MPC++ Performance for Commodity Clustering

Yoshiaki Sakae[1] and Satoshi Matsuoka[2]

[1] Tokyo Institute of Technology, Japan
[2] Tokyo Institute of Technology/JST, Japan
{sakae,matsu}@is.titech.ac.jp

Abstract. In order to verify the viability finer-grained parallel language MPC++, which had originally been developed for Myrinet-specific environments, we performed ports on top of different breeds of MPI, to be executed on two networks of large performance/cost difference, as well as porting NPB 2.3 apps to test ease of expressiveness of parallel programs. Results were positive, (a) the port of the NPB 2.3 apps were effortless, (b) small penalty of additional MPI layer was negligible for NPB applications, and (c) for large data sets, MPC++/MPI on the 100Base-T network was competitive to both the C+MPI on Myrinet, and the original implementation of MPC++ on PM/Myrinet.

1 Introduction

Although commodity "Beowulf" clusters are becoming widespread, programming on such with languages with the class of parallel languages that provide finer-grained multi-threading and fast message passing at the language level has been believed to require expensive messaging hardware with low latency and high bandwidth. Examples are parallel object-oriented language MPC++[1], *independently* developed in the past at the Real World Computing Partnership (RWCP), Split-C[2](UC-Berkeley) and Charm++[3](UIUC). In particular, the original MPC++ assumes a specialized user-level messaging library PM[4] on top of fast and relatively expensive networks, such as Myrinet[5].

We claim that such languages still lacks systematic studies to identify (1) whether they embody sufficient expressive power to easily describe traditional parallel programs, (2) how much performance one expects to maintain/sacrifice by using commodity software/hardware, especially commodity networking layer (including software), for their implementation, and (3) the degree of scalability compared to dedicated software/hardware implementations.

In order to verify the viability of such parallel languages, we took MPC++ which originally required specialized software/hardware layers, in a portable way on top of different breeds of MPI, to be executed on two networks of substantial performance/cost differences, namely, Myrinet and 100Base-T Ethernet. We then investigated whether some NAS Parallel Benchmark (NPB)-2.3 applications (CG, IS) can be ported "naturally" on top of MPC++, to be benchmarked in

such a environment. Finally we performed detailed analysis of the overhead of MPI layer and compared its performance to original implementation of MPC++ on more "expensive" platforms. We have found that, (1) portings of programs from C+MPI were a matter of few hours, including debugging, (2) benchmark apps ported to MPC++ on MPI performs and scales well, compared both to a) the original application written with C+MPI as well as b) the application ported to run on MPC++, but run on the original implementation that employed the fast user-level communication library PM on Myrinet, and (3) benchmark apps on MPC++ on MPI on commodity 100Base-T network ran and scaled competitively, so long as the problem size was large, and the local computation/communication ratio increased as we scaled the problem larger. The results indicate that languages such as MPC++ could be made to run efficiently on a commodity clusters when running 'standard' parallel programs.

2 MPC++ and Port to MPI

The language features of MPC++ v.2.0 Level 0[1] include object-oriented features of C++, finer-grained, user-level multi-threading, fast remote method invocation, remote memory read/write, synchronizing data structures, etc., basically embodying the features of so-called "concurrent object-oriented programming languages". Program code is distributed identically to all physical processors and a process for the program runs on each processor. Each process has several finer-grained threads of control which are not preemptable. A program may locally or remotely invoke a *function instance* with its own thread of control, using the `invoke` and `ainvoke` template functions. Invoking a function instance will involve creation of a new thread and the execution of the function. The original thread invoking the function instance could either block until the end of the invoked function instance execution, or could continue asynchronously in parallel. All variables are processor-local. In order to access variables of remote processors, a *global pointer* must be employed which provides remote variable read-write transparently at the language level.

The original MPC++ implementation was tightly coupled with the underlying PM communication library[4], which in turn was originally tightly coupled with Myrinet. This provided for both low-latency and high-bandwidth communication, as well as finer-grained multithreading via fast user-level communication handling.

In order to achieve portability on commodity clusters, we segregated MPC++ from PM and the custom threading layer, as seen in (Fig. 1). We centralized and re-defined the communication layer API so that various communication libraries could be employed; for the purpose of the paper we have employed MPI, but other communication libraries such as VIA[6] could be used. We reimplemented the threads as standard user-level threads using standard techniques, gaining portability at the expense of efficiency of context switching and synchronization. The resulting artifact, MPC++ on MPI, could be run on a commodity clustering environment without any special software, driver, or hardware installation.

