen how the performance can benefit from a tuned memory copy routine.

6 Conclusion and perspective

In this paper, we have presented the design and implementation of POSH, an OpenSHMEM implementation based on a shared memory engine provided by Boost.Interprocess, which is itself based on the POSIX `shm` API. We have presented its architecture, a model for its communications and proved some properties that some implementation choices rely on. We have presented an evaluation of its performance and compared it with a state-of-the-art implementation of UPC, another programming API that follows the same communication model (one-sided communications).

We have seen that POSH achieves a performance which is comparable with this other library and with simple memory copies. We have also shown how it can be tuned in order to benefit from optimized low-level routines.

That communication model opens perspectives on novel work on distributed algorithms. The architectural choices that were made in POSH make it possible to use it as a platform for implementing and evaluating them in practice. For instance, locks, atomic operations and collective operations are classical problems in distributed algorithms. They can be reviewed and re-examined in that model in order to create novel algorithms with an original approach.

References

[1] B. W. Barrett, R. Brightwell, K. S. Hemmert, K. T. Pedretti, K. B. Wheeler, and K. D. Underwood. Enhanced support for OpenSHMEM communication in Portals. In *Hot Interconnects*, pages 61–69, 2011.

[2] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. *Cilk: An efficient multithreaded runtime system*, volume 30. ACM, 1995.

[3] D. Bonachea. GASNet Specification, v1.1. Technical Report UCB/CSD-02-1207, U.C. Berkeley, 2002.

[4] R. Brightwell. A new MPI implementation for Cray SHMEM. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 11th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'04)*, pages 122–130, 2004.

[5] F. Butelle and C. Coti. Data coherency in distributed shared memory. *IJNC*, 2(1):117–130, 2012.

[6] F. Cappello, E. Caron, M. Dayde, F. Desprez, Y. Jegou, P. V.-B. Primet, E. Jeannot, S. Lanteri, J. Leduc, N. Melab, G. Mornet, B. Quetier, and O. Richard. Grid'5000: A large scale and highly reconfigurable grid experimental testbed. In *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing CD (SC—05)*, pages 99–106, Seattle, Washington, USA, Nov. 2005. IEEE/ACM.

[7] U. Consortium. UPC Language Specifications, v1.2. Technical Report LBNL-59208, Lawrence Berkeley National, 2005.

[8] I. Cray Research. SHMEM Technical Note for C. Technical Report SG-2516 2.3, 1994.

[9] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5:46–55, January 1998.

[10] B. Documentation. Chapter 9. boost interprocess. Available at `http://www.boost.org/doc/libs/1_55_0/doc/html/interprocess.html`.

[11] S. Dolev. *Self-Stabilization.* MIT Press, 2000.

[12] M. P. I. Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, Department of Computer Science, University of Tennessee, Apr. 1994. Tue, 22 May 101 17:44:55 GMT.

[13] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 11th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'04)*, pages 97–104, Budapest, Hungary, September 2004.

[14] A. Geist, W. D. Gropp, S. Huss-Lederman, A. Lumsdaine, E. L. Lusk, W. Saphir, A. Skjellum, and M. Snir. MPI-2: Extending the message-passing interface. In L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *1st European Conference on Parallel and Distributed Computing (EuroPar'96)*, volume 1123 of *Lecture Notes in Computer Science*, pages 128–135. Springer, 1996.

[15] W. Gropp. MPICH2: A New Start for MPI Implementations. In D. Kranzlmller, J. Volkert, P. Kacsuk, and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 2474 of *Lecture Notes in Computer Science*, pages 37–42. Springer Berlin / Heidelberg, 2002.

[16] H. P. C. T. group at the University of Houston and O. R. N. L. Extreme Scale Systems Center. OpenSHMEM application programming interface, version 1.0 final. `http://www.openshmem.org`, Jan. 2012.

[17] J. Liu, J. Wu, and D. K. Panda. High Performance RDMA-Based MPI Implementation over InfiniBand. In *17th Annual ACM International Conference on Supercomputing (ICS'03)*, 2003.

[18] J. Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism.* O'Reilly Media, Inc., 2010.

[19] SGI. SHMEM API for Parallel Programming . `http://www.shmem.org`.

[20] Z. Smith. Bandwidth: a memory bandwidth benchmark for x86 / x86_64 based Linux/Windows/-MacOSX. `http://zsmith.co/bandwidth.html`, June 2010.

[21] G. Tel. *Introduction to Distributed Algorithms.* Cambridge University Press, 1994.

[22] T.-H. Weng and B. M. Chapman. Implementing OpenMP Using Dataflow Execution Model for Data Locality and Efficient Parallel Execution. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, IPDPS '02, pages 180–, Washington, DC, USA, 2002. IEEE Computer Society.