Vol. 47 No. SIG 12(ACS 15) IPSJ Transactions on Advanced Computing Systems Sep. 2006

Regular Paper

Improving Linpack Performance on SMP Clusters with Asynchronous MPI Programming

Ta Quoc Viet† and Tsutomu Yoshinaga†

This study proposes asynchronous MPI, a simple and effective parallel programming model for SMP clusters, to reimplement the High PerformanceLinpack benchmark. The proposed model forces processors of an SMP node to work in different phases, thereby avoiding unnecessary communication and computation bottlenecks. As a result, we can achieve significant improvements in performance with a minimal programming effort. In comparison with a de-facto flat MPI solution, our algorithm can yield a 20.6% performance improvement for a 16-node cluster of Xeon dual-processor SMPs.

1. Introduction

Our study was aimed at improving the performance of the High Performance Linpack (HPL) benchmark [9] on a cluster of Symmetric Multiprocessors (SMPs). HPL is a standard and prestigious tool to evaluate the computation capacity of modern super computing systems. The HPL package applies a flat MPI model that treats all processes equally, without taking the dependency between the computation and communication performance of processes running on physical processors on the same SMP node into account. Our study examines this dependency and proposes an asynchronous MPI programming model, which is proven to be more effective for SMP clusters.

We noted that the performance of an SMP node increases when its processors work asynchronously, i.e., a number of processors performs a computation task while the remainder performs a communication task. This phenomenon can be explained based on limitations with shared resources for computation and/or communication. When all processors work synchronously, they require the same kinds of resources. This may lead to a scarcity of resources, thereby forming an execution bottleneck. Network and memory bus bandwidth limitations are respective root causes of communication and computation bottlenecks in our system.

To overcome this situation, we propose an asynchronous MPI model in which a node's processors perform computation and communication tasks asynchronously. The model requires the execution of communication to be rearranged, which also avoids unnecessary internode communication. The programming effort required to develop our new solution from existing flat MPI code is fairly small. The solution is especially suitable for problems of matrix-calculation problems including the HPL. The basic principles behind our idea and its efficiency on executing a matrix multiplication were also introduced in Ref. 12).

Our experimental environment consisted of a cluster of 16 Intel Xeon 2.8-GHz dual-processor nodes connected via a Gigabit Ethernet network. Each node had 1.5 GB of memory, Red Hat Linux 9.0 was the operating system, and MPICH 1.2.6 [8] was the MPI library. The local matrix calculation functions (e.g., dgemm()) were carried out with the Goto-BLAS library [7].

The significant features of our study were: (1) a clear and easy-to-apply MPI-only asynchronous parallel-programming model for SMP clusters and (2) a formula for evaluating its improvement in performance using variables defined for the nature of the problem, its size, and system specifications.

The remainder of the paper is organized as follows. Section 2 introduces related studies that have also examined programming models for SMP clusters. Section 3 presents our model in details. Section 4 describes the process of applying the model to the HPL problem. Section 5 discusses the experimental results, and Section 6 concludes the paper.

† Graduate School of Information Systems, University of Electro-Communications

2. Related Studies

2.1 Communication-Computation Overlap

The main idea behinde the asynchronous model is the node-level communication-computation overlap (referred to as "overlap" from now). The proceeses inside a node are forced to simultaneously execute computation and communication tasks. In fact, another type of overlap, the global-level, has long been proposed and discussed [1),3)], where a collective communication function is divided into several stages, and in each stage only a group of processes of the communicator is involved. The remaining processes can execute computation tasks at that time. As a result, some processes at a point in time are involved in communication while the others are in computation phases. However, other than for the node-level type, there is no guarantee of overlap inside a node. For example, we can enable global-level overlap with the original HPL package by assigning a positive value for parameter $DEPTH$ defined inside its data file, $HPL.dat$.

2.2 Hybrid Programming Models

Several studies on SMP clusters, whose specificity is the hierarchical memory architecture, have proposed hybrid MPI-OpenMP programming models in which each SMP node only runs a single MPI process and parallelization of computation inside a node is deployed by OpenMP. In comparison with flat MPI, hybrid models not only replace internode communication by using shared variables located in shared memory areas but they also reduce the number of MPI processes.

Hybrid models are classified in terms of process-to-process communication (hybrid PC) and thread-to-thread communication (hybrid TC). The hybrid PC model was examined earlier but no positive results were obtained. Cappello et al. found a common path to develop a fine-grained hybrid PC code from an existing MPI model [5)]. Based on this path, they derived a fine-grained hybrid PC solution to the NAS benchmarks and compared its performance with that of a flat MPI model for a cluster of IBM SP nodes [4),6)]. Using COSMO, a cluster of Intel dual-processor nodes, Boku, et al. compared hybrid PC with flat MPI by solving the smooth particle applied mechanics (SPAM) problem [2)]. These studies revealed that in most of the cases, hybrid PC is inferior to flat MPI despite its three main advantages of (1) low communication costs, (2) dynamic load balancing capabilities, and (3) coarse-grained communication capabilities [2)]. The poor performance of the fine-grained hybrid PC model is primarily due to its poor efficiency in intra-node OpenMP parallelization [4)] resulting from an extremely low cache hit ratio [2)].

In previous studies, we proposed an enhanced hybrid version–the hybrid TC model [15),16)], which was also discussed by Wellein, et al. [17)] and Rabenseifner, et al. [10),11)]. We also proposed a medium-grained approach that allowed hybrid TC to achieve impressive performance on different platforms in various types of experiments [14)]. The essence of hybrid TC is also node-level overlap. However, to gain that level of performance, it requires huge programming efforts with complicated task assignation techniques. Moreover, hybrid TC suffers from a critical problem with extra communication costs with the prime numbers of SMP nodes due to an unbalanced process grid. In these cases, it even loses out to flat MPI in performance [13)].

The asynchronous MPI model we propose in this paper aims at overcoming these disadvantages of hybrid TC. While yielding high performance, it is still easy to implement and does not suffer from the problem of an unbalanced process grid.

3. Asynchronous Model

3.1 Flat and Asynchronous MPI

Figure 1 outlines the activities of a dual-processor SMP node for flat and asynchronous MPI solutions.

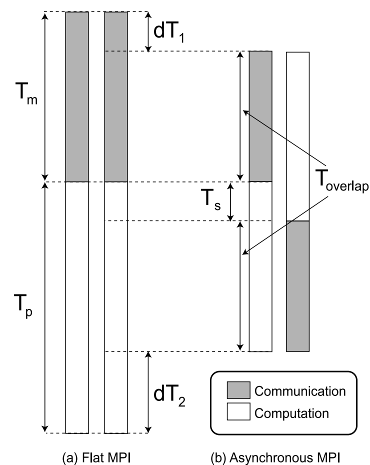


Fig. 1 MPI variations for a dual-processor SMP node.

All the processors of a node work in the same phase for the flat MPI solution. They always execute communication or computation simultaneously. The execution time T_{flat} is the sum of the communication time T_m and the computation time T_p, which usually depends on the size of the problem:

$$T_{flat} = T_m + T_p.$$

Node-level overlap is created for the asynchronous MPI solution. The time in which node's processors work asynchronously is denoted by $T_{overlap}$. The time in which processors work simultaneously is denoted by T_s. The sum of $T_{overlap}$ and T_s yields the asynchronous model execution time T_{async}:

$$T_{async} = T_{overlap} + T_s.$$

Communication speed-up S_m is defined by the ratio between communication speeds during $T_{overlap}$ and T_s. Computation speed-up S_p is defined similarly. Note that communication and computation speeds during T_s are equivelent to that of the flat MPI model. During $T_{overlap}$, an SMP node has better communication and computation speeds than that of the flat MPI model. In other words,

$$S_m > 1 \quad \mathit{and} \quad S_p > 1.$$

We now evaluate the benefits of the asynchronous model. With regard to Fig. 1,

$$T_{overlap} = 2 \times \min\left(\frac{T_m}{S_m}, \frac{T_p}{S_p}\right), \qquad (1)$$

and the difference in the execution times dT between the two models is given by

$$dT = T_{flat} - T_{async} = dT_1 + dT_2.$$

On the other hand,

$$dT_1 = \frac{1}{2} \times T_{overlap}(S_m - 1)$$

and

$$dT_2 = \frac{1}{2} \times T_{overlap}(S_p - 1).$$

Consequently,

$$dT = \frac{1}{2} \times T_{overlap}(S_m + S_p - 2). \qquad (2)$$

The performance speed-up, S, of the model is evaluated by

$$S = \frac{T_{flat}}{T_{async}} = 1 + \frac{1}{2} \times \frac{T_{overlap} \times (S_m + S_p - 2)}{T_{overlap} + T_s}. \qquad (3)$$

Equations (1), (2), and (3) define the time saved by and the speed-up of an asynchronous solution through T_m, T_p, S_m, and S_p.

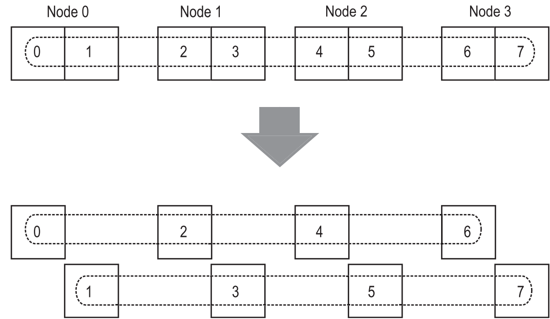


Fig. 2 Communication rearrangement.

T_m and T_p generally increase with the size of the problem, which also increases dT. This implies that as the size of the problem increases, the time saved by the asynchronous model also increases.

Speed-up S achieves a maximum value of

$$S = \frac{(S_m + S_p)}{2}$$

if

$$T_s = 0,$$

i.e.,

$$\frac{T_m}{S_m} = \frac{T_p}{S_p}.$$

3.2 Communication Rearrangement

The asynchronous model requires a communication to be rearranged, where internode communication can only be performed between processes with the same phase. The original flat MPI communication pattern does not always satisfy this requirement.

We propose a simple method of achieving the required rearrangement, which is outlined in **Fig. 2**, where the squares represent MPI processes, the numbers inside these squares denote the process IDs, and the dotted elongations indicate communicators. The upper and lower parts of the figure portray the communicators before and after rearrangement. We assumed a flat MPI communicator with eight processes originating from four dual-processor nodes. This communicator is split into two separate internode parity communicators, odd and even, in terms of the process IDs. There were four additional intranode communicators, which are not shown in the figure. All communication operations were rearranged to only be performed inside the newly formed communicators. An internode communication operation between two processes of different parity communicators is replaced by an intranode op-

eration and an internode operation inside the same parity communicator. For example, communication between processes 2 and 7 may be replaced by that between processes 2 and 3 (intranode communication) together with that between processes 3 and 7 (internode communication inside the odd communicator).

3.3 Task Dependency Problem

Data independence between communication and computation tasks is another requirement for the asynchronous model. We can only arrange tasks in any order if they are data independent. Section 3 of Ref. 14) has already demonstrated how to eliminate these dependencies. We will explain the process of eliminating dependencies for HPL in Section 4.3,

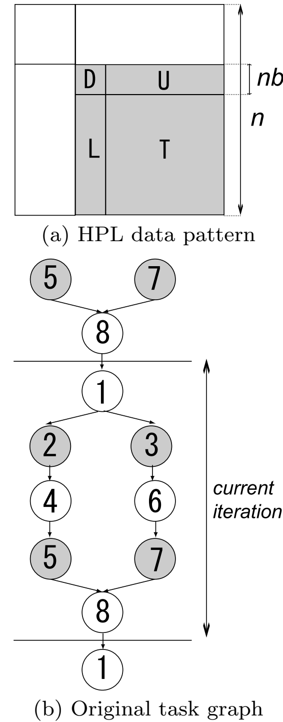
4. Asynchronous Solution for HPL

4.1 Problem Description

HPL solves a random dense linear equation system using a block LU decomposition algorithm. Its major task is to factorize $n \times n$ random dense square coefficient matrix A into corresponding *upper* and *lower* triangulars, U and L, such that $A = U \cdot L$. When n is sufficiently large, this factorization consumes more than 99% of the overall execution time. Users can set the problem size, n, upon execution. HPL accepts any value for the *nprocs* processes, which are organized into a $P \times Q$ process grid. The multiplication between dense matrices involves most of the computation cost.

The data in HPL are stored in $nb \times nb$ square blocks, where nb is the block size and can be adjusted on execution to obtain the best performance. Blocks are distributed onto *nprocs* processes according to a *block-cyclic* scheme, i.e., they are cyclically dealt onto the $P \times Q$ process grid. Such a data distribution assists in decreasing communication costs [9].

According to the right looking variant, LU factorization is done by a loop with $\lceil n/nb \rceil$ iterations. Data related to the ith iteration are outlined in **Fig. 3** (a). D is the ith block of the main diagonal. L, U, and T are the current parts of the lower, upper, and trailing matrices, respectively. **Table 1** lists the tasks for such an iteration. Tasks 2, 3, 5, and 7 are communication tasks. Task 1 depends on task 8 of the previous iteration. Symbols U^0 and L^0 for tasks 4 and 6 represent the current values of U and L, respectively. Each of these should be replaced by the root of the correlative equation after the task finishes. Task 8 involves major computation. Meanwhile, tasks 5 and 7 occupy almost all the communication volume. A task dependency graph is created based on Table 1, shown in Fig. 3 (b), where communication tasks are represented by the shaded circles.



(a) HPL data pattern

(b) Original task graph

Fig. 3 HPL: data pattern and original task dependency graph.

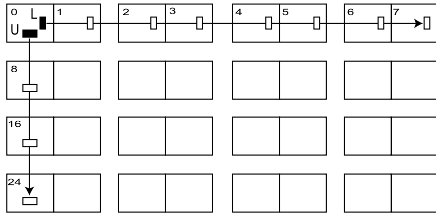
Table 1 HPL: original task list.

No.	Description	Cur. dep.	Prev. dep.
1	Decom(D)	-	8
2*	Bcast(D, *col_comm*)	1	-
3*	Bcast(D, *row_comm*)	1	-
4	Solve($DU = U^0$)	2	-
5*	Bcast(U, *col_comm*)	4	-
6	Solve($LD = L^0$)	3	-
7*	Bcast(L, *row_comm*)	6	-
8	$T = T - LU$	5, 7	-

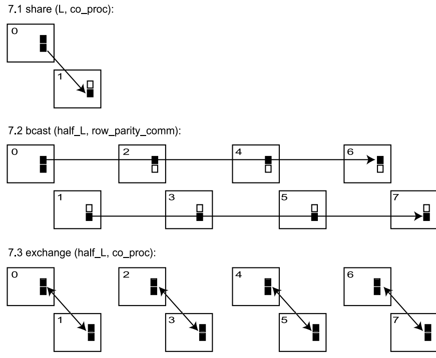
*: Communication tasks
Cur. dep.: Current iteration's dependencies
Prev. dep.: Previous iteration's dependencies
Row_comm: Process row communicator
Col_comm: Process col communicator

4.2 Process Grid and Communication Rearrangement

The process grid is organized to minimizes the amount of rearrangement. For example, processors can be arranged such that all processors of a single node belong to the same process row. In this way we developed a 4×8 process grid for a 16-node cluster, in which a row

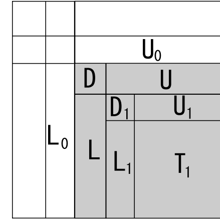


(a) Broadcast operations within flat MPI

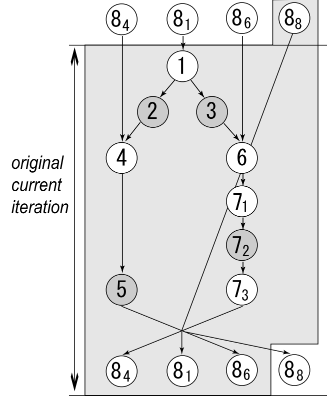


(b) Rearrangement for bcast (a_i, row_comm)

Fig. 4 Communication rearrangement for a 16-node dual-processor cluster.



(a) New HPL data pattern



(b) New HPL task graph

Fig. 5 Construction of asynchronous task schedules.

communicator contains processes of the same parity; thus, it is not necessary to rearrange bcast(D, col_comm) and bcast(U, col_comm). Only broadcasting over the row_comm needs to be rearranged.

The rearrangement in communication is outlined in **Fig. 4** assuming that processor 0 is the source of U and L. Figure 4 (a) illustrates both of the two broadcast operations within flat MPI. Figure 4 (b) shows the three operations that have replaced the original task 7 bcast(L, row_comm):

- 7_1 share($half_L, co_proc$): intranode communication. In reality, this step is implemented by a set of *send* and *recv* calls evoked by the source process and *co_proc*– the remaining node process.
- 7_2 bcast($half_L, row_parity_comm$): internode communication. Even and odd row communicators broadcast different halves of L. After this step, every process has half the L, while two processes of the same node have different halves.
- 7_3 exchange($half_L, co_proc$): intranode communication. Each process exchanges its half with its *co_proc*. After this step, all processes have a complete L.

Internode communication step 7_2 can be executed within the asychronous execution time. The intranode communication steps should be executed simultaneously by all processes.

Similarly, task 3, bcast(D, row_comm), can be replaced by new tasks 3_1, 3_2, and 3_3. However, for a sufficiently large problem, the cost of executing task 3 is quite small in comparision with that for task 5 or 7, and thereby not worth rearranging. As a result, we did not rearrange task 3 in our implementation.

4.3 Asynchronous Task Schedules

We eliminated the data dependencies beween the main computation and communication tasks to enable asynchronous sections. As can be seen from Fig. 3 (b), all tasks of the current iteration depend on task 8 of the previous iteration that updates the "previous" trailing matrix including the "current" D, U, L, and T. To break up this dependency into several smaller ones, we split task 8 into 8_1, 8_4, 8_6, and 8_8 that updated D, U, L, and T, respectively. We then have a new task dependency graph shown in **Fig. 5** (b). Tasks 1, 4, 6, and 8 now depend on the previous tasks 8_1, 8_4, 8_6, and 8_8, respectively. Communication task 7 is now rearranged and replaced by 7_1, 7_2, and 7_3 in the figure. Figure 5 (a) illustrates the corresponding data pattern.

Since tasks 5 and 7_2 (major communication) and the previous task 8_8 (major computation) are data independent, we reconstructed the

Table 2 Updated task list for asynchronous MPI.

No.	Description	Dependencies
1	Decom(D)	-
2	Bcast(D, col_comm)	1
3	Bcast(D, row_comm)	1
4	Solve$(DU = U^0)$	2
5*	Bcast(U, col_comm)	4
6	Solve$(LD = L^0)$	3
7_1	Share$(haft_L, co_proc)$	6
7_2*	Bcast$(haft_L, row_parity_comm)$	7_1
7_3	Exchange$(haft_L, co_proc)$	7_2
8_1	$D_1 = D_1 - LU$	5, 7_3, 8_8
8_4	$U_1 = U_1 - LU$	5, 7_3, 8_8
8_6	$L_1 = L_1 - LU$	5, 7_3, 8_8
8_8*	$T_1 = T_1 - LU$	-

*: Tasks placed inside asynchronous section

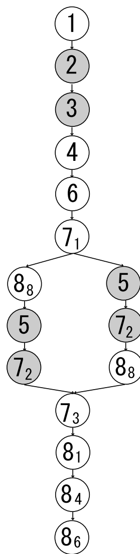


Fig. 6 Asynchronous task schedules.

loop such that they were in the same iteration and become available for the asynchronous phase. In Fig. 5 (a), the tasks for a new loop iteration are bounded by the shaded polygon. This includes tasks belonging to the two original iterations: task 8_8 from the previous and the remaining tasks from the current iteration. The updated task list is shown in **Table 2** wherein tasks that have been planned to overlap have been marked with an asterisk.

Figure 6 outlines the new task schedules to be applied to asynchronous MPI. Tasks from 1 to 7_1 are executed simultaneously by all processes. The even and odd processes then follow the left and right branches of the diagram, respectively. That is, even processes perform computation first with task 8_8, then carry out communication with tasks 5 and 7_2. Odd processes run in a reverse order: tasks 5 and 7_2, then 8_8. After that, all processes work together once again with tasks from 7_3 to 8_6. **Figure 7** shows the pseudo-code that strictly follows these task schedules. The code is relatively simple and easy to implement.

```
...
MPI_Comm_rank(MPI_COMM_WORLD, &pid);
Loop Initialize();
for (i=0; i < number of iterations; i++)
{
  task_1();
  task_2();
  task_3();
  task_4();
  task_6();
  task_7_1();
  // Asynchronous section
  if (pid%2==0) {
    task_8_8();
    task_5();
    task_7_2();
  }
  else {
    task_5();
    task_7_2();
    task_8_8();
  }
  // End of asynchronous section
  task_7_3();
  task_8_1();
  task_8_4();
  task_8_6();
}
...
```

Fig. 7 Asynchronous MPI pseudo-code.

Since tasks 5, 7_2, and 8_8 consume the major computation and communication costs for the problem, node-level overlap is archieved during most of the execution time.

5. Experimental Results

The communication speed-up, S_m, and computation speed-up, S_p, of the overlaping sections in the asynchronous model over the flat MPI model can be estimated by implementing some simple simulations. Their values generally change under different computation and communication conditions. **Figure 8** shows a pseudo-code for a simulation that defines S_m for HPL, wherein communcation always occurs with large messages and major computation is carried out by BLAS level 3 function dgemm(). While odd processes do a typical HPL communication task (broadcasting a large amount of data over a process column), even processes perform a typical HPL computation task (executing a local dgemm() function). The computation task is set so that it is large enough to cover the entire communication time. Then, the

```
...
// Simulating overlap condition, Tm < Tp.
if (pid%2==0)
  large_HPL_typical_comp_task();
else {
  t_0 = MPI_Wtime();
  HPL_typical_comm_task();
  t_overlap = MPI_Wtime() - t_0;
}
// Simulating flat MPI condition.
t_0 = MPI_Wtime();
HPL_typical_comm_task();
t_flat = MPI_Wtime() - t_0;
// Evaluating Sm.
Sm = t_flat / t_overlap;
```

Fig. 8 Simple simulation to define S_m for HPL.

communication pattern for the flat MPI model is simulated. The ratio between the communication times for flat MPI and the overlap stages to solve the same communication task yields S_m. The value of S_p can be defined in a similar way, where the communication task is set so that it is large enough to cover the computation time.

Broadcasting a $112 \times 3{,}000$ matrix over a process row was used as a communication task in our simulation, and multiplication between a $3{,}000 \times 112$ and a $112 \times 3{,}000$ matrix was used as a computation task. By running this computation (communication) task several times, we could create a computation (communcation) task large enough to cover the communication (computation) time to estimate S_m (S_p). Under these conditions, we obtained

$$S_m \approx 1.32 \quad and \quad S_p \approx 1.18.$$

Further simulations we did with different data sizes also revealed that, when the matrix size was large enough, the values for S_m and S_p were relatively stable.

The experimental results for flat MPI, hybrid TC, and asynchronous MPI on all 16 nodes corresponding to different-sized problems are plotted in **Fig. 9**. Matrix size n varies between 20,000 and 50,000, which is sufficiently large to stabilize the local Goto-BLAS functions and small enough to accommodate the memory limit. The block size, nb, was fixed to 112.

The results clearly prove the asynchronous MPI outperformed flat MPI. At n = 20,000, the difference was approximately 20.6% (13.2 GFlops). However, the distance between the two performance lines gradually reduced. At n = 50,000, the difference was only 8.6% (7.9 GFlops). These results can be explained by Eqs. (2) and (3) together with the nature of the

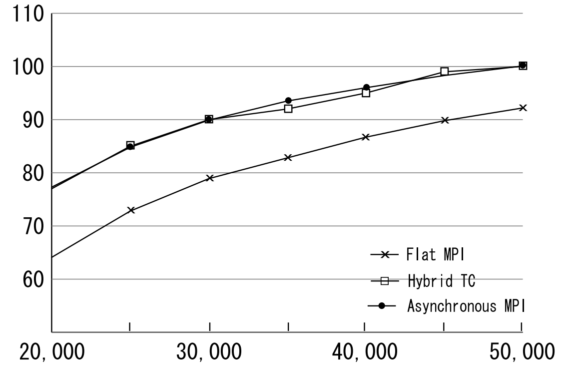


Fig. 9 Performance with 16 nodes, GFlops by problem size, n.

Table 3 Theoretic and actual time saved, dT_t and dT_a.

n	T_m (sec.)	T_p (sec.)	dT_t (sec.)	dT_a (sec.)
20,000	37.4	45.1	14.2	13.6
25,000	53.7	88.9	20.3	19.7
30,000	73.6	153.9	27.9	27.0
35,000	99.8	244.8	37.8	36.4
40,000	128.5	372.9	48.7	47.2
45,000	155.0	519.6	58.7	57.2
50,000	188.4	713.8	71.4	69.4

HPL problem. That is, the growth rate of T_p is $O(n^3)$, while that of T_m is only $O(n^2)$. Consequently, the growth rates of dT and T_{async} are $O(n^2)$ and $O(n^3)$, respectively. This implies that the increase in dT is slower than that in T_{async}, and the overall speed-up, S, becomes smaller with an increase in n.

The difference in execution time of flat MPI and asynchronous MPI, on the other hand, increases with the size of the problem. **Table 3** lists the theoretic and actual values for the reduced time, dT_t and dT_a, respectively. The theoretic values were evaluated with Eq. (2). The actual values are in good agreement with these, although there is a small disparity of several percent that may result from (1) various measurement errors and (2) changes in execution conditions, i.e., the actual values for S_m and S_p are not completely stable with different-sized problems, while the theorical values are defined by simulations with a certain fixed problem size. This small disparity may also reslult from (3) the influence of the communication rearrangements, i.e., althought rearrangement avoids unnecessary internode communication, it also cuts the communication data size to half while broadcasting L over the process row and this may slow down the communciation speed to some degree. Although the above-mentioned

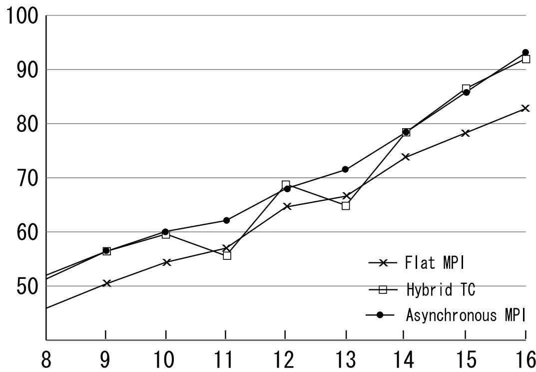


Fig. 10 Performance with $n = 35{,}000$, GFlops by number of nodes.

factors do affect overall performance, their influence is relatively small, and we assume that, they are not worth further analysis in more details.

The performance lines for asynchronous MPI and hybrid TC nearly coincide, which can be explained by the similar way they perform computation and communcation. Although they are completely different in their implementation, both of them aim at exploiting the advantages of node-level communication-computation overlap. However, the programming effort needed by hybrid TC is much greater than that by the asynchronous model. As discussed in Section 2, apart from complicated task assignation, hybrid TC requires the knowledge and skill of both MPI and OpenMP programming. Meanwhile, asynchronous MPI retains the original flat MPI computation pattern and does not require OpenMP support.

In addition to simplicity, asynchronous MPI, in some cases, also surpasses hybrid TC in performance. **Figure 10** plots HPL performance with a fixed value of $n = 35{,}000$ corresponding to different numbers of nodes. Due to the problem with the unbalanced process grid, hybrid TC obtains extremely poor performance with prime numbers of nodes. Running 11 or 13 nodes, hybrid TC is even slower than flat MPI. Its process grid is then 1×11 or 1×13, which critically increases the communication cost [14]. This degradation does not happen with asynchronous MPI. It can always organize a less unbalanced process grid (2×11 or 2×13), thereby obtaining a similarly shaped performance line to that of flat MPI.

6. Conclusions and Discussions

The proposed asynchronous MPI model is a simple and effective parallel programming approach. We demonstrated its advantages through solving the HPL problem on a cluster of dual SMP nodes. Asynchronous MPI significantly outperformed the traditional and widely accepted flat MPI model. Apart from saving a significant amount of programming effort, asynchronous MPI also solved the problem of an unbalanced process grid within prime numbers of SMP nodes compared with a strong but complicated hybrid TC model.

The formulas for evaluating the increase in performance were also remarkable. They were used to evaluate the benefits of the asynchronous model without implementing the entire solution. They also proved its scalability.

For problems where the computation and communication costs have different growth rates (e.g., HPL), the increase in performance decreases with large problems. For problems of different sizes with constant computation and communication cost ratios (e.g., NAS-CG), we expect steady improvement with the asynchronous model.

Although the analyses and experiments in this paper were mainly for a dual-processor cluster, the conclusions can be expanded to other clusters with more processors per node. In these cases, it would be necessary to determine an effective number of phases and corresponding task schedules.

We would like to verify the efficiency of our novel model on different hardware platforms in future studies, including clusters of Intel dual-core or AMD Opteron multiprocessors, with different types of interconnection networks. In addition, we plan to build an asynchronous MPI library that includes major matrix calculation functions for an SMP cluster environment.

Acknowledgments This research was supported in part by Grants-in-Aid for Scientific Research made available by the Japan Society for the Promotion of Science (JSPS) (Nos.15500033(C) and 17360178(B)), and a Sasakawa Scientific Research Grant made by the Japan Science Society (JSS) (No.17-251).

References

1) Baden, S.B. and Fink, S.J.: Communication Overlap in Nulti-tier Parallel Algorithms, *Proc. 1998 ACM/IEEE Conference on Supercomputing*, pp.1–20, San Jose, US (1998).

2) Boku, T., Yoshikawa, S. and Sato, M.: Implementation and Performance evaluation of

SPAM article code with OpenMP-MPI hybrid programming, *Proc. European Workshop on OpenMP 2001* (2001).

3) Brightwell, R. and Underwood, K.D.: An Analysis of the Impact of MPI Overlap and Independent Progress, *Proc.18th Annual International Conference on Supercomputing*, pp.298–305, France (2004).

4) Cappello, F. and Etiemble, D.: MPI versus MPI+OpenMP on IBM SP for the NAS Benchmark, *Proc. Supercomputing 2000* (2000).

5) Cappello, F. and Richard, O.: Intra Node Parallelization of MPI Programs with OpenMP, Technical Report TR-CAP-9901, http://www.lri.fr/~fci/goinfreWWW/1196.ps.gz (1998).

6) Cappello, F., Richard, O. and Etiemble, D.: Investigating the Performance of Two Programming Models for Clusters of SMP PCs, *Proc. High Performance Computer Architecture*, pp.349–359 (2000).

7) Goto, K.: High-Performance BLAS by Kazushige Goto. http://www.cs.utexas.edu/users/flame/goto/

8) MPICH Team: MPICH, a Portable MPI Implementation. http://www-unix.mcs.anl.gov/mpi/mpich/

9) Petitet, A., Whaley, R.C., Dongarra, J. and Cleary, A.: HPL—A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers. http://www.netlib.org/benchmark/hpl/

10) Rabenseifner, R.: Hybrid Parallel Programming: Performance Problems and Chances, *Proc. 45th CUG* (*Cray User Group*) *Conference 2003* (2003).

11) Rabenseifner, R. and Wellein, G.: Communication and Optimization Aspects of Parallel Programming Models on Hybrid Architectures, *The International Journal of High Performance Computing Application*, Vol.17, No.1 (2003).

12) Viet, T.Q. and Yoshinaga, T.: Asynchronous Parallel Programming Model for SMP Clusters, *Proc. 17th IASTED International Conference on Parallel and Distributed Computing and Systems* (*PDCS2005*) Phoenix, US (2005).

13) Viet, T.Q., Yoshinaga, T. and Abderazek, B.A.: Performance Enhancement for Matrix Multiplication on an SMP PC Cluster, *IPSG SIG notes 2005-HPC-103*, pp.115–120 (2005).

14) Viet, T.Q., Yoshinaga, T., Abderazek, B.A. and Sowa, M.: Construction of Hybrid MPI-OpenMP Solutions for SMP Clusters, *IPSJ Transactions on Advanced Computing Systems*, Vol.46, No.SIG 3(ACS 8), pp.25–37 (2005).

15) Viet, T.Q., Yoshinaga, T., Abderazek, B.A. and Sowa, M.: A Hybrid MPI-OpenMP Solution for a Linear System on a Cluster of SMPs, *Proc. Symposium on Advanced Computing Systems and Infrastructures*, pp.299–306 (2003).

16) Viet, T.Q., Yoshinaga, T. and Sowa, M.: A Master-Slaver Algorithm for Hybrid MPI-OpenMP Programming on a Cluster of SMPs, *IPSJ SIG notes 2002-HPC-91-19*, pp.107–112 (2002).

17) Wellein, G., Hager, G., Basermann, A. and Fehske, H.: Fast sparse matrix-vector muliplication for TeraFlop/s computers, *Proc. Vector and Parallel Processing* (2002).

(Received January 27, 2006)
(Accepted May 16, 2006)

Ta Quoc Viet received his M.E. degree from Graduate School of Information Systems, University of Electro-Communications (UEC) in 2004 and is currently a Ph.D. student. His research interests include high performance computing, cluster computing, and parallel programming models. He is a member of IEEE.

Tsutomu Yoshinaga received his B.E., M.E., and D.E. degrees from Utsunomiya University in 1986, 1988, and 1997, respectively. From 1988 to July 2000, he was a research associate of Faculty of Engineering, Utsunomiya University. He was also a visiting researcher at Electro-Technical Laboratory from 1997 to 1998. Since August 2000, he has been with the Graduate School of Information Systems, UEC. His research interests include interconnection networks for MPPs, cluster computing, and P2P networks. He is a member of IEEE and IEICE.