J Supercomput (2009) 50: 177–207
DOI 10.1007/s11227-008-0258-1

A directive-based MPI code generator for Linux PC clusters

Chao-Tung Yang · Kuan-Chou Lai

Published online: 16 December 2008

Abstract Computation requirements in scientific fields are getting heavier and heavier. The advent of clustering systems provides an affordable alternative to expensive conventional supercomputers. However, parallel programming is not easy for non-computer scientists to do. We developed the Directive-Based MPI Code Generator (DMCG) that transforms C program codes from sequential form to parallel message-passing form. We also introduce a loop scheduling method for load balancing that depends on a message-passing analyzer, and is easy and straightforward to use. This approach provides a completely different view of loop parallelism from that in the literature, which relies on dependence abstractions. Experimental results show our approach can achieve efficient outcomes, and DMCG could be a general-purpose tool to help parallel programming beginners construct programs quickly and port existing sequential programs to PC Clusters.

Keywords Linux PC cluster · Message passing programming · Parallel loops · Speedup · Cluster computing

1 Introduction

As computation requirements in the science field become more demanding, the role of parallel architectures in helping to satisfy those requirements becomes increas-

C.-T. Yang (✉)
High-Performance Computing Laboratory, Department of Computer Science, Tunghai University, Taichung 40704, Taiwan
e-mail: ctyang@thu.edu.tw

K.-C. Lai
Department of Computer and Information Science, National Taichung University, Taichung 40306, Taiwan
e-mail: kclai@ntcu.edu.tw

ingly important. Parallel architectures include shared-memory multiprocessor systems, distributed-memory multiprocessor systems, and clustering systems.

Shared-memory multiprocessor systems contain two or more processors that all share the same memory address. One operating system manages the processors, and they share all resources, making communication among them very convenient in the programmer's view. However, there can be memory-sharing and scalability problems in such systems. Distributed-memory multiprocessor systems may also contain many processors, but each has its own local memory. The processors exchange data and coordinate processes by message-passing. However, writing parallel programs for such systems is not so convenient for programmers.

Another high performance computing system choice is the PCs/workstation cluster [1–3]. Many individual PCs/workstations, generally with homogeneous system architectures, are connected by high-speed networks, such as Beowulf Cluster, Network of Workstations (NOW), Cluster of Workstations (COW), PC/Workstation Farms, among others. Message-passing techniques such as Message Passing Interface (MPI) [4] and Parallel Virtual Machine (PVM) [5] can be used to achieve parallel programming in cluster computing systems. The advantages of cluster computing systems include high scalability, high availability, and low cost/performance ratios. Clusters for performing computation-intensive experiments like the N-body problem, DNA sequence simulations, weather prediction, nuclear simulation, and high-energy physics, etc., are easy to build. According to the latest thirtieth top 500 report [6], clustered architecture systems make up about 80% of the world's 500 most powerful systems; they are competitive with conventional supercomputers on performance. Given their great performance and affordable price, it is believed that clustering systems will enter the high-performance architecture mainstream and become increasingly popular.

Full utilization of clustering systems requires parallel programming, with methods suitable for various parallel architectures, e.g., threading for shared-memory systems and message-passing for distributed-memory systems. Programmers can create parallel programming for message-passing systems via three approaches: by using a new parallel programming language, extending an existing sequential language to handle message passing, and by using an existing sequential language with a library of external functions for message passing. The third option is the most popular using either MPI [4] or PVM [5]. In such programming, programmers must explicitly distribute data to each processor. All computations within a processor can involve only its own data. Whenever nonlocal data are required, the programmers must facilitate message-passing function calls to transfer data among processors.

Parallelizing compilers find parallelisms in sequential programs and generate appropriate parallel codes for parallel systems. Parallelization entails three subproblems: identifying potential parallelisms, automatically or by programmers mapping processes and data layouts to processors, optimizing, and generating target parallel codes [7]. Many parallelizing compilers for shared-memory systems and global addressing systems exist: SUIF at Stanford University, Polaris at Purdue University and the University of Illinois at Urbana-Champaign, ParaScope at Rice University, etc. This includes OpenMP compilers, perhaps the most popular in this field.

Most parallel compilers assume the underlying systems are shared-memory systems when generating parallel codes, making them unsuitable for cluster computing.

Table 1 Parallelizing compilers for distributed-memory systems

Compiler	Company or Laboratory	Availability/Cost
Paraguin	University of North Carolina at Wilmington	Under development; prototype available
PARADIGM	University of Illinois	Not available
VAST-HPF	Crescent Bay Software	$2,395, 8-node license
Bert 77	HPC Design	$2,449, 8-node academic single-user license
PGI PGHPF (part of CDK)	Portland Group Compiler Technology team at STMicroelectronics	$2,958, 10 processors node-locked

Users may adopt the Distributed-Shared Memory (DSM) library to create convenient abstractions that allow parallel codes written for shared-memory systems to be run by distributed systems. However, distributed shared memories are still sources of inefficiency. Even though a few companies and laboratories have developed parallelizing compilers for distributed-memory systems, none has a low cost/performance ratio, as shown in Table 1. The lack of such parallelizing compilers for distributed-memory systems prompted us to develop one.

In this study, we report on developing an MPI code generator called the Directive-Based MPI Code Generator (DMCG), and an assisted-learning tool for MPI programming beginners. Our approach was based on analyzing communication models for distributed-memory systems. It provided a completely different view of loop parallelism from those that rely on dependence abstractions. Experimental results show our approach outperforms hand-revised codes, and that DMCG could be a general-purpose tool for creating parallel programming quickly and porting existing sequential programs to PC Clusters.

The rest of this paper is organized as follows. Section 2 introduces the concepts of data dependencies, loop partitioning, message-passing interfacing and communication models. Section 3 presents a performance evaluation of collective communication. Section 4 describes our system and kernel technologies. In Sect. 5, we present experimental results, and comparisons with hand-revised codes. Finally, concluding remarks and future directions are given in Sect. 6.