Although there have been middle-tier libraries such as Nexus[7] and Madeleine[8] that provide common infrastructure for communication and multithreading, the point of this particular research is what happens if such features are provided integrally at the language level, rather being employed as a library . Providing features at the language level does have its pros and cons; one could either eliminate the overhead by using language-level semantics, but one could also decrease efficiency by constraining a user to the particular programming model and implementation thereof. For example, the implementation of CC++[9] on Nexus largely assumes coarser-grained user programs than MPC++.

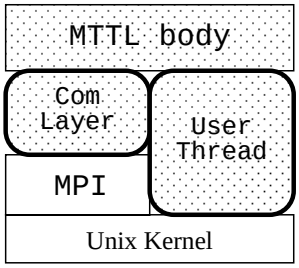

Fig. 1. Structure of MPC++/MPI

3 Evaluation Procedure

Because of the commodity platform, the overhead includes the a) extra software overhead imposed by the MPI software layer, and b) overhead of the underlying communication layer on which MPI is implemented, such as the TCP/IP, including both software and hardware layers. Threading efficiency is sacrificed as mentioned earlier. Also, in the current implementation, collective communication such as "Reduction" or "Barrier" is implemented at the MPC++ level, and currently does not utilize the collective communication features of the underlying MPI. Although this was done for portability reasons this could turn out to be less efficient due to extra software handling overhead.

In order to investigate (1) whether MPC++ allows easy expressiveness of traditional SPMD style parallel programs, (2) how much performance one retains/loses by using commodity software/hardware, and (3) the degree of scalability compared to dedicated software/hardware implementations, we first created the portable implementation of MPC++ as mentioned in Section 2. Then, we ported the CG and IS from NPB-2.3 benchmarks onto MPC++ to identify whether MPC++ could readily express these applications. Next, we performed comprehensive benchmarks of the applications, varying the followings:

- number of processors (1–32),
- problem size (class A and B),
- messaging layer (MPI vs. Native PM),
- different incarnations of MPI (LAM[10] vs. MPICH[11]),
- low-level software messaging layer (PM vs. TCP/IP)
- underlying hardware messaging layer (Myrinet vs. 100Base-T Ethernet).

3.1 Porting NPB CG and IS to MPC++

The NPB-2.3 benchmarks are written in parallel SPMD-style. All are in Fortran+MPI, except for IS which is written in C+MPI.

We had initially considered rewriting the NPB benchmarks in MPC++ from scratch, according to the NPB-2.3 specification[12]. However, we decided to port from their MPI versions, due to: (1) it would be difficult to precisely quantify the difference of respective communication layers, due to drastically different code base, (2) development effort will be substantial, and (3) it was deemed that, even if written from scratch, there is a good change that it will resemble the code structure of a port. This decision still restricted us to the use of NPB to CG and IS, due to most of the NPB programs being written with Fortran+MPI. (We have used the port of CG to C + threads by Yoshio Tanaka at Electrotechnical Laboratory, Tsukuba, Japan.)

Here are the general strategies employed to port CG and IS from their C+MPI versions to the MPC++ versions:

- *The main control thread on node 0 distribute work to slave threads on other nodes.* MPI program written in SPMD style is executed equally on each compute node. Thus, on parallelization by dividing the task to each node, one must make conditionals according to the node number of each node. For example, serialized region must be guarded by a `if` statement to check for execution on node 0. For MPC++, on the other hand, even if the program is written in SPMD style, its main thread will be executed only on node 0 and the other nodes will each wait for a divided task to be assigned from node 0 via remote invocations. Hence we must modify the MPI program so that the control thread on node 0 performs such distribution to slave threads on other nodes on initialization of a parallel region.
- *Translation of message send/receive pair to remote read/write.* In a SPMD MPI program, the programmer usually pairs the `MPI_Send` on the send side and `MPI_Recv` on the receive side manually, and will be careful to avoid any deadlocks since all the nodes will be executing the (possibly blocking) `MPI_Recv`. On the other hand, in MPC++ the programmer can perform one sided access to remote memory with a global pointer, if the remote address is known earlier. Here, one must be careful with synchronization and updating so that correct data will be read by the receiver. This is usually achieved by preceding the reads/writes with a global barrier.
- Translation of other, simpler MPI primitives to MPC++. These include operations such as collective operations.

