Regular Paper

Construction of Hybrid MPI-OpenMP Solutions for SMP Clusters

Ta Quoc Viet,† Tsutomu Yoshinaga,† Ben A. Abderazek† and Masahiro Sowa†

This paper proposes a middle-grain approach to construct hybrid MPI-OpenMP solutions for SMP clusters from an existing MPI algorithm. Experiments on different cluster platforms show that our solutions exceed the solutions that are based on the de-facto MPI model in most cases, and occasionally by as much as 40% of performance. We also prove an automatic outperformance of a thread-to-thread communication model over a traditional process-to-process communication model in hybrid solutions. In addition, the paper performs a detailed analysis on the hardware and software factors affecting the performance of MPI in comparison to hybrid models.

1. Introduction

Clusters of Symmetric Multiprocessors (SMPs) have recently gained great popularity. Consequently, the proper choice of a parallel programming model becomes extremely important. The three model candidates are listed as follows:

1. *Pure MPI* (*MPI*): each processing element (PE) is used for one MPI process, which has its own address space [1]. A PE communicates with others by passing messages.
2. *Hybrid MPI-OpenMP with process-to-process communication* (*Hybrid PC*): each SMP node is used for one MPI process. OpenMP [2] is applied for computation parallelization within SMP nodes. MPI communication is executed outside OpenMP parallel regions.
3. *Hybrid MPI-OpenMP with thread-to-thread communication* (*Hybrid TC*): similar to the hybrid PC model. However, MPI communication tasks are performed within OpenMP parallel regions by a single thread. During communication, non-communicating threads are assigned to perform computation tasks. A computation-communication overlap within a node is the particularity of the model.

Thus, the number of processes in MPI, $nprocs_{(M)}$, is the product of the number of nodes, *nnodes*, and the number of PEs per node, *nppn*. This number is equal to *nnodes* for hybrid models:

$nprocs_{(M)} = nnodes \times nppn$

$nprocs_{(PC)} = nprocs_{(TC)} = nnodes$

Until date, hybrid models apply two common methods to parallelize computation tasks with OpenMP within a node: (1) fine-grain loop-level parallelization and (2) coarse-grain SPMD parallelization. In the former, OpenMP parallel directives are inserted into all available parallel loops. In the latter, each thread manages its own data and performs computation tasks as if it is a process in MPI with no further utility of the *"!$Omp do"* construct [3].

This paper proposes and applies an alternate, the *middle-grain* approach. While avoiding the complexity of the SPMD method, it also denies a poor performance characteristic of the fine-grain parallelization (Section 3.1). Applying the middle-grain method along with an *overlapping-oriented* task-schedule (Section 3.3), we can create effective hybrid TC solutions that exceed MPI in performance, occasionally by as much as 40%.

To achieve multiformity, we choose experimental platforms and problems with contrary characteristics. A cluster of Sun Enterprise 3500 SMPs and a cluster of Intel Dual-processor SMPs are used as experimental platforms (Section 5). The NAS-CG benchmark—CG [4] (Section 6) and the High Performance Linpack Benchmark—HPL [5] (Section 7) are selected as experimental problems. Both benchmarks are important and well-known in the high performance computing world and are therefore, examined by various studies [3),6)~9)].

The main contributions of this paper are listed as follows:

1. *A detailed comparison among the three models, and a complete analysis on the hardware and software factors that affect their performance.*
2. *A proof for automatic outperformance of hy-*

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

brid TC over hybrid PC under any circumstances.

3. *An algorithm to construct a hybrid TC solution based on the middle-grain approach and an overlapping-oriented task-schedule, which dominates the de-facto MPI solution in most cases.*

2. Related Works

The hybrid PC model has been examined in several prior studies. In Ref. 6), the authors have shown a common path to construct a fine-grain hybrid PC code, which is referred to as the "Hybrid Memory Model—HMM," from an existing MPI model. On the basis of this path, the authors of Refs. 7) and 8) built a fine-grain hybrid PC solution for the NAS benchmarks and compared its performance to a pure MPI model on a cluster of IBM SP nodes. Using COSMO—a cluster of Intel dual processor nodes, the authors of Ref. 10) compared hybrid PC and pure MPI by solving the Smooth Particle Applied Mechanics (SPAM) problem. The above mentioned studies revealed that hybrid PC is worse than pure MPI in most cases despite its three main advantages: (1) low communication cost, (2) dynamic load balancing availability, and (3) coarse-grain communication availability [10]. Even on the Earth Simulator with the CG problem, hybrid PC outperforms pure MPI only when *nnodes* is a considerably large [11].

The poor performance of the fine-grain hybrid PC model was explained by its poor inner-node OpenMP parallelization efficiency [8], which was due to a bad cache hit ratio [10]. These studies also discussed replacing the fine-grain loop-level OpenMP parallelization by the coarse-grain SPMD OpenMP model with extra thread data localization. However, they also predicted two huge disadvantages for such a solution: (1) complexity in programming and (2) sacrifice of the dynamic load balancing availability.

Another study has described the construction of a coarse-grain SPMD *pure OpenMP* solution on a shared-memory platform [3]. This solution outperforms a pure MPI solution in all experiments. The results proved its ability to obtain an inner-node OpenMP performance better than that of the MPI model. However, the application of the approach to hybrid solutions is not yet clear.

We initially proposed the basic principles of hybrid TC in Refs. 12) and 13). In Ref. 13), a hybrid TC solution for the HPL benchmark outperformed MPI by approximately 10% on the Sun cluster although it was created by a relatively primitive algorithm. However, the lack of experiments did not permit us to arrive to a conclusion regarding the dominance of hybrid TC. The performance for hybrid TC will improve by approximately 27.5% by proposing and applying a more progressive algorithm as shown in this paper.

Hybrid TC was then discussed and compared with the hybrid PC of Refs. 14), 15), and 16). According to the authors, there is no automatic outperformance for hybrid TC. In Section 4.4, we can prove that hybrid TC always exceeds hybrid PC in performance.

3. Hybrid Solution Construction

This section presents a new method to create hybrid solutions from original MPI algorithms. The hybrid TC solution is based on the middle-grain approach and an overlapping-oriented task-schedule. The hybrid PC solution applies only the middle-grain approach along with an original MPI task-schedule.

3.1 The Middle-grain Approach

Until date, there are two common approaches for OpenMP to parallelize computation tasks within an SMP node: fine-grain and coarse-grain SPMD methods. The former simply inserts OpenMP *"fork-and-join"* directives in all available loops from an original MPI code. However, it exhibits a rather poor performance in comparison to the MPI model. The latter treats the threads like MPI processes in the MPI model. The data is thread-localized. A single OpenMP *"fork-and-join"* construct is applied to the whole solution. The *"!$Omp do"* directive is no longer used. The communication part from the original MPI solution is also simulated and replaced by memory copying. Within an SMP, the application of the approach results in a brilliant performance that outperforms the original MPI model in all the examined experiments [3]. However, this approach lacks simplicity, a particularity of OpenMP. Furthermore, the application of the approach to hybrid solutions is not yet clear.

In this study, we propose a novel *middle-grain* approach that has advantages over both the above mentioned methods: simplicity and effectiveness. Its main features are listed as follows:

MPI-algorithm basement: the solution is based on the original MPI algorithm rather

than its code.

Middle-grain parallelization: an OpenMP parallel construct is applied for a *block*, which may contain several loops. A block includes both computation and communication tasks. Computation tasks are partitioned into indexed grains that will be executed by the *"!$Omp do"* directives.

Intra-node flag communication: all data-dependencies are followed by status flags. A thread must update the shared status flag to announce the completion of a task. In order to confirm the completion of the task, other threads can check the flag's status.

3.2 The Middle-grain's Effect

Until date, a *traditional* hybrid model with the fine-grain approach is assumed to be less effective than MPI in computation. The poor performance is explained by (1) extra OpenMP synchronization overheads caused by too many parallel regions and (2) a low cache hit ratio caused by bad memory access behavior. However, by the middle-grain approach, hybrid models can overcome both the above obstacles and achieve the same performance as MPI:

(1) *OpenMP synchronization overheads:*

The middle-grain approach aids in significantly decreasing the number of OpenMP parallel regions, which makes the synchronization cost become negligible.

(2) *Memory access behavior:*

Without data localization, a hybrid model may force a thread to manipulate the entire data distributed to a node, and the cache size becomes insufficient [10]. Due to an MPI-algorithm basement, the middle-grain approach is flexible enough to split a task into grains such that the necessary data of which are separated, and a certain thread will have to manipulate a smaller data area. In any case, a programmer can simulate the memory usage pattern of the MPI model and obtain an identical cache effect.

In addition, OpenMP may cause concurrent access by several threads to the same memory location. In order to avoid this problem, certain data with high rate of concurrent access are declared as *private* variables. In our experiments, the private declaration is required for only certain scalars and the addresses of vectors/matrices. Therefore, the cost for this data localization is negligible and can be omitted.

It should be noted that in this paper, the middle-grain effect over the fine-grain approach is not represented directly by experimental results since all the experiments with hybrid models are performed by the middle-grain approach. For the fine-grain approach, a user has several options of parallelization levels in a nested loop. Furthermore, not all the OpenMP available loops are OpenMP effective. Therefore, it is rather complicated to build the most effective variant of the fine-grain approach [8]. The middle-grain affect is represented indirectly by the fact that it aids hybrid PC in obtaining a comparable performance with that of MPI.

3.3 Overlapping-oriented Task-schedule

The overlap volume is the key-factor of hybrid TC. An overlap is available if there is no data-dependency between certain computation and communication tasks. In the case of the original MPI task-schedule, the available overlapping volume is usually small. Hereafter, we propose a 4-step algorithm to build a new task-schedule from an original MPI algorithm that permits a greater overlap volume.

Step 1. Select blocks for parallelization. A block should include both communication and computation and occupy a noticeable percentage of the execution time. A global loop-iteration is a good candidate. For each block, perform steps 2 to 4 described below:

Step 2. Build a task-dependency graph for the block. If the block is a loop-iteration, the graph should also include the dependencies concerning the previous and next iterations.

Step 3. Try one or more of the following techniques to enlarge the available ovelapping part:

(a) If the block is a loop-iteration, reconstruct the loop such that larger computation and communication tasks with no data-dependency appear.

(b) If a communication task M depends on only a part of a computation task P, split P into P_1 and P_2 such that M depends on P_1 only. Now, we can overlap P_2 with M.

(c) If a large communication task M depends on a large computation task P, split M and P into M_1, M_2, M_3... and P_1, P_2, P_3..., respectively, such that M_i depends on P_i only. Now M_i and P_j $(i \neq j)$ become independent and can be overlapped. However, the size of M_i should be large enough to avoid a decrease in the communication speed [10].

Step 4. Rebuild the task-dependency graph with the modifications caused by steps 2 and 3. Based on the newly created graph, build the

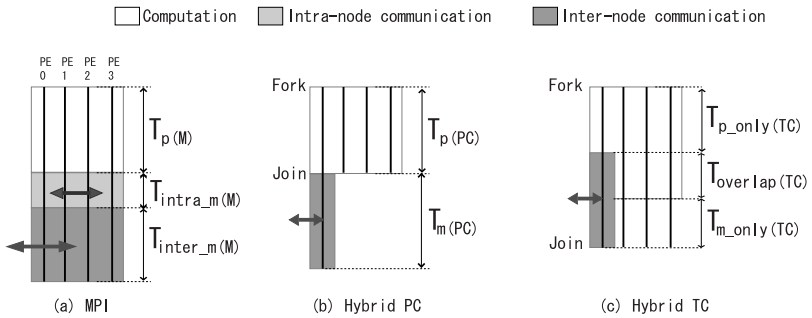

Fig. 1 Time breakdown for parallel models.

hybrid TC task-schedule.

The application of this 4-step algorithm to CG and HPL will be described in details in Sections 6.2 and 7.2, respectively.

4. Parallel Model Comparison

4.1 Methodology

We compare the performance of the three models with respect to their execution time T, which includes com**P**utation time T_p and com**M**unication time T_m. We have $T_p = V_p/S_p$, where V_p and S_p are the computation volume and speed, respectively; and $T_m = V_m/S_m$, where V_m and S_m are the communication volume and speed, respectively. When the workload is not balanced among the nodes, the execution time of the heaviest node should be considered. The bracketed strings "M," "PC," and "TC" are added into the variable names to distinguish among MPI, hybrid PC, and hybrid TC, respectively. For example, $T_{p(M)}$ represents the computation time for the MPI model. The comparison would be performed under the following bases:

Basis 1. $V_{p(M)} = V_{p(PC)} = V_{p(TC)} = V_p$:
All models have the same computation volume. This is natural because all of them are based on the same algorithm.