2 Background

2.1 Data dependence

There are two types of dependence: control and data [10–12]. If, in a control flow graph, there is a path from statement S_1 to statement S_2, i.e., S_1 determines whether S_2 can be executed, we then say that a control dependence exists between S_1 and S_2. For example:

$$\begin{array}{ll} S_1 & \text{if } (a = 5)\{ \\ S_2 & \qquad\qquad b = 2 \\ S_3 & \}. \end{array}$$

Data dependence exists between statements S_1 and S_2 if both access the same memory location and at least one of them writes to that location. There are three types of data dependence: flow dependence (also called true dependence), anti-dependence, and output dependence. S_5 is flow-dependent on S_4 when S_4 must be executed before S_5 because S_4 writes a value that S_5 must read.

$$\begin{array}{ll} S_4 & a = 10; \\ S_5 & b = a + 5 \end{array}$$

Antidependence occurs when S_6 reads a memory location to which S_7 later writes. We say that S_7 is antidependent on S_6 since the read-write order between S_6 and S_7 is in reverse order of flow dependence. Indeed, we can resolve this kind of data dependence by using a temporary variable to hold the value stored in variable a. If S_7 is executed before S_6, S_6 can use this temporary variable since it will have the correct value.

$$\begin{array}{ll} S_6 & b = a + 5 \\ S_7 & a = 10 \end{array}$$

Output dependence exists when statements S_8 and S_9 write to the same memory location. The value stored in memory should be nearest the statement that will read it.

$$\begin{array}{ll} S_8 & a = b * 10 + 5 \\ S_9 & a = c + d \end{array}$$

The dependencies above occur in codes without loops. Dependencies bridging between iterations are called loop-carried dependencies. S_{11} and S_{12} have no dependent relationship in one iteration, but in the next iteration, S_{12} reads a value that was written by S_{11} in the previous iteration creating flow dependence. We say that S_{12} is flow-dependent on S_{11} with a distance vector $\{(1)\}$. We note that distance vectors describe dependencies between iterations not array elements. However, when there is data dependence between statements, the statements must be executed in sequence and cannot be parallelized.

$$\begin{array}{ll} S_{10} & \text{for } (i = 1; i < n; i++)\{ \\ S_{11} & \qquad a[i] = c + d \\ S_{12} & \qquad b[i] = a[i-1] * 10 \\ S_{13} & \} \end{array}$$

2.2 Loop scheduling

If loops can be executed in parallel, we want to split them into sets of tasks on different processors. Therefore, a good loop-scheduling algorithm can achieve good load balancing with only minimal overhead. Several loop-scheduling methods are currently available, Pure Self-Scheduling, Guided Self-Scheduling, Chunked Self-Scheduling, and Factoring Self-Scheduling [13–18]. We describe these scheduling in the following:

Pure Self-Scheduling (PSS) is the first straightforward dynamic loop scheduling algorithm. In this paper, a processor is said to be idle if it has not been assigned a

Table 2 Margin formulas and examples of dynamic loop scheduling

Scheme	Formulas	$N = 1000$, $P = 4$
SS	$K_i = 1$	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1...
CSS(k)	$K_i = k$	125 125 125 125 125 125 125 125
CSS/λ	$K_i = \lceil N/\lambda \rceil$	250 250 250 250
GSS	$K_i = \lceil R_i/P \rceil$, $R_0 = N$, $R_{i+1} = R_i - K_i$	250 188 141 106 79 59 45 33 25 19 14 11 8 6 4 3 3 2 1 1 1 1
FSS	$K_i = (1/2)^{\lceil i/P \rceil} \times N/P$	125 125 125 125 62 62 62 62 32 32 32 32 16 16 16 16 8 8 8 8 4 4 4 4 2 2 2 2 1 1 1 1

chunk of workload or it has finished the assigned workload. That is, an idle node does not have a chunk of workload to execute. Whenever a processor gets idle, iterations are assigned to it. This algorithm achieves good load balancing, but induces excessive overhead [10].

Chunk Self-Scheduling (CSS) assigns k iterations each time, where k, the chunk size, is fixed and must be specified by either the programmer or by the compiler. When k is 1, the scheme is purely self-scheduling, as discussed above. Large chunk sizes cause load imbalances, while small chunk sizes are likely to produce excessive scheduling overhead [10].

Guided Self-Scheduling (GSS) can dynamically change the numbers of iterations assigned to idle processors [11]. More specifically, the next chunk size is determined by dividing the number of remaining iterations of a parallel loop by the number of available processors. The property of decreasing chunk size implies that an effort is made to achieve load balancing and to reduce the scheduling overhead. By assigning large chunks at the beginning of a parallel loop, one can reduce the frequency of communication between master and slaves. The small chunks at the end of a loop partition serve to balance the workload across all working processors.

Factoring Self-Scheduling (FSS) assigns loop iterations to working processors in phases [9]. During each phase, only a subset of remaining loops iterations (usually half) is equally divided among available processors. Because FSS assigns a subset of the remaining iterations in each phase, it balances workloads better than GSS when loop iteration computation times vary substantially. The synchronization overhead of FSS is not significantly greater than that of GSS.

Assume P available processors, N iterations in the DOALL loop, and a K_i-size ith partition. Several algorithms' formulae for calculating K_i are listed in Table 2, where the CSS/k algorithm distributes the DOALL loop in k equal-sized chunks. Table 2 also gives sample sizes for SS, CSS(125), CSS/4, GSS, FSS, and when $N = 1000$ and $P = 4$.

The approach above is dynamic loop scheduling in which each loop partition must be mapped to a processor. An alternative approach is static scheduling. There are two static loop scheduling methods: block and cyclic [19]. In static scheduling, the number of chunks equals the number of processors, i.e., a scheduler is not needed when each partition is assigned to one processor. Adopting block or cyclic scheduling involves a trade-off between locality and workload distribution since block scheduling

assigns blocks of continuous iterations to one processor, while cyclic scheduling assigns specific amounts of cyclic iterations to each processor.

2.3 PC clusters

A PC cluster uses multicomputer architecture and features a parallel computing system that consists of one or more master nodes and available compute nodes or cluster nodes, interconnected via widely available network interconnects. All of the nodes in a typical PC cluster are commodity systems-PCs, workstations, or servers-running commodity software such as Linux.

The master node acts as a server for Network File System (NFS) and as a gateway to the outside world. As an NFS server, the master node provides user file space and other common system software to the compute nodes via NFS. As a gateway, the master node allows users to gain access through it to the compute nodes. Usually, the master node is the only machine that is also connected to the outside world using a second network interface card (NIC). The sole task of the compute nodes is to execute parallel jobs. In most cases, therefore, the compute nodes do not have keyboards, mice, video cards, or monitors. All access to the client nodes is provided via remote connections from the master node. Since compute nodes do not need to access machines outside the cluster, nor do machines outside the cluster need to access compute nodes directly, compute nodes commonly use private IP addresses, such as the 10.0.0.0/8 or 192.168.0.0/16 address ranges.