Although it is not clear whether all SPMD program in this manner, in practice, we have found it rather straightforward to port C NPB programs into MPC++, using the above strategies. The specifics of each program is as below:

Port of CG CG measures the time elapsed in solving an unstructured sparse linear system with the conjugate gradient method. For direct comparison with an MPC++ version, we used the port of the original Fortran+MPI CG by Tanaka which parallelized the program with C+Threads into C+MPI, and then subsequently ported it to MPC++. The modification from C+MPI into MPC++ took approximately 2 hours, and the total modified lines of code are about 60 out of 800 lines.

Port of IS IS is an integer sorting program. Its kernel loop consists of node internal histogram calculation, total histogram gathering, data re-distribution, ranked internal sorting, and subsequent validation. Three of these phases need to communicate: total histogram gathering, exchanging number of data to be re-distributed, and data re-distribution. These are implemented using MPI_Reduce, MPI_Alltoall and MPI_Alltoallv in the original C+MPI version. Due to the simplicity of the data structures, we were able to do the port in approximately 1.5 hours. The total lines of code modified are about 50 out of 1000 lines.

4 Performance Evaluation

Based on the ported codes, we conduct performance evaluations on a major subset of the combinations as described in Section 3, as shown in Table 1. Some combinations are simply not available (e.g., MPC++ on LAM on PM). Here, Native-PM denote the original incarnation of PM on Myrinet, while PM-Ether[13] denote the port of PM onto Ethernet. MPICH-PM[14] is MPICH with PM as the underlying communication layer. Any combination of MPC++ with underlying MPI are the portable implementation in Section 2, while MPC++ directly coupled with PM (MPC++/PM-Ether and MPC++/Native-PM) is the original MPC++ implementation where PM is hardwired.

Table 1. Evaluated Data

	Ethernet	Myrinet
CG	Fortran/LAM	Fortran/MPICH-PM/Native-PM
	C++/LAM	C++/MPICH-PM/Native-PM
	MPC++/LAM	MPC++/MPICH-PM/Native-PM
	MPC++/PM-Ether	MPC++/Native-PM
	MPC++/MPICH-PM/PM-Ether	
IS	C/LAM	C/MPICH-PM/Native-PM
	MPC++/LAM	MPC++/MPICH-PM/Native-PM
		MPC++/Native-PM

4.1 Evaluation Environment

As an evaluation platform, we employ a 32-node portion of our Presto I experimental PC cluster interconnected with 3 networks, i.e., dual independent full 32-port switches (Planex FHSW-3232NW) for the 100Base-T Ethernets, and with Myricom M2M-OCT-SW8 switch × 2 for the Myricom Myrinet (M2M-PCI64A-21). Each node has a single Pentium II 350Mhz processor with 256MB of SDRAM. The operating system is Linux 2.2.14, augmented with the RWCP SCore 3.0 as the clustering environment for Myrinet. Only one 100Base-T network (Intel EtherExpress Pro) was used. We employed the pgcc (Pentium gcc) compiler for all programs with the switches `-O6 -mcpu=i686 -malign-double -fstrength-reduce -funroll-loops -fexpensive-optimizations`.

4.2 Benchmark Results of CG and IS

The graphs Fig. 2, Fig. 3 show the number of nodes vs. elapsed time for CG for different combinations described above. Overall they exhibit reasonable scalability up to 16 nodes for class A and 32 nodes for class B, for all combinations. For class A, as node size increases (up to 32) the working-set size per node becomes quite small and the communication time becomes dominant for Ethernet, as we will verify later. As a result, there is no room for further speedup for node increase, both for C+MPI and MPC++. Fortran+LAM still shows speedup at 32, and this might be due to the relatively coarse-grained communication as compared to the C version (communication granularity for the two programs differ). For Myrinet, on the other hand, we still observe scalability at 32 nodes for all combinations. For Class B, since computation increases with $\mathcal{O}(n^2)$ while communication increases only with $\mathcal{O}(\log(n))$, communication time is not as dominant. Here, all systems except for MPC++/PM-Ether, shows scalability up to and possibly beyond 32 nodes [1].

We note that, even for Class A, as long as we are executing within the number of nodes where a particular combination is exhibiting scalability (up to 16–32 nodes), there is little significant difference in execution time, usually within 10-20%, and even at worst well within the factor of 2.

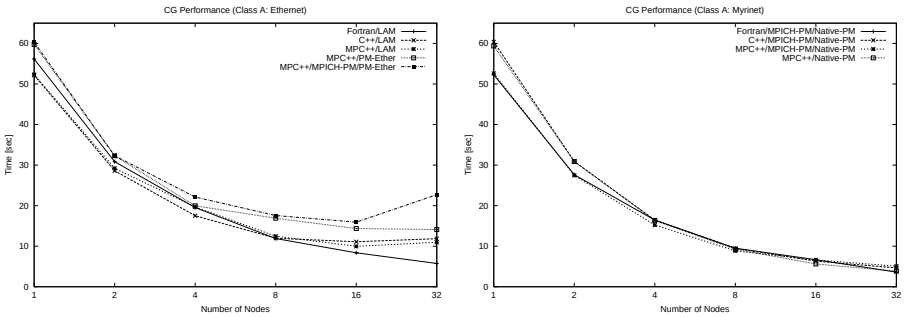

Fig. 2. CG Performance (Class A)

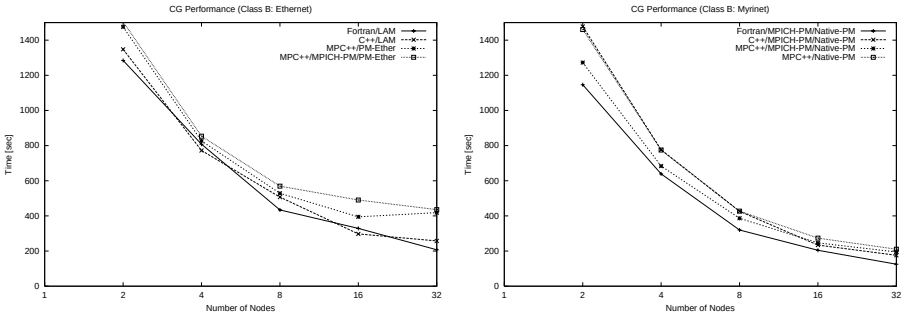

Fig. 3. CG Performance (Class B)

[1] Fig. 3 dose not include MPC++/LAM graph, because of its mysterious behavior on Class B Ethernet

The graphs Fig. 4, Fig. 5 show the number of nodes vs. elapsed time for IS, again for different combinations described above. Since IS involves finer-grained communication compared to CG, communication overhead increases significantly along with the number of nodes. Still, they show some degree of speedup up to 16 nodes for class A and 32 nodes for class B, for all combinations. For Class A, Myrinet provides good scalability even at 32 nodes, irrespective of the intermediate software layer—as a matter of a fact The C/MPICH-PM/Native-PM combination shows almost the same performance as the MPC++/MPICH-PM/Native-PM version, and in fact is superior to the MPC++/Native-PM version. By contrast scalability can only be achieved up to 16 nodes for Ethernet, and moreover, the speedup is not as stable compared to Myrinet. For Class B, because of a larger working set we obtain much better scalability for Ethernet. One interesting note is that MPC++/LAM continues to scale whereas it levels off for C/LAM. There are several potential reasons for this, but they will require further detailed profiling to determine.

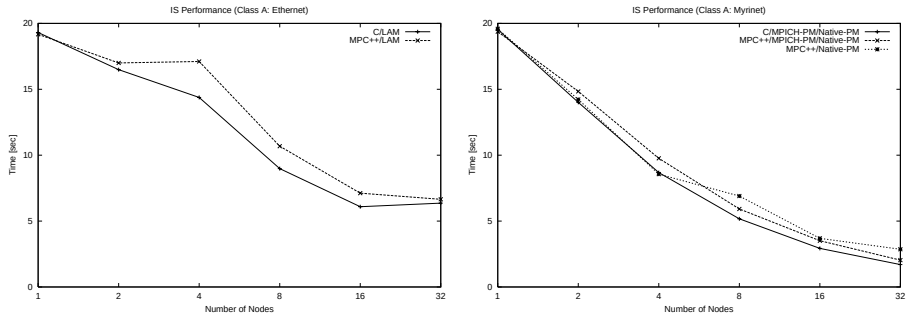

Fig. 4. IS Performance (Class A)

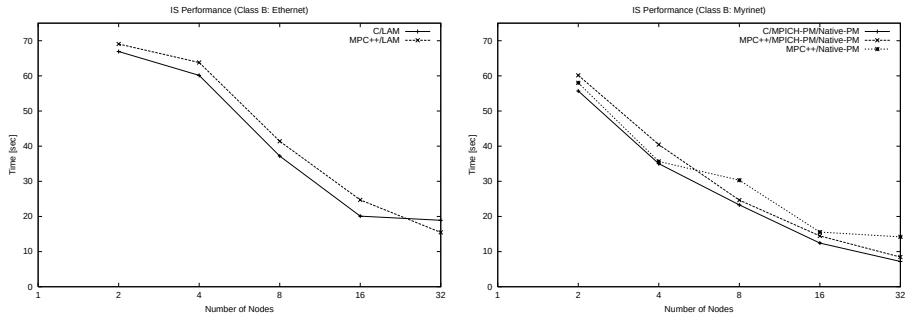

Fig. 5. IS Performance (Class B)

Although scalability was somewhat limited beyond 16 nodes for class A, for (more realistic) class B both benchmarks exhibited competitive performance and scaled well under both 100Base-T and Myrinet. These observations support the viability (albeit somewhat preliminarily) of portable MPC++/MPI implementation to execute well not only on dedicated platforms, but also on everyday commodity platforms.

4.3 Details of CG (Class A)

We analyzed the breakdowns of communication, to investigate the rather subtle performance difference between the original NPB and the MPC++ CG code. Table 2 (C/LAM) and Table 3 (MPC++/LAM) show the results. The numbers indicate the average number of respective MPI operations per node (collective communication is tallied as count of one for each node), "AVG" indicates the average message length per message, and "Total" denotes the average of total number of bytes sent as messages per node. Figure 6 shows the breakdown of time spent on communication/computation for CG for C/LAM and MPC++/LAM. (Possible overlap of communication/computation are not taken into account.)

Table 2. CG (Class A: C/LAM) Breakdown of Communication (per node)

Nodes	2	4	8	16	32
Send	1,200	2,790	5,160	9,090	16,140
Irecv	1,200	2,790	5,160	9,090	16,140
Wait	1,200	2,790	5,160	9,090	16,140
AVG(KB)	18.2	11.7	7.4	4.5	2.6
Total(MB)	20.8	31.3	36.5	39.1	40.4

Table 3. CG (Class A: MPC++/LAM) Breakdown of Communication (per node)

Nodes	2	4	8	16	32
Send	1,591	3,572	6,333	10,654	18,095
Isend	390	1,170	2,730	0	0
Recv	1,981	4,742	9,063	10,654	18,095
Iprobe	1,598	3,575	6,334	10,654	18095
Test	390	1170	2,730	0	0
AVG(KB)	10.80	6.77	4.14	3.87	2.37
Total(MB)	20.9	31.4	36.6	39.4	40.8

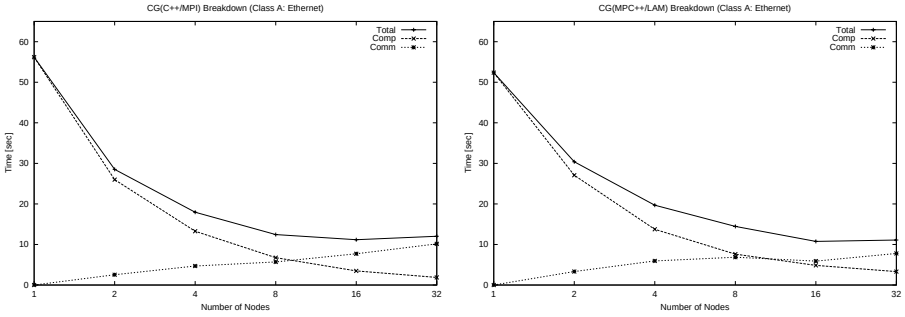

Fig. 6. CG Breakdown (Class A)

In comparison, both send almost equivalent amount of data, but the MPC++/LAM average message size is smaller, resulting in greater number of messages. This is attributable to small control messages, as well as the artifact of converting from C+MPI to MPC++ code. Still, the effect of this is largely negligible as we have seen. As a small note the number of Isend's dropping to 0 is due to the change in communication strategy in MPC++/MPI when message size becomes small.

Figure 6 shows that communication time largely becomes dominant. This is despite that the total size of messages sent per node decreases considerably—rather, the number of communications per node increases almost linearly as the number of nodes increase. Thus, the performance penalty is largely due to the communication latency in this case, for both C++ and MPC++ versions, and the superior low-latency characteristics of PM over Myrinet, not the bandwidth, is likely the cause of superior scalability over 100Base-T Ethernet.

We have performed similar analysis for IS, but we omit the details for brevity. In a nutshell, the MPI operations and the number of messages sent greatly differ between the original IS and the MPC++ version, due to the difference in the collective operations—the original IS uses the MPI collective operations, while the MPC++ version uses its own language directive, which is in turn implemented in terms of point-to-point MPI communication.

5 Conclusion

We have performed detailed analysis of viability of MPC++ on commodity clustering environments. Compared to the original MPC++ which assumed a specialized user-level messaging library PM on top Myrinet, a portable implementation could sit on top of various messaging layers such as MPI, which could in turn might sacrifice performance in various ways. We ported NPB-2.3 CG and IS into MPC++, respecting its natural programming style (e.g., remote read/write through global pointers instead of MPI send/receive), and verified that the efforts involved is quite small, i.e. approximately 60 out of 800 lines for CG and about 50 out of 1000 lines for IS, each taking only a few hours. This is an (albeit indirect) evidence that one could program parallel applications that resemble those in NPB naturally in MPC++. We then performed performance analysis of running on different combinations of applications, the programming language, the underlying software messaging layers, and networking hardwares. The results show that for larger, realistic data sets (class B), the portable implementation of MPC++ on MPI and commodity clustering hardware using 100Base-T Ethernet scales quite competitively to both the original NPB code and the versions running on Myrinet. For smaller data sets, we have performed more detailed analysis to examine the source of the overhead.

Still, many future work remain. First of all, our results are restricted to a few benchmarks. Secondly, we need to perform more finer-grained analysis of scalability, especially the profiling of different parts of code to investigate what usage pattern in the algorithm results in what kind/size/frequency of communication, affecting the overall performance. Next, we need to perform further detailed analysis of the communication characteristics. In particular, scalability beyond 32 nodes, and the effect of use of high-bandwidth, low-latency commodity networking must be investigated. Another interesting endeavor to increase portability while maintaining performance is to employ efficient middle-tier layer such as Nexus and Madeleine, as mentioned earlier. Their implementation will effectively be the middle-ground approach compared to our 'direct' porting of

MPC++ onto commodity clusters, and as such could further clarify where the overhead is for scaling by comparing the results with the current ones.

References

[1] Yutaka Ishikawa. Multi Thread Template Library – MPC++ Version 2.0 Level 0 Document –. Technical Report 012, Tsukuba Research Center, Real World Computing Partnership, September 1996.

[2] Stephan S Luna. Implementing an Efficient Portable Global Memory Layer on Distributed Memory Multiprocessors. Technical report, UCB, May 1994.

[3] L. V. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, September 1993.

[4] Hiroshi Tezuka, Atsushi Hori, Yutaka Ishikawa, and Mitsuhisa Sato. PM: An Operating System Coordinated High Performance Communication Library. In *HPCN '97*, pages 708–717. LNCS, April 1997.

[5] http://www.myri.com/.

[6] Compaq Computer Corp., Intel Corporation, and Microsoft Corporation. *Virtual Interface Architecture Specification*, draft revision 1.0 edition, December 1997.

[7] I. Foster, C. Kesselman, and S. Tuecke. The Nexus approach to integrating multithreading and communication. *Journal on Parallel and Distributed Computing*, pages 37:70–82, 1996.

[8] Luc Boug, Jean-Franois Mhaut, and Raymond Namyst. Madeleine: An efficient and portable communication interface for RPC-based multithreaded environments (revised version). Technical report, LIP, ENS Lyon, December 1999.

[9] K. Mani Chandy and Carl Kesselman. CC++: A Declarative Concurrent Object Oriented Programming Notation. Technical Report CS-92-01, California Institute of Technology, September 1992.

[10] Gregory D. Burns, Raja B. Daoud, and James R. Vaigl. LAM: An Open Cluster Environment for MPI. In *Supercomputing Symposium '94*, June 1994.

[11] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[12] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. THE NAS PARALLEL BENCHMARKS. Technical Report 007, RNR, 1994.

[13] Shinji Sumimoto, Hiroshi Tezuka, Atsushi Hori, Hiroshi Harada, Toshiyuki Takahashi, and Yutaka Ishikawa. High Performance Communication using a Commodity Network for Cluster System. In *HPDC 2000*, August 2000.

[14] Francis O'Carroll, Hiroshi Tezuka, Atsushi Hori, and Yutaka Ishikawa. The Design and Implementation of Zero Copy MPI Using Commodity Hardware with a High Performance Network. In *ACM SIGARCH ICS'98*, pages 243–250, July 1998.