Basis 2. $S_{p(M)} = S_{p(PC)}$:
MPI and hybrid PC have the same computation speed. As discussed in Section 3.1, the middle-grain approach supplies hybrid PC (and hybrid TC in the *computation-only* phase) the same computation speed as that of MPI. Hybrid TC in a computation-communication overlap is slower because one of the processors is busy with communication at that time.

Basis 3. $V_m = f(nprocs)$:
The communication volume is a function of *nprocs*. This function varies from problem to problem.

Basis 4. $S_{m(PC)} = S_{m(TC)} = S_{m(hybrid)}$ and $S_{inter\_m(PC)} \geq S_{m(hybrid)}$:
Hybrid models have the same communication speed, which is slower than the inter-node communication speed of MPI.

On the Intel cluster, if hybrid TC uses *"!$Omp single nowait"* to implement the communication, the overlap slows down the communication speed. This slowdown can be avoided by using the *"!$Omp master"* directive instead. Such a problem does not occur on the Sun cluster. A possible reason is a higher priority that the Intel OpenMP implementation assigns to the master thread.

Unlike hybrid models, MPI simultaneously performs the communication tasks by *nppn* channels. In platforms where a single PE cannot saturate the inter-node communication bandwidth, the communication speed of MPI will be faster than that of the hybrid models [15),16)].

4.2 Execution Time Calculation

Figure 1 illustrates a breakdown of the execution time for the three models. In reality, each component may be broken into several non-continuous parts; however, for simplicity, we merged them. This mergence does not affect the correctness of the time calculation.

MPI Execution Time $T_{(M)}$:

$$T_{(M)} = T_{p(M)} + T_{intra\_m(M)} + T_{inter\_m(M)}, \quad (1)$$

where $T_{intra\_m(M)}$ and $T_{inter\_m(M)}$ are the times spent for the intra-node and the inter-node communication, respectively.

Hybrid PC Execution Time $T_{(PC)}$:

$$T_{(PC)} = T_{p(PC)} + T_{m(PC)}. \quad (2)$$

Hybrid TC Execution time $T_{(TC)}$:

In hybrid TC, the main objective is to overlap the computation and communication tasks

to the maximum possible extent. However, due to the task-dependencies and the difference between the computation and communication times, usually, there still exist computation and communication tasks that must be performed outside the overlap. With

$T_{p\_only(TC)}$: computation-only time,
$T_{m\_only(TC)}$: communication-only time, and
$T_{overlap(TC)}$: overlap time,

we have

$$T_{m(TC)} = T_{m\_only(TC)} + T_{overlap(TC)},$$

and $T_{(TC)}$ is given by

$$T_{(TC)} = T_{p\_only(TC)} + T_{m(TC)}. \quad (3)$$

4.3 MPI versus Hybrid PC

Due to basis 1 and basis 2, we have

$$T_{p(M)} = T_{p(PC)}. \quad (4)$$

From Eqs. (1), (2), and (4), we can calculate the difference in the execution time between MPI and hybrid PC as

$$\begin{aligned} &T_{(M)} - T_{(PC)} \\ &\quad = T_{intra\_m(M)} + T_{inter\_m(M)} - T_{m(PC)} \\ &\quad = T_{intra\_m(M)} + \frac{V_{inter\_m(M)}}{S_{inter\_m(M)}} - \frac{V_{m(PC)}}{S_{m(PC)}}. \end{aligned}$$

This subtraction, where a positive result would imply outperformance for hybrid PC, cannot determine the better model. In comparison to MPI, hybrid PC does not suffer intra-node communication. On exchange, it has a slower inter-node communication speed (basis 4). In order to determine the better model, we should evaluate the following factors:

Factor 1: $S_{intra\_m(M)}$. Slow intra-node communication speed would cause MPI more costs.

Factor 2: nppn. In general, the MPI intra-node communication volume increases along with *nppn*.

Factor 3: $S_{m(hybrid)}$. If a single channel can saturate (or occupy most) the inter-node communication bandwidth, the communication speed advantage of MPI will be eliminated.

Factor 4: The function V_m of *nprocs*. If V_m is an increasing function of *nprocs*, MPI undergoes more communication.

4.4 Hybrid PC versus Hybrid TC

We compare (2) and (3). According to bases 3 and 4, hybrid PC and hybrid TC has the same communication time $T_{m(hybrid)}$:

$$T_{m(PC)} = T_{m(TC)} = T_{m(hybrid)}. \quad (5)$$

During the computation-only stage in hybrid TC, a node uses all *nppn* PEs for computation as in hybrid PC; hence, we have

$$S_{p\_only(TC)} = S_{p(PC)}. \quad (6)$$

From Eqs. (2), (3), (5), and (6), the difference in the execution time between hybrid PC and hybrid TC is

$$T_{(PC)} - T_{(TC)} = \frac{V_p - V_{p\_only(TC)}}{S_{p(PC)}}.$$

On the other hand,

$$V_{p\_only(TC)} = V_p - V_{p\_overlap(TC)}$$

and

$$V_{p\_overlap(TC)} = T_{overlap(TC)} \times S_{p\_overlap(TC)}.$$

Consequently,

$$T_{(PC)} - T_{(TC)} = T_{overlap(TC)} \times \frac{S_{p\_overlap(TC)}}{S_{p(PC)}}. \quad (7)$$

Thus, *hybrid TC is always faster than hybrid PC.* The difference in the execution time is *in direct proportion to the overlap time.* In the absence of an overlap, the two models have the same execution time.

4.5 MPI versus Hybrid TC

The two models can be compared with respect to the hybrid PC model. Since hybrid TC has great advantages over hybrid PC, and hybrid PC is not so poor in comparison to MPI, we expect an advantage for hybrid TC in most cases.

5. Platform Specification

Table 1 lists the configurations of the Sun and Intel clusters applied. The exchanging bandwidth is measured as the speed at which a node exchanges data. For the Sun cluster, the exchanging bandwidth increases along with the message sizes and almost stabilizes at a size of 2000 double precision numbers. For the Intel cluster, the bandwidth attains a maximum value at a size of 1750 double precision numbers. Therefore, these sizes are used for measuring the exchanging bandwidth. The exchanging bandwidth of hybrid models and the intra-node bandwidth are shown as their ratio to the inter-node exchanging bandwidth of the MPI model.

An examination of the factor list discussed in Section 4.3 reveals that the Sun cluster is more suitable for hybrid models than the Intel cluster with respect to factors 2 and 3 and less suitable with respect to factor 1.

6. Hybrid Solutions for CG

6.1 Problem Description

CG is one of the NAS set of benchmarks. It

Table 1 Sun and Intel cluster specification.

	Sun	Intel
nnodes	2	8
nppn	8	2
Processor type	Ultra Sparc-II	Xeon
Frequency (MHz)	336	2800
IPC	1	2
Peak perf. (GFlops)	5.376	89.600
Cache per PE (MB)	4	0.5
Network (Mbps)	100	1000
OS	Solaris 9	Red Hat 9
MPI library	HPC Cluster 4	MPICH 1.2.5
BLAS library	Forte 6	ATLAS 3.4.1
MPI BW (Mbps)	89.15	607.12
Hybrid BW	75.27%	58.91%
Intra. BW	10.63 times	5.24 times

IPC : instructions per cycle
MPI BW : MPI inter-node exchanging bandwidth
Hybrid BW : in comparison to MPI BW
Intra. BW : intra-node BW, in comparison to MPI BW

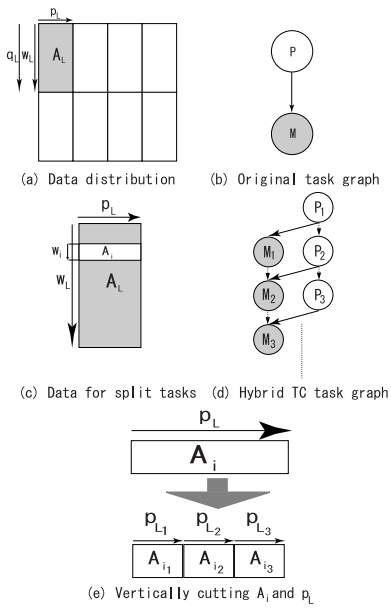

Fig. 2 Construction of a CG hybrid TC task graph.

uses the inverse power method to determine an estimate of the largest eigenvalue of a *fixed* symmetric positive definite $n \times n$ sparse matrix A with a random pattern of nonzeros. The problem size is specified by classes. While class A is extremely small, classes C and D are extremely large for our platforms. As a result, we select class B with $n = 75000$ for our experiments [4].

CG accepts only a power-of-2 *nprocs*, which are mapped onto a $nprows \times npcols$ process-grid. If *nprocs* is a square, $npcols = nprows$; otherwise, $npcols = 2 \times nprows$.

The sparse matrix A is stored by a Compressed Row Storage (CRS) method. The data are equally distributed over processes. A process i stores and operates on a $(n/nprows) \times (n/npcols)$ sparse local matrix A_L. **Figure 2** (a) illustrates data distribution in a sample case where $nprocs = 8$; $nprows = 2$; $npcols = 4$.

6.2 Hybrid TC Task-schedule

Step 1: Block Selection

The performance analysis reveals that more than 90% of the computation cost are spent for a matrix-vector product $q = Ap$, where p is a dense vector and q is the result vector. The product also occupies most of the communication cost of the benchmark. Therefore, we select this matrix-vector product as the block for our middle-grain solution.

Step 2: Original Task-dependency Graph

In a block, an MPI process has to perform two tasks, P and M:

1. P (Computation): multiplies the local matrix A_L by the local vector p_L and stores the result in vector w_L: $w_L = A_L p_L$. With CRS, p_L has a random memory access pattern during this multiplication.

2. M (Communication): exchanges w_L with other processes in the same process-row to reduce the sum to the local result q_L, then exchanges q_L with the *transposing* process to prepare for the next multiplication.

Communication task M depends on the computation task P. There is no independent task-pair, and thus, no available overlap. We continue with step 3.

Step 3: Overlap-generating Techniques

Techniques (a) and (b) are not applicable for CG. We apply technique (c): divide P and M into several parts P_1, P_2, P_3..., and M_1, M_2, M_3..., respectively. A split task P_i now computes a partial product $w_i = A_i p$, and M_i exchanges the partial w_i only. M_i is performed by the master thread and depends on P_i. M_i can be overlapped with any $P_j (i \neq j)$. Figure 2 (c) shows the concerning data for the partial tasks P_i and M_i; Figure 2 (d) shows the new task-dependency graph. Solid arrows indicate *real* dependencies; dotted arrows only indicate that we perform P_i and M_i in an increasing order: P_1, P_2, P_3..., and M_1, M_2, M_3... M_i should not be too small. Here, we set M_i such that w_i has a length of 1750 double precision numbers for the Intel cluster and 2000 double precision numbers for the Sun cluster.

6.3 Task-partitioning and Pseudo-code

For simplicity and without losing performance, we choose each P_i as a grain. Therefore, the number of grains, n_{grains}, is given by

$$n_{grains} = ceiling\left(\frac{sizeof(w_L)}{sizeof(w_i)}\right).$$

During a grain's execution, each element of p_L is accessed repeatedly after a period of time, the length of which depends on the size of p_L. If p_L is extremely large (in comparison to the cache size), the element will not remain in the cache, and this results in a cache-miss. This problem can be solved by vertically cutting A_i and p_L, for example, into A_{i_1}, A_{i_2}, and A_{i_3} and p_{L_1}, p_{L_2}, and p_{L_3}, respectively. Now we can represent w_i as

$$w_i = A_{i_1}p_{L_1} + A_{i_2}p_{L_2} + A_{i_3}p_{L_3}.$$

Each of the three new products has a smaller vertor size and consequently, a better cache hit ratio (Fig. 2 (e)). The technique is also valid for the MPI model. For example, the amended MPI code is four times faster than the original version on the Intel platform with $npcols = 1$ (332.12 Mflops versus 80.24 Mflops). In the case of the Sun cluster, the cache of which is larger, or with the Intel cluster when $npcols > 2$ ($nprocs > 4$), this technique loses its utility.

A hybrid TC pseudo-code based on the middle-grain approach along with the newly created task-schedule is shown in **Fig. 3**. All communication tasks $M_i (i = 1...n_{grains})$ are performed by the master thread. On completion of the computation task P_i, the thread-in-charge should switch on $flag_i$. The execution of the corresponding M_i has to wait until P_i finishes, which is implemented by *"wait until $flag_i$ = .true.."* To minimize this delay, n_{p_only} computation grains should be performed in advance by all threads in the *computing only* phase. A good value of n_{p_only} can be defined based on the computation and communication speeds of an SMP node.

```
!$omp parallel default(shared) private(i)
!$omp do schedule(dynamic)
do i=1, n_p_only
   call P_i
   flag_i=.true.
enddo
!$omp end do nowait
!$omp master
do i=1, n_grains
   wait until flag_i = .true.
   call M_i
enddo
!$omp end master
!$omp do schedule(dynamic)
do i=n_p_only + 1, n_grains
   call P_i
   flag_i=.true.
enddo
!$omp end do
!$omp end parallel
```

Fig. 3 CG, hybrid TC pseudo-code for $q = Ap$ block.

6.4 Communication Volume

During a block, a process first exchanges its local w_L with others from the same process-row $\log_2 npcols$ times; it then exchanges local q_L with its transpose-process once. As a result, the number of messages, $nmsgs$, is $\log_2 npcols + 1$. All these messages have the same length as p_L, the size of which is $n/npcols$. Taking a length of n double-precision numbers as a measurement unit, the communication volume per processor, V_{mpp}, can be calculated as

$$V_{mpp} = \frac{\log_2 npcols + 1}{npcols} \quad (\textbf{Table 2}).$$

For MPI, the communication volume of a node is given by

$$V_{m(M)} = V_{mpp} \times nppn.$$

Meanwhile, for the hybrid models,

$$V_{m(hybrid)} = V_{mpp}.$$

When $nprocs = 2$, the transpose-process of a process is just itself. Under MPI, several processes of a process-row belong to the same SMP node. Thus, the inter-node communication volume can be determined by subtracting the volumes of the self-communication and the intra-node communciation from the total. **Table 3** (a) and (b) list the inter-node communication volume per node for the Sun and Intel clusters, respectively. For hybrid models, the volume is the same for all nodes. For MPI, the value listed belongs to the heaviest node. Occasionally, the heaviest node does not perform inter-node communication by all $nppn$ processes. The number within parentheses rep-

Table 2 CG communication volume per process.

$nprocs$	$nprows$	$npcols$	V_{mpp}
2	1	2	1
4	2	2	1
8	2	4	0.75
16	4	4	0.75

Table 3 CG inter-node communication volume.

(a) The Sun cluster

$nppn$	MPI*	hyb.
2	0.5(1)	0.5
4	0.5(2)	0.5
8	1(4)	0.5

(b) The Intel cluster

$nnodes$	MPI*	hyb.
2	0.5(1)	0.5
4	1(2)	1
8	1(2)	0.75

*: Within parenthesis: number of MPI comm. channels.

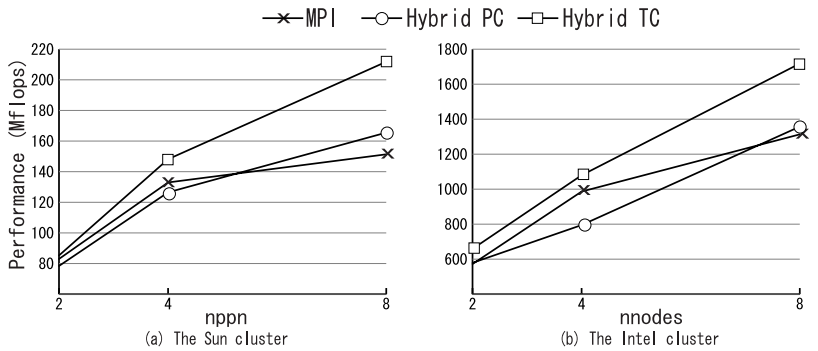

Fig. 4 CG class B experimental results.

resents the number of inter-node communicating channels in this case.

6.5 CG Experimental Results

Figure 4 (a) shows the experimental results of CG on the Sun cluster with $nnodes = 2$ and various values of $nppn$. Meantime, Fig. 4 (b) shows the results on the Intel cluster with $nppn = 2$ and various values of $nnodes$. While the difference in performance between hybrid TC and hybrid PC represents effect of the computation-communication overlap, the difference between hybrid PC and MPI reflects their communication cost.

Due to the crossbar memory architecture, the computation speed of the Sun cluster is almost in direct proportion to the number of processors particpating in computation. Thus,

$$\frac{S_{p\_overlap(TC)}}{S_{p(PC)}} = \frac{nppn - 1}{nppn}.$$

Furthermore, the hybrid TC task-schedule permits us to overlap nearly the entire communication volume with all available values of $nppn$. Together with Eq. (5), we have

$$T_{overlap(TC)} = T_{m(hybrid)},$$

and formula (7) can be rewritten as

$$T_{(PC)} - T_{(TC)} = T_{m(hybrid)} \times \frac{nppn - 1}{nppn}.$$

Thus, the difference in execution time between hybrid PC and hybrid TC increases along with $nppn$. The formula extremely coincides with the actual values listed in **Table 4** (a). A small disparity is due to the remaining part of communication that cannot be overlapped and the measurement error.

Meantime, hybrid PC is worse than MPI while $nppn = 2$ or $nppn = 4$ and better than MPI while $nppn = 8$, which agrees with discussions on factor 2 in Section 4.3. When $nppn = 2$, hybrid PC and MPI have the same communication volume and speed (Table 3 (a)). However, MPI is slightly better than hybrid PC due to the out-of-block part of the solution, which is not thread-parallelized in hybrid models. When $nppn = 4$, while the communication volumes are equal, communication speed of MPI is better due to its two-channel communication pattern. Consequently, it outperforms hybrid PC by approximately 5%. However, with $nppn = 8$, MPI suffers a larger inter-node communication volume (1 versus 0.5). At the same time, a larger $nppn$ causes MPI more intra-node communication cost. As a result, hybrid PC exceeds MPI by approximately 9%.

On the Intel cluster, the difference in execution time between the two hybrid models varies along with the communication volumes. According to Table 3 (b) and Table 4 (b), when $nnodes = 4$, the communication volume is largest ($V_{m(hybrid)} = 1$), and the difference obtains the highest value (17.6 seconds), which results in an outperformance of 34% for hybrid TC (Fig. 4 (b)).

Meantime, MPI and hybrid PC have nearly

Table 4 CG class B, hybrid models execution time.

(a) The Sun cluster, $nnodes = 2$, $T_{m(hybrid)} = 114$ secs.

$nppn$	2	4	8
$(nppn-1)/nppn$	0.5	0.75	0.875
$T_{(TC)}$ (secs.)	636	367	258
$T_{(PC)}$ (secs.)	693	452	359
$T_{(PC)} - T_{(TC)}$*	56(49%)	85(74%)	101(89%)

*: Within parenthesis: ratios to $T_{m(hybrid)}$

(b) The Intel cluster, $nppn = 2$

$nnodes$	2	4	8
$T_{(TC)}$ (secs.)	84.2	51.0	31.8
$T_{(PC)}$ (secs.)	91.6	68.6	41.1
$T_{(PC)} - T_{(TC)}$	7.4	17.6	9.3

the same performance while *nnodes* = 2, MPI is better while *nnodes* = 4, and hybrid PC is better while *nnodes* = 8. These results can also be explained by Table 3 (b). With 2 nodes, MPI and hybrid PC have the same communication pattern and thus, nearly the same performance. It should be noted that the out-of-block part does not play any role on the Intel platform because of its memory bus bottle-neck. For low cache hit ratio problems such as CG, an extra processor cannot provide any computation advantage. With *nnodes* = 4, MPI and hybrid PC have the same communication volume but MPI has two communication channels, which causes a large performance difference (22%). With *nnodes* = 8, MPI undergoes more communication (1 against 0.75), and therefore, its performance is approximately 3% lower than that of hybrid PC.

Hybrid TC exceeds MPI in all the cases. For example, with *nppn* = 8 on the Sun cluster and *nnodes* = 8 on the Intel cluster, hybrid TC outperforms MPI by as much as 40% and 31%, respectively. The everlap always permits hybrid TC to overcome MPI even at the points where MPI obtains the multi-channel communication advantage.

It is noticeable that the difference among the three models is relatively small while *nppn* (for the Sun cluster) or *nnodes* (for the Intel cluster) is small. For the Intel cluster, the decrease of *nnodes* results in an increase of the computation volume per node. For the Sun cluster, the decrease of *nppn* results in a decrease of the computation speeed. In both cases, it leads to a decrease of the rate of the communication cost, and consequently, a decrease in the performance difference among the three models.

7. Hybrid Solutions for HPL

7.1 Problem Description

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

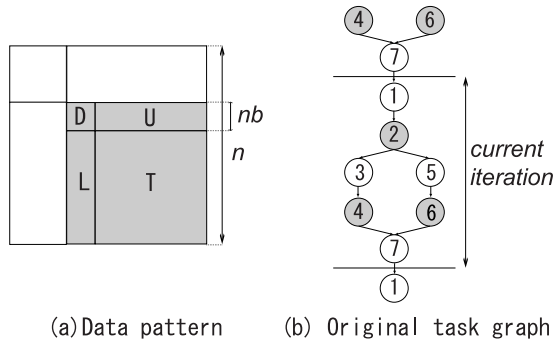

Fig. 5 HPL, data pattern and an original task-dependency graph.

Table 5 HPL, original task list.

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

*: Communication tasks
Cur. dep.: Current iteration's dependencies
Prev. dep.: Previous iteration's dependencies

The data in HPL are stored in $nb \times nb$ square blocks, where *nb* is the *block size* and can be adjusted upon 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 aids in decreasing communication cost [5].

According to the right-looking variant, LU factorization is performed by a loop with $ceiling(n/nb)$ iterations. Related data for the i^{th} iteration is shown in **Fig. 5** (a). D is the i^{th} block of the main diagonal. L, U, and T are the current parts of the lower, upper, and trailing matrices, respectively. **Table 5** lists the task-list for such an iteration. Tasks 2, 4, and 6 are communication tasks. Task 1 depends on task 7 of the previous iteration. For tasks 4 and 7, symbols U^0 and L^0 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 7 is the major computation task. Meanwhile, tasks 4 and 6 occupy almost all the communication volume.

7.2 Hybrid TC Task-schedule

Step 1: Block Selection

A loop-iteration described above is chosen as the middle-grain block.

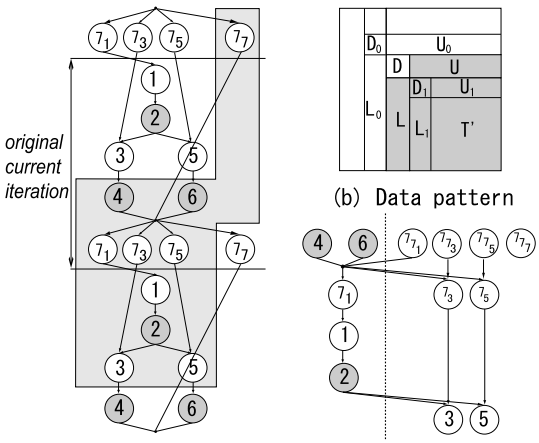

Fig. 6 HPL, building a hybrid TC task-schedule.

Step 2: Original Task-dependency Graph

Based on Table 5, a task-dependency graph is created and shown in Fig. 5 (b), wherein communication tasks are represented by shaded circles. As seen in the graph, there are only two independent pairs of tasks available for an overlap: 3 with 6 or 5 with 4. However, computation tasks 3 and 5 are extremely small; hence, we should go to step 3.

Step 3: Overlap-Generation Techniques

As shown in Fig. 5 (b), all tasks of the current iteration depend on task 7 of the previous iteration that updates the "previous" trailing matrix including the "current" D, U, L, and T. Therefore, we apply technique (b) to split task 7 into 7_1, 7_3, 7_5, and 7_7 that updates D, U, L, and T, respectively. We have a new task-dependency graph shown in **Fig. 6** (a). Tasks 1, 3, 5, and 7 now depend on the previous tasks 7_1, 7_3, 7_5 and 7_7, respectively. Since tasks 4 and 6 (major communication) and previous task 7_7 (major computation) are independent, we apply technique (a) to reconstruct the loop so that they lie in the same iteration and become available for overlap. In Fig. 6 (a), tasks of a new loop iteration are bounded by a shaded polygon. It includes tasks belonging to three original iterations. Then, technique (b) is applied once more: 7_7 are split into 7_{7_1}, 7_{7_3}, 7_{7_5}, and 7_{7_7} to break the "big" dependency into smaller ones. **Table 6** and Fig. 6 (b) show the new task-list and corresponding data blocks. Figure 6 (c) shows the final version of the hybrid TC task-dependency graph. Tasks 7_{7_1}, 7_{7_3}, 7_{7_5}, and 7_{7_7} are obtained from the previous iteration; tasks 1, 2, 3, and 5 are obtained from the next iteration. Tasks in the left of the vertical line are assigned to the master thread. The remaining are partitioned into grains and should be parallelized by *"!$Omp do nowait."*

Table 6 HPL, hybrid TC task list.

No.	Description	Dependencies
4*	Bcast(U)	-
6*	Bcast(L)	-
7_{7_1}	$D_1 = D_1 - L_0U_0$	-
7_{7_3}	$U_1 = U_1 - L_0U_0$	-
7_{7_5}	$L_1 = L_1 - L_0U_0$	-
7_{7_7}	$T' = T' - L_0U_0$	-
7_1*	$D_1 = D_1 - LU$	$4, 6, 7_{7_1}$
7_3	$U_1 = U_1 - LU$	$4, 6, 7_{7_3}$
7_5	$L_1 = L_1 - LU$	$4, 6, 7_{7_5}$
1*	Decom(D_1)	7_1
2*	Bcast(D_1)	1
3	Solve($D_1U_1 = U_1^0$)	2
5	Solve($L_1D_1 = L_1^0$)	2

*: Tasks assigned to the master thread

```
#pragma omp parallel default(shared) private(i)
{
  #pragma omp master
  {
    (4); flag(4)=1;
    (6); flag(6)=1;
    wait untill flag(7_7_1); (7_1);
    (1);
    (2); flag(2)=1;
  }
  #pragma omp single nowait
  {
    (7_7_1); flag(7_7_1)=1;
  }
  #pragma omp for schedule(dynamic) nowait
  for (i=0; i<sizeof(U_1); i++) {
    (7_7_3[i]); flag(7_7_3[i])=1;
  }
  #pragma omp for schedule(dynamic) nowait
  for (i=0; i<sizeof(L_1); i++) {
    (7_7_5[i]); flag(7_7_5[i])=1;
  }
  #pragma omp for schedule(dynamic) nowait
  for (i=0; i<sizeof(T'); i++) (7_7_7[i]);
  #pragma omp for schedule(dynamic) nowait
  for (i=0; i<sizeof(U_1); i++) {
    Wait until flags(4,6,7_7_3[i]); (7_3[i]); flag(7_3[i])=1;
  }
  #pragma omp for schedule(dynamic) nowait
  for (i=0; i<sizeof(L_1); i++) {
    Wait until flags(4,6,7_7_5[i]); (7_5[i]); flag(7_5[i])=1;
  }
  #pragma omp for schedule(dynamic) nowait
  for (i=0; i<sizeof(U_1); i++) {
    Wait until flags(2,7_3[i]); (3[i]);
  }
  #pragma omp for schedule(dynamic) nowait
  for (i=0; i<sizeof(L_1); i++) {
    Wait until flags(2,7_5[i]); (5[i]);
  }
}
```

Fig. 7 HPL, hybrid TC pseudo-code for an iteration.

7.3 Task-partitioning and Pseudo-code

Tasks on the right hand side of Fig. 6 (c) are partitioned into grains such that each grain up-

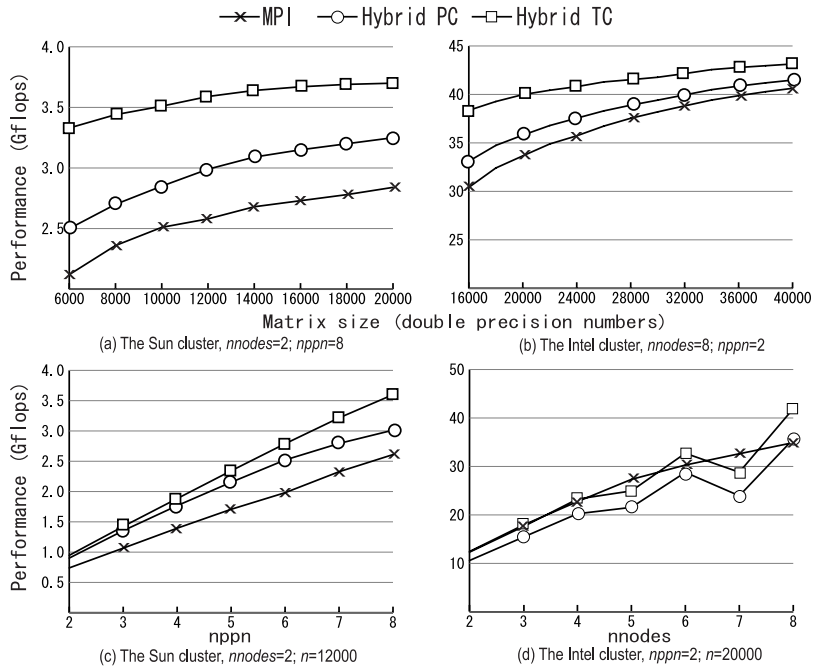

Fig. 8 HPL experimental results.

Table 7 HPL communication volume.

nprocs	P	Q	V_m
2	1	2	1
3	1	3	2
4	2	2	1
5	1	5	4
6	2	3	1.33
7	1	7	6
8	2	4	1.75

dates an $nb \times nb$ data block. Taking a data block as a measurement unit, according to Table 6, tasks 7_{7_3}, 7_3, and 3 are partitioned into $sizeof(U_1)$ grains; tasks 7_{7_5}, 7_5, and 5 are partitioned into $sizeof(L_1)$ grains. Meanwhile, the numbers of grains for tasks 7_{7_1} and 7_{7_7} are 1 and $sizeof(T')$, respectively. All the grains are performed by applying functions of a well-tuned BLAS library. By this method, hybrid models achieve the same computation speed as MPI.

A C-style hybrid TC pseudo-code is shown in **Fig. 7**. All computation tasks are performed by *"#pragma omp for schedule(dynamic) nowait"* with an exception for task 7_{7_1} that contains a single grain and is performed by *"#pragma omp single nowait."* All the data-dependencies are followed by flags. By this implementation, if the master thread completes its tasks earlier, it can join the computation team.

7.4 Communication Volume

In HPL, processes communicate by broadcasting data. During an iteration, the process owner of data block D is the heaviest one. It broadcasts its local upper triangular U_L to $(P-1)$ process-destinations in the same process-column. Then, it broadcasts its local lower triangular L_L to $(Q-1)$ process-destinations in the same process-row. The sizes of U_L and L_L are $sizeof(U)/Q$ and $sizeof(L)/P$, respectively. U and L always have the same size. The D-owner process also broadcasts D to $((P-1)+(Q-1))$ destinations; however, when U is large enough, the cost of this broadcast is negligible.

Table 7 lists the communication volume for the heaviest process during an iteration. Broadcasting D is omitted. The size of U is considered as a measurement unit. For simplicity, we assume that the communication volume is in direct proportion to the number of destinations. Thus, the communication volume is given by

$$V_m = \frac{P-1}{Q} + \frac{Q-1}{P}.$$

Values of P and Q are selected such that V_m is minimized. For hybrid models, the inter-node communication volume is the same as the values shown in the table. For MPI, we should multiply these values by *nppn*; and then subtract the intra-node communication volume.

7.5 HPL Experimental Results

Figure 8 shows the experimental results for HPL. The block size *nb* is selected such that MPI can obtain the best performance ($nb = 80$ in almost all cases on both the platforms).

Figure 8 (a) and Fig. 8 (b) show the experimental results using maximum resources of the clusters and varying the problem size to the limit restricted by the system memory. A significant outperformance of hybrid TC over hybrid PC on both the systems indicates the effect of the new task-schedule, which allows a great overlap volume. Meantime, hybrid PC is better than MPI on both the systems, which represents that MPI undergoes a greater communication cost.

When the matrix size is growing, the communication cost increases slower than the computation cost and therefore, the rate of the communication cost decreases. Consequently, the performance difference between the three models also decreases, and the three performance lines of the graph go closer. For example, the difference between hybrid TC and MPI at the maximum value of the matrix size are 30% for the Sun cluster and 6% for the Intel cluster. These values were 57% and 26%, respectively, at the beginning points. However, the time saved by the hybrid models also increases along with the problem size, as shown by **Fig. 9**.

In Fig. 8 (c) and Fig. 8 (d), we fix the problem size and change *nppn* (for the Sun cluster) and *nnodes* (for the Intel cluster). On the Sun cluster, hybrid TC still dominates. Hybrid PC is still better than MPI. The difference among the three models increases along with *nppn*. However, we observe a different pattern on the Intel cluster. Hybrid models get poor performance when a well-balanced $P \times Q$ process-grid is not available (*nnodes* = 5 and *nnodes* = 7). Table 7 shows that the communication volume is extremely heavy at these points and the multi-channel communication advantage of MPI becomes noticeable. This problem for hybrid models can be solved by: (1) decreasing *nnodes* to a reasonable value (for example, *nnodes* = 6 provides a better performance than *nnodes* = 7); and (2) applying a better broadcasting algorithm to decrease the communication cost [5].

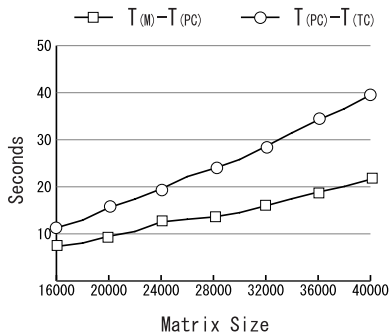

Fig. 9 Difference in execution time among models. The Intel Cluster, $nnodes = 8, nppn = 2$.

Similar to the CG problem, with a small value of *nppn* (for the Sun cluster) and *nnodes* (for the Intel cluster), the performance difference among the three models is small due to a decrease in the rate of the communication cost.

8. Conclusions

In this paper, we proposed and applied an algorithm to build effective hybrid MPI-OpenMP solutions for SMP clusters, which includes a middle-grain approach and an ovelapping-oriented task-schedule. Our hybrid TC solution dominates MPI in all experiments on different cluster platforms. For CG class B, hybrid TC outperforms MPI by 40% and 31% on a Sun and an Intel cluster, respectively. For HPL, with a middle problem size, hybrid TC exceeds MPI 57% and 26%; with the maximum available problem size, hybrid TC still dominates MPI by 30% and 6%, respectively, for the Sun and the Intel clusters. Hybrid TC should be considered as the first priority for a parallel programming model on SMP clusters.

Acknowledgments This research is supported in part by the Grants-in-Aid for Scientific Research of Japan Society for the Promotion of Science (JSPS) No.15500033.

References

1) MPI: Message Passing Interface Forum. http://www.mpi-forum.org/
2) OpenMP: The OpenMP Architecture Review Board. http://www.openmp.org/
3) Krawezik, G. and Cappello, F.: Performance Comparison of MPI and three OpenMP Programming Styles on Shared Memory Multiprocessors, *Proc. 15th annual ACM symposium on Parallel Algorithm and Architectures*, pp.118–127 (2003).
4) NAS: The NAS Parallel Benchmark. http://www.nas.nasa.gov/Software/NPB/
5) Petitet, A., Whaley, R. C., Dongarra, J. and Cleary, A.: HPL—A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Comput-

ers. http://www.netlib.org/benchmark/hpl/
6) 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).
7) 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).
8) Cappello, F. and Etiemble, D.: MPI versus MPI+OpenMP on IBM SP for the NAS Benchmark, *Proc. Supercomputing 2000* (2000).
9) Boku, T., Itakura, I., Yoshikawa, S., Kondo, M. and Sato, M.: Performance Analysis of PC-CLUMP based on SMP-Bus Utilization, *Proc. Workshop on Cluster Based Computing 2000* (2000).
10) 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).
11) Nakajima, K.: OpenMP/MPI hybrid vs. Flat MPI on the Earth Simulator: Parallel Iterative Solvers for Finite Element Method, *Proc. International Workshop on OpenMP 2003* (2003).
12) 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).
13) 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).
14) Wellein, G., Hager, G., Basermann, A. and Fehske, H.: Fast sparse matrix-vector muliplication for TeraFlop/s computers, *Proc. Vector and Parallel Precessing* (2002).
15) Rabenseifner, R. and Wellein, G.: Communication and Optimization Aspects of Parallel Programming Models on Hybrid Architectures, *International Journal of High Performance Computing Application*, Vol.17, No.1 (2003).
16) Rabenseifner, R.: Hybrid Parallel Programming: Performance Problems and Chances, *Proc. 45th CUG (Cray User Group) Conference 2003* (2003).

(Received May 18, 2004)
(Accepted October 2, 2004)

Ta Quoc Viet received his M.E. degree from the Graduate School of Information Systems, the 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 Jul. 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 Aug. 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.

Ben A. Abderazek received his Ph.D. degree in Computer Engineering from UEC in 2002. Currently he is working on Queue processor hardware implementation on FPGA. His main research interests include parallel processor, system implementations, reconfigurable architectures, and distributed storage systems. His is a member of IEEE, ACM and IEICE.

Masahiro Sowa received the D.E. degree from Nagoya University in 1974 and then, became an assistant professor of the University of Gunma. From 1986, he was a Professor of Nagoya University. Since 1993, he has been a Professor of the Graduate School of Information Systems, UEC. His research interests include parallel processing. He is a member of IEEE, IEICE, JSSST, and ACM.