From a user's perspective, a PC cluster appears as a distributed memory multiprocessor system. The most common methods of using the system are to access the master node either directly or through Telnet or remote login from personal workstations [1]. Once on the master node, users can prepare and compile their parallel applications, and also spawn jobs on a desired number of compute nodes in the cluster. Applications must be written in parallel and use the message-passing programming model. Jobs of a parallel application are spawned on compute nodes, which work collaboratively until finishing the application. During the execution, compute nodes use standard message-passing middleware, such as Message Passing Interface (MPI) [4, 20] and Parallel Virtual Machine (PVM) [5], to exchange information.

- Cluster computing focuses on platforms consisting of often homogeneous interconnected nodes in a single administrative domain,
- Clusters often consist of PCs or workstations and relatively fast networks,
- Cluster components can be shared or dedicated,
- Application focus is on cycle-stealing computations, high-throughput computations, and distributed computations.

2.4 Message passing interface

The *Message Passing Interface standard* (MPI) [1] describes a message-passing application programming interface with protocol and semantic specifications such as message buffering and message delivery progress requirements. MPI includes point-to-point message passing and collective (global) operations, and provides a substantial set of libraries for writing, debugging, and performance testing distributed programs.

MPI was established for writing message-passing programs, and its main advantages are portability and ease-of-use. The benefits of standardization are especially apparent in distributed-memory communication environments in which higher-level routines and/or abstractions are built upon lower-level message passing routines. Furthermore, defining a message-passing standard provides vendors with well-defined routines they can implement efficiently, or in some cases provide hardware support for enhancing scalability. Before MPI, many message-passing libraries were offered by various parallel computing system vendors; however, their portability was a big problem.

There are various implementations of MPI, for example, LAM/MPI [20], now supported and maintained by the University of Notre Dame, MPICH, implemented by the Argonne National Lab/Mississippi State University [21], MPI/Pro, a commercial implementation for clusters, *Windows NT*, and multicomputers provided by MPI Software Technology, Inc. [22]. We use the LAM/MPI implementation in this study because it is implemented according to the MPI standard and creates no MPI programming portability problems in our system.

2.5 Communication model

Most loop parallelization technologies depend on abstract dependence analyses. Communication model analysis plays an important role in message-passing parallel programming; given the importance of translating sequential codes into parallel ones and the fact that various models have different send/receive patterns.

In [23], McGarvey et al., classified four categories of point update methodology: Independent, Nearest Neighbor, Quasi-Global, and Global. Since we care most about communication behavior among processors, we simplify classification into three categories: *Independent*, *Semi-Global* (merging Neighbor and Quasi-Global), and *Global*, as shown in Fig. 1.

Each node (processor) executes update algorithms and depends on data from previous steps. If an update algorithm requires only data from previous steps, it is *Independent* and needs no communication with other processors. It is commonly referred to as "embarrassingly parallel", such as calculating the value of PI, Mandelbrot set, matrix manipulation, etc.

If an update algorithm requires some outside data, it is *Semi-Global*. This complicates the mapping from sequential to parallel because semantics must be parsed precisely to determine which nodes need which data. For this problem, we introduce a new concept, "data distance" directives annotated by users to tell the compiler how far away the data is from it. Note that the data distance here resembles the distance vector in data dependence, but they are not the same. Our system does not parallelize program blocks subject to any form of data dependence. For example, Jacobi iteration, which updates each item with values from its neighbors, is in this category. For this communication model, users must indicate data distance from the compiler: $(-1, -1)$, $(-1, 1)$, $(1, -1)$, and $(1, 1)$. We leave parallelizing loops in this communication model for future work.

The last communication model is *Global* in which the update algorithm requires data from all others. All-pairs shortest-path solved by Floyd's algorithm [24], which

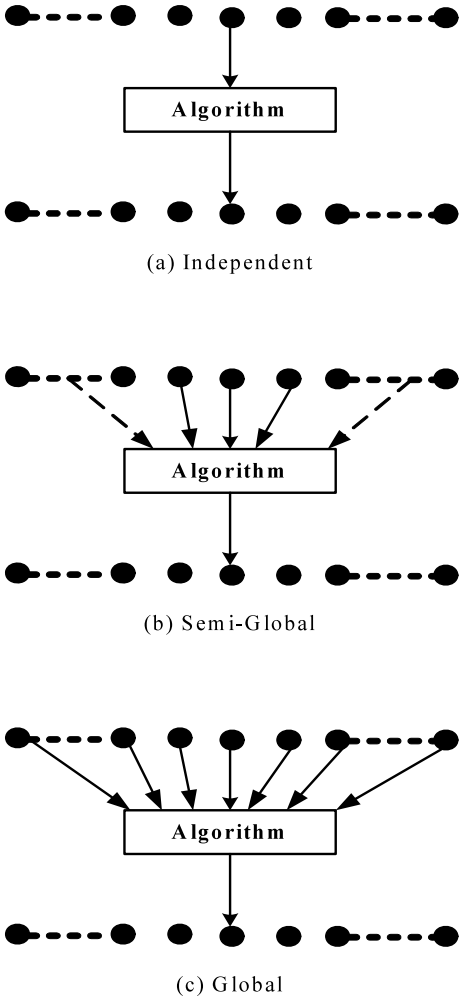


Fig. 1 Three type of communication models for loop parallelization

updates paths stored in a two-dimensional array during each iteration, belongs in this category.

3 Performance evaluation of collective communication

3.1 Collective communication

Various application types exhibit certain communication patterns, and rather than coding these patterns with point-to-point communication functions, MPI provides collective communication functions to handle them. Collective communication involves groups of processes that manipulate "common" pieces or sets of information as mentioned above. Generally, efficient algorithms are used to build collective communication functions from point-to-point communication functions.

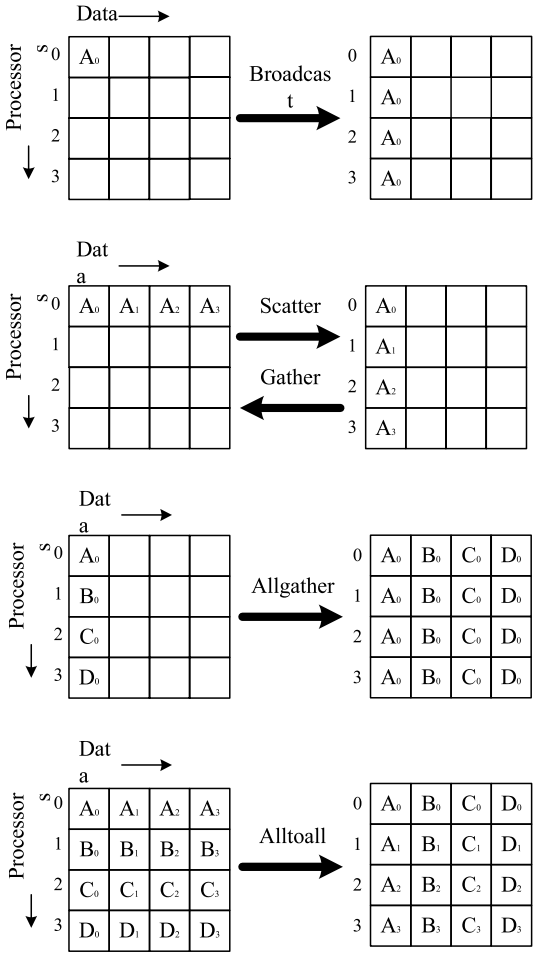


Fig. 2 Collective communication functions

MPI collective communication functions can be divided into three categories: *synchronization*, *data movement*, and *global computation*. Here, we discuss only data movement functions, as shown in Fig. 2. Some say collective communication functions are more efficient than point-to-point communication functions, but that is not so clear-cut because collective communication implementations are sometimes related to synchronizations, some of which are redundant. Some suggest that send/receive in parallel programs should be "considered harmful" and avoided in favor of collective communication functions as far as possible. The benefits over send/receive include: simplicity, programmability, performance, expressiveness, and predictability [25].

In this section, we report on experiments conducted to evaluate the performance of MPI collective communication data movement functions; the results were taken into consideration in our parallelizing system.

3.2 Timing methodologies

Program execution times must be measured when evaluating performance. There are two timing methodology categories: coding for single nodes and coding for parallel machines. In coding for single nodes, we must distinguish between *wallclock time* and *CPU time*. Time-sharing systems may be executing many processes simultaneously. Wallclock time is the "real" or elapsed time used by all processes. CPU time is that actually used by processes.

In parallel systems, parallelism is accomplished via message passing, so communication time must be separated from total execution time since it plays an important role in performance analyses. Two factors determine the communication performance: *latency* and *bandwidth*. Latency is the time required to send a message of zero length from one node to another. It is heavily dependent on network protocols. Bandwidth determines the rate at which messages are sent.

Wallclock time is best used on dedicated machines. When a program first runs, it may vary due to acquiring page frames. Thus, CPU time is useful in evaluating whole programs. Certain system function calls and MPI implementations measure these time categories, as shown in Fig. 3.

System call *gettimeofday* or MPI implementation *MPI_Wtime* may be used to get wallclock time. As we known, *time*, a UNIX system command, can be used to measure total execution (CPU) time, or the standard C function call *clock* can be inserted into a program. Examples of these functions are shown in Fig. 3. In (a), the time unit is microseconds, in (b), seconds, (c) shows a shell command not used in programs; its time unit is seconds, and in (d), the time unit is microseconds. To show it in seconds, we divide total time by the constant CLOCKS_PER_SEC.

3.3 Communication analysis

Since we compared point-to-point and collective communication performances, we must analyze collective communication behavior. Collective communication functions are all implemented by point-to-point communication and are all block. If we are willing to do it, we can go ahead, but its performance will only be as good as its implementations. Here, we analyze three collective communication functions: *broadcast*, *scatter,* and *allgather*. Below, we look more closely at their behavior and analyze it mathematically [26].

Often, a processor needs to broadcast data to all processes in a group. MPI provides the broadcast primitive MPI_Bcast to accomplish this. Assume an M-byte message is sent from one node to another, α stands for network latency and β stands for the network communication rate, i.e., the bandwidth is $1/\beta$. The time used is then "$\alpha + M\beta$". If all collective communication functions are implemented using an algorithm based on a binary tree, the MPI_Bcast execution time would be $\log(p) \times (\alpha + M\beta)$. Here, p represents number of group processes.

If an array is scattered throughout all processors in the group, MPI_Gather is used to collect all pieces of the array into a specified process in order of process rank. Conversely, MPI_Scatter is used to distribute data in p equal segments, where the ith segment is sent to the ith process in the group, which has p processes. Since MPI_Gather

Fig. 3 Timing methodologies

```
#include <sys/time.h>
struct timeval t_start, t_end;
long totaltime;
gettimeofday(&t_start, NULL);
     /* code segment here */
gettimeofday(&t_end, NULL);
totaltime = (t_end.tv_sec -
t_start.tv_sec) * 1000000 +
(t_end.tv_usec - t_start.tv_usec);
```

(a) System function: gettimeofday

```
#include <mpi.h>
double t_start = MPI_Wtime();
     /* code segment here */
double t_end = MPI_Wtime();
double totaltime = t_end - t_start;
```

(b) MPI function: MPI_Wtime

```
time mpirun -np 5 ./a.out
```

(c) System command: time

```
#include <time.h>
clock_t t_start, t_end;
t_start = clock();
     /* code segment here */
t_end = clock();
clock_t totaltime = t_end - t_start
```

(d) C function: clock

and MPI_Scatter behave similarly except in data movement direction, we may use MPI_Scatter to evaluate data movement behavior. Assume the root processor owns p messages $m_0, m_1, \ldots, m_{p-1}$, each of size M bytes, and each is sent to Processors $1, 2, \ldots, p-1$. First, Processor 0 sends $m_{p/2}, \ldots, m_{p-1}$ to Processor $p/2$. Next, Processor 0 sends $m_{p/4}, \ldots, m_{p/2-1}$ to Processor $p/4$, and concurrently, Processor $p/2$ sends messages $m_{3p/4}, \ldots, m_{p-1}$ to $3p/4$. The scatter is completed by repeating $\log(p)$ steps. The total number of pieces sent is $(p-1)$, so the execution time is $\alpha \log(p) + (p-1)M\beta$.

MPI_Allgather can be seen as MPI_Gather where all processes receive the result along with the root. It is usually implemented to cyclically shift messages

Table 3 Communication analysis

Function name	Mathematical analysis
MPI_Bcast	$\log(p) \times (\alpha + M\beta)$
MPI_Scatter	$\alpha \log(p) + (p-1)M\beta$
MPI_Allgather	$(p-1) \times (\alpha + M\beta)$

Table 4 Startup time

Parameter	Time (s)	Parameter	Time (s)
-np1	0.51	-np2	0.51
-np3	0.52	-np4	0.53
-np5	0.60	-np6	0.62
-np20	1.78	-np40	3.5

on p processors [26]. It takes a total of $(p-1)$ steps, so the execution time is $(p-1) \times (\alpha + M\beta)$. Table 3 summarizes these analyses.

3.4 Evaluation results

As described above, network latency and bandwidth deeply affect the performance of message-passing systems, so we conducted experiments to measure the startup time and inter-node/intranode transformation overhead for LAM/MPI on our PC Cluster. To measure the performance improvements in using collective communication functions, there are experiments to get the MPI_Send/MPI_Recv to MPI_Bcast, MPI_Scatter, and MPI_Allgather ratios, and a matrix multiplication program is used to evaluate the effects of the computation to communication ratio.

3.4.1 Startup for LAM/MPI on PC cluster

In this case, we started up one node consisting of two processors. The time used is listed in Table 4. This is trivial, only for curiosity. The startup time includes MPI environment and program initialization. Of course, no one spawns processes outnumbering processors since it is redundant.

3.4.2 Network transformation

Table 5 shows execution times and bus to network communication ratios. Ideally, the internode to intranode ratio is *(system bus rate) / (network transformation rate) = (133*8) / (100) = 10.64.* But we see approximately 5, about half that of ideal performance. This is because data is first sent to NIC and then sent back to the system. Given this, threading, rather than message-passing, is used to send messages to processors sharing the same memory. MPI is thread-safe, i.e., implementations made following the MPI standard should ensure correct threading.

Table 5 Network transformation times

Parameters	Execution time (s)				Ratio
	Intra-node		Inter-node		
	-np 2	Average	n0-1	Average	
Data size: 10 MB; Transfer times: 1	0.24	0.24	1.05	1.05	4.3750
Data size: 10 MB; Transfer times: 2	0.43	0.22	2.02	1.01	4.5909
Data size: 10 MB; Transfer times: 4	0.81	0.20	3.97	0.99	4.9500
Data size: 10 MB; Transfer times: 8	1.56	0.20	7.84	0.98	4.9000
Data size: 100 MB; Transfer times: 1	2.38	2.38	10.39	10.39	4.3655
Data size: 100 MB; Transfer times: 2	4.18	2.09	20.16	10.08	4.8230
Data size: 100 MB; Transfer times: 4	7.98	2.00	39.69	9.92	4.9600
Data size: 100 MB; Transfer times: 8	15.52	1.94	78.73	9.84	5.0722

Table 6 Send/receive to broadcast ratios

Parameter	Execution time (s)		
	Point-to-point	Broadcast	Ratio
Data size: 10 MB; processors: 2	0.21	0.21	1.0000
Data size: 10 MB; processors: 4	2.11	2.04	1.0343
Data size: 10 MB; processors: 8	2.12	2.10	1.0095
Data size: 10 MB; processors: 16	5.92	4.62	1.2814
Data size: 100 MB; processors: 2	13.53	8.36	1.6184
Data size: 100 MB; processors: 4	21.33	20.38	1.0466
Data size: 100 MB; processors: 8	59.54	47.00	1.2668
Data size: 100 MB; processors: 16	135.98	89.55	1.5185

3.4.3 Ratios of various communication functions

Here, we report on experiments in two categories: performance ratios for point-to-point to broadcast, and to scatter. As described above, allgather can be viewed as "broadcast after gather", and its performance depends on the combination of these two functions.

Table 6 shows execution times and point-to-point to collective communication ratios for MPI_Bcast. Note that the larger the message and the more processors used, the more obvious the effect of collective communication becomes. The experimental results suggest that broadcasting is especially effective for large groups of processors and large amounts of message transmissions.

Table 7 shows point-to-point to collective communication ratios for MPI_Scatter. According to the analysis above, it has few advantages over point-to-point behavior, and Table 8 shows scatter offers no advantage over point-to-point behavior. Perhaps this is because LAM was implemented with synchronization in some way. Our experiments show scatter provides no advantage over point-to-point collective communication.

Table 7 Send/receive to scatter ratios

Parameter	Execution time (s)		
	Point-to-point	Scatter	Ratio
Data size: 10 MB; processors: 2	0.24	0.24	1.0000
Data size: 10 MB; processors: 4	2.14	2.15	0.9953
Data size: 10 MB; processors: 8	5.94	5.95	0.9983
Data size: 10 MB; processors: 16	13.55	13.56	0.9993
Data size: 100 MB; processors: 2	2.36	2.36	1.0000
Data size: 100 MB; processors: 4	21.51	21.6	0.9958
Data size: 100 MB; processors: 8	59.74	59.86	0.9980

3.4.4 Computation to communication effects

Table 8 shows matrix multiplication execution times rounded to 0.01 second. Note that the execution times for two versions are almost equal since programs have few communication chances. If we repeat our experiment with all-pair-shortest-path, which executes data exchanges during each iteration, the results would be very different.

We evaluated the performance of collective communication functions. The results showed that collective communication has little advantage over point-to-point communication other than in broadcasting, mainly because communication is only a small part of the whole program. Though collective communication is not more effective than point-to-point communication in our case, its coding would be simpler and clearer than that of point-to-point versions. This is very important for our system, which is partly aimed at parallel programming training. To improve performance, we included MPI_Bcast in our system.

4 Design approach

4.1 Problem statement

Most sequential programs spend a major part of their execution time processing loops [27], which is why we focused mainly on loop transformation in our system. Before we describe our system, we use an example to introduce the concepts and explain the goal of our system. There is a loop in Fig. 4a marked by the C comment format directives "DOALL_BEGIN" and "DOALL_END" asserting to our system that iterations may be executed concurrently. The compiler follows the owner-computes rule—the processor that owns the left-hand side of the computation computes it—in generating its corresponding parallel message-passing code. Data distribution is needed for the parallel style of data parallelism.

Each processor computes only the array slices it owns. The compiler assigns the loop partitioning option and the communication model detected by the communication model analyzer to compute the loop iteration upper and lower bounds for each

Table 8 Matrix multiplication execution times

Problem size		Execution time									
		1		2		4		8		16	
		Send/Recv	Collective	Send/Recv	Collective	Send/Recv	Collective	Send/Recv	Collective	Send/Recv	Collective
128	Computation	0.07	0.07	0.03	0.03	0.02	0.02	0.01	0.01	0.01	0.01
	Communication	0.00	0.00	0.00	0.00	0.02	0.02	0.04	0.04	0.09	0.06
256	Computation	1.46	1.72	0.79	1.08	0.53	0.52	0.27	0.31	0.13	0.13
	Communication	0.00	0.00	0.01	0.01	0.08	0.08	0.42	0.16	0.60	0.25
512	Computation	21.48	22.20	12.79	13.22	6.40	6.57	3.21	3.30	1.61	1.68
	Communication	0.00	0.01	0.05	0.05	0.31	0.32	0.73	0.63	1.65	1.03
1024	Computation	176.80	182.95	104.72	107.58	52.35	53.78	26.22	26.94	13.12	13.48
	Communication	0.00	0.04	0.20	0.21	1.29	1.27	2.99	2.51	6.14	4.32

```
/* DOALL_BEGIN */
for (i = 0; i < N; i++){
     A[i] = B[i];
     C[i] = D[i];
}
/* DOALL_END */
```

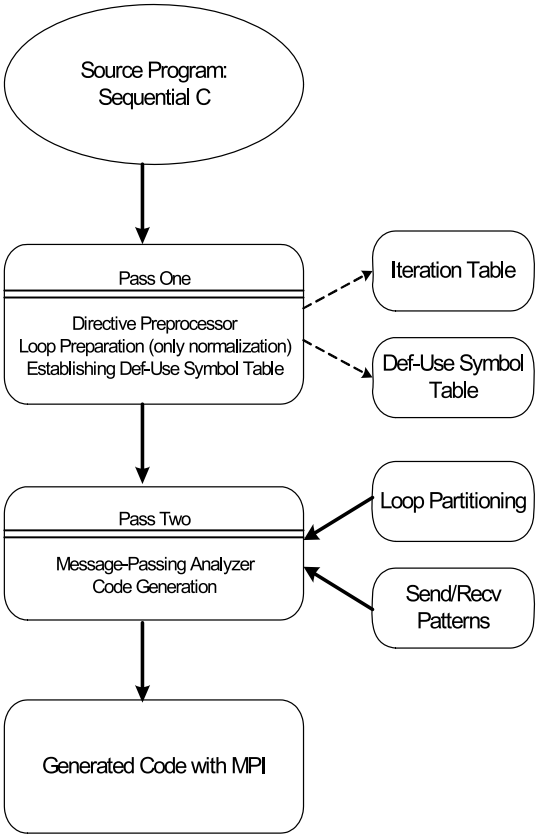
(a) Sequential code

```
(communication: get slice of B)
(communication: get slice of D)
for (i = my_start; i < my_end; i++){
     A[i] = B[i];
     C[i] = D[i];
}
(communication: "collect" A and C)
```

(b) Pseudo parallel code

Fig. 4 Parallel code generated from sample code

Fig. 5 System model



processor. And then compiler will pass necessary data to each processor, as shown in Fig. 4b. The pseudo parallel code control flow structure is identical to its sequential code with the difference being data distribution before and after the loop body. This is the main characteristic of message-passing programming. Put more simply, it is a true parallelizing compiler's front end focused mainly on loops. Its chief function is translating parallelizable loops into their parallel forms with correct data distribution.

Fig. 6 A pseudo source code

```
int main (int argc, char* argv[]){
  Variable declaration area
  …
  /* INIT_BEGIN */
    Initialization for all nodes
  /* INIT_END */
  …
  /* DOALL_BEGIN P=CYC */
    Loop Blocks that will be parallelized
  /* DOALL_END */
  …
  return (0);
}
```

Table 9 Directives for parallelism

Name:	$/*\ DOALL\_BEGIN\,P = XXX\ */$
Usage:	Tell the system to parallelize the following loop block. Character P stands for loop partitioning option. The system now supports only static partitioning: BLK for block, CYC for cyclic. CSS, GSS, FSS, TSS are reserved for future versions.
Name:	*/* DOALL_END */*
Usage:	Enclose the block parallelized with respect to DOALL_BEGIN.
Name:	*/* INIT_BEGIN */*
Usage:	Tell the system this block will be initialized for all nodes.
Name:	*/* INIT_END */*
Usage:	Enclose the block initialized with respect to INIT_BEGIN.
Name:	*/* SYN */*
Usage:	Tell the system to synchronize. Synchronization is done automatically done for parallelizable loops, so this directive works only when a loop is not parallelized, but a user wants synchronization for safety.
Name:	/* $DD\ par(X, X\ldots)$ */
Usage:	Tell the system the data distance vector for array variable par. This directive is used when the communication model is semi-global.

4.2 System model

Our system is a *directive-based MPI code generator* that translates sequential source code into parallel coding; Fig. 5 shows a skeleton diagram of our system. The sample source program shown in Fig. 6 was fed into our system. It is a sequential C program with the comment format directives listed in Table 9 as examples of source code. The implementation is based on these assumptions:

- The loop is expressed by a *for* statement: There is much syntax for loop expression. Some technologies format *while* loops as *for* loops [28], but for simplicity parallelizable loops should be expressed as *for* statements.

```
/* DoALL_BEGIN */
/* P = CYC */                    /* DOLL_BEGIN P=CYC */
/* SYN */
(a) Multi-line directives   (b) Preprocessed directive
```

Fig. 7 Directive preprocessing

- Parallelizable loops in loop nests are made the outermost loops. Generally, synchronization costs for parallelized inner loops are higher on each outermost loop iteration.
- Pointers should be excluded from parallelizable loops. First, because analyzing pointer structures is difficult. Second, MPI does not support this data structure type for communication functions, since implementing pointer-pass messages in general forms is difficult.
- For loop normalization and unrolling iteration spaces, the boundaries of each loop are static and known at compile time. For this reason, array index functions should be linear functions of the index variables.

Our system is a two-pass technology. After source code is input, it is preprocessed in Pass One to format the source program and extract directives. Parallelizable loops are normalized and each is assigned a unique number as an identifier. Iteration information is extracted from the normalized loops and stored in the iteration table built for loop partitioning. In this phase, semantics are parsed and the def-use symbol table is established for further analyses.

In Pass Two, the message-passing analyzer examines parallelizable loops according to the def-use symbol table established in Pass One, and eventually generates MPI coding. More details on each phase are discussed below.

4.3 Pass one

The main work of Pass One is to scan the source program and extract information for Pass Two. It consists of three main tasks: directive processing, loop preparation, and establishing a def-use symbol table. Two tables are built: the iteration table for loop partitioning and the def-use symbol table for the message-passing analyzer.

4.3.1 Directive preprocessor

The directive preprocessor formats directives and loops and corrects their syntax errors. This may seem trivial, but it is necessary for future parsing. The preprocessor combines all directive lines, if more than one, into one-line directives. Of course, some spelling errors are also corrected, and redundant synchronizations of parallelizable loops omitted. In the example given in Fig. 7a, the synchronization in the third line is redundant because the loop will be synchronized automatically after parallelization. Figure 7b shows the formatted and corrected directive.

Fig. 8 For loop preprocessing

```
for (i = 2; i <= 100; i += 5)
```

```
for (i = 2; i <= 100; i = i + 5)
```

(a) Formatting for loop

```
#define CONST_NUMBER 200
for (i = 0; i <= CONST_NUMBER; i++)
```

```
for (i = 0; i <= 200; i = i + 1)
```

(b) Formatting for loop with constant substitution

4.3.2 Loop preparation

The preprocessor substitutes constants and formats loop iterations, especially for statement incremental expressions. Figure 8a shows incremental expression formatting, and Fig. 8b formatting for incremental expressions and constant substitution.

Loop optimization encompasses many techniques, e.g., redundancy elimination, normalization, reordering, fusion, etc. A detailed list of loop transformations is given in [8]. We implemented loop normalization for easy and convenient subscript analysis. Other transformations could be added to the loop preparation phase for further optimization.

Loop normalization [31] converts all loops such that the induction variable is initially 0 (1 in Fortran) and is incremented by 1 during each iteration. This ensures that the loop iteration space is regular. The transformation (algorithm shown in Fig. 9) is very simple. Only two actions in iteration space are necessary: shifting the induction variable to 0 and scaling the distance of each item to 1. After the boundaries have been set, the original index is recomputed to its original value from the new index during each iteration. The normalization produces two assignment statements on the initial loop index. The first, at the beginning of the loop body, assigns its new index value function to it, and the second, at the end of the loop, assigns its final value to it. The loop body remains unchanged.

An iteration table is built during this phase. With loop normalization, loop control variables are inserted into the table. Each item consists of a variable name, upper boundary, and a unique loop identification number.

4.3.3 Def-use symbol table

Our system is block-oriented. Source programs are separated into blocks according to DOALL loops bracketed by DOALL_BEGIN and DOALL_END directives. Thus, DOALL loops are break points. Each block is indexed by a global counter that starts at 1 and increases by 1 during each iteration. A def-use symbol table based on this is established to analyze message-passing behavior. Each item in the table consists of three fields: name tuple, def-chain tuple, and use-chain tuple. Table 10 gives definitions for each of these.

```
for (i = lower; i <= upper; i = i + incre){
     Loop_Body;
}
```

(a) Unnormalized

```
for (NI=0; NI < (upper-lower+ incre)/incre; NI++){
     i = lower + incre * NI;
     Loop_Body;
}
i=lower + incre * MAX(0, (upper-lower+incre)/incre);
```

(b) Normalized

Fig. 9 Loop normalization algorithm

Table 10 Def-use symbol definitions

Tuple name	Definition
name(N, A)	N: variable name
	A: $\begin{cases} 1, \text{if } n \text{ is an array} \\ 0, \text{otherwise} \end{cases}$
def-chain(B, D, S, OP)	B: block index
	D: $\begin{cases} 1, \text{if inside DOALL block} \\ 0, \text{otherwise} \end{cases}$
	S: sequence order
	OP: self-reference on OP operation
use-chain(B, D, S)	B: block index
	D: $\begin{cases} 1, \text{if inside DOALL block} \\ 0, \text{otherwise} \end{cases}$
	S: sequence order

Table 11 OP table

Operation	OP value	Operation	OP value	Operation	OP value
None	0	$*$	4	$>=$	8
Initialization	1	/	5	$<$	9
$+$	2	%	6	$<=$	10
$-$	3	$>$	7		

The two parameters shown in Table 11, S and OP, must be described more fully. S is sequence order according to statement order and assists in determining the order of write and read operations. OP represents operations on variables. That is, in some instances or during some operations, a variable may appear on both left-hand and right-hand sides. OP is considered and reperformed right after a DOLL loop to "collect" data from all nodes.

Some rules are required to maintain the def-use symbol table. Given variable η

```
     /* INIT_BEGIN */
S1   initialMatrix(c)                                  Block 1
     /* INIT_END */
-------------------------------------------------------------
     /* DOALL_BEGIN */
S2   for (i = 0; i < M; i++)
S3      for (j = 0; j < N; j++)                        Block 2
S4         for (k = 0; k < P; k++)
S5            c[i][j] = c[i][j] + a[i][k]*b[k][j];
     /* DOALL_END */
-------------------------------------------------------------
S6   printMatrix(c);                                   Block 3
```

Fig. 10 Matrix multiplication code segment

Table 12 Def-use symbol table for the example code

Def-use field	Name field		
	(c, 1)	(a, 1)	(b, 1)
Def-chain	(1, 0, 1, 1)		
	(2, 1, 2, 2)		
Use-chain	(2, 1, 2)	(2, 1, 1)	(2, 1, 1)
	(3, 0, 3)		

- if η is new to the table, create a new item in the table, the tuple field name is (N, A);
- if η is defined (write to η), add (B, D, S, OP) to the def-chain field (here, OP indicates whether variable η is self-referenced);
- if η is used (read from η), add (B, D, S) to the use-chain field.

For example, the matrix multiplication code segment shown in Fig. 10 has three blocks. When S_1 is parsed, Array C is new and defined, so a new item is created with the name field (c, 1). Because S_1 is for variable initialization, the def-chain of (c, 1) is (1, 0, 1, 1), meaning that Array C is in block 1, which is not a DOALL loop, its sequence order is 1, and the variable is initialized for all nodes. Next comes a DOALL loop. Iteration variables are first parsed and the information recorded in the iteration table early in the loop preparation phase. In S_6, the block index is 2 and the sequence order is 2. Array c has self-reference on add operation and so (2, 1, 2, 2) is linked to the (c, 1) def-chain. After parsing this code segment from S_1 to S_6, the def-chain symbol table shown in Table 12 is built.

4.4 Pass two

4.4.1 Message-passing analyzer

The analyzer checks the def-use symbol table for message-passing behavior and further communication models, performing two analyses: *read-write relation analysis* and *data space analysis*. The read-write relation affects the demand and direction of data movement from the master to slaves (message in: into the loop) and from slaves

Table 13 MPI functions used in our system

Function name	Functionality
MPI_Init	Start up environment for MPI
MPI_Finalize	Shut down MPI
MPI_Comm_rank	Return the rank of calling process
MPI_Comm_size	Return the size of communicator relative to calling process
MPI_Send	Send data to destination process
MPI_Recv	Receive data sent by source process
MPI_Bcast	Send data to every process

to the master (message out: out of the loop). "Message in" means a parallelizable loop requires data assigned before the loop; "message out" means at least one statement requires data assigned in a parallelizable loop.

If there is a use-chain tuple in a DOALL block, it uses data in the previous block. This means a message-in from the previous block nearest to the current block is required. If the use-chain tuple belongs to an initialization block, the message-in action is redundant and is not performed. Variable (c, 1) in Table 13 has a use-chain tuple (2, 1, 2) and a def-chain tuple (1, 0, 1, 1), and so, message-in action is required, but the OP field of this def-chain tuple is 1, meaning initialization, so the read-write relation is ignored.

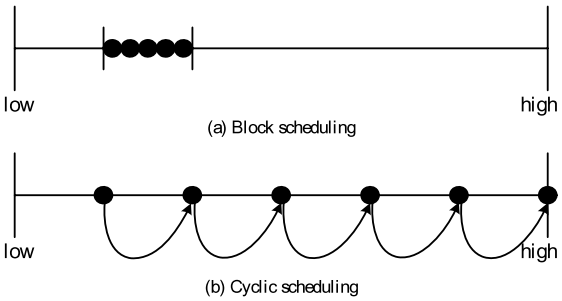
A def-chain tuple in a DOALL block affects later statements that read the variable. Only when following statements read variables assigned in a previous DOALL loop is message-out action required. For example, the variable (c, 1) in Table 13 has the use-chain tuple (3, 0, 3) and the def-chain tuple (2, 1, 2, 2), so message-out action is required. Sometimes temporal variables are used in DOALL loops and never used after the DOALL loops, i.e., they have def-chain tuples, but no corresponding use-chain tuple. In such cases, the read-write relation is also ignored.

The second analysis, data-space analysis, detects whether an update algorithm in one node requires data from others. In other words, it recognizes a communication model for each parallelizable loop. If a variable data type is not array, other nodes during other iterations will never reference the variable. It is independent. If a variable data type is array, the system unrolls array subscripts, and the iteration space, checking to see if they are intersecting. If the data space (array subscript space) matches the iteration space, it is *Independent*. If not, it may be *Global* or *Semi-Global*. If the data space interests "all other's" iteration space, it is *Global*; otherwise, it is *Semi-Global*. For example, Floyd's algorithm for all-pairs shortest-path stores all paths in a square matrix and represents the cost from A to B to C as the summation of path_matrix[A][B] and path_matrix[B][C]. Here, because A space is partitioned, B will outstrip the first dimension of the path_matrix array. The algorithm must update its current path cost during each iteration. In this case, B intersecting all nodes, so it is *Global*. In our system, we unroll data space directly and check to see if it locates on the iteration spaces of other nodes, however, this may be inefficient for large spaces. We will solve this problem by using libraries like Omega data dependence test in the near future.

Fig. 11 Pattern of code generation

```
Loop_partition_function
Communication (master to slaves)
/* DOALL_BEGIN */
   ...
   Communication if the model of this DOALL loop is Global
/* DOALL_END */
Communication (slaves to master)
Some operations for self-referenced variables
```

Fig. 12 Static partitioning



4.4.2 Code generation

Code generation is also block-oriented, non-DOALL and noninitialization blocks, excluding variable declaration, belong only to the master. Other parts of source programs belong to both masters and slaves. Parts of the master will be enclosed by if (adppg_rank == 0) control flows in cooperation with an error-handling mechanism that ensures all processes exit at the same time when one or more errors occur.

After message-passing analysis, code generation focuses on DOALL loops. The code generated for DOALL loops is shown in Fig. 11. Control flows are not changed, but loop partitioning and data movement do occur. A loop partitioning function that partitions the iteration space for each node comes first. We use block and cyclic static partitioning, described in Sect. 2. Figure 12 shows schematically how the loop partitioning function partitions the iteration space. Before and after DOALL loops, there is communication for data movement from the master to slaves and from slaves to the master.

Communication is also necessary during each iteration if the communication model inside a parallelizable loop is *Global*. After a self-referenced variable with a slaves-to-master data movement direction has been sent, the operation stored in the def-chain for this variable is performed in order to "collect" all data from slaves.

Table 13 shows the MPI functions used in our system. Only six primitives are required to make an MPI program. According to the evaluation in Sect. 3, using broadcasting is worthwhile, so we include MPI_Bcast. Collective communication allows Global communication mode programs to exchange data for all processes.

Taking the matrix multiplication in Fig. 10 as an example, S_1 is inside an initialization block, thus, it belongs to all nodes, and remains unchanged; for all nodes each variable inside an initialization block holds its value. Next is a loop-partitioning function and in this case the option is block. According to the message-passing an-

```
initialMatrix(c);

loop_partitioning_function (BLK)
communication: send array a and b to slaves
for (i = it.begin; i < it.end; i = i + it.step)
    for (j = 0; j < N; j++)
        for (k = 0; k < P; k++)
            adppg_c[i][j] = adppg_c[i][j] + a[i][k]*b[k][j];
communication: receive array adppg_c from slaves
c = the summation of adppg_c from all slaves

if (adppg_my_rank == 0){
    printMatrix(c);
}
```

Fig. 13 Parallel code generated for matrix multiplication

alyzer, Array A and Array B must be sent to slaves. Because the INIT_BEGIN and INIT_END directives enclose Array C, all nodes must execute the statements in this initialization block, i.e., the Array C data movement is unnecessary. If the user had not used the INIT_BEGIN and INIT_END directives, the Array C data movement would have been generated and been redundant. The loop control variable is substituted accompanying the loop-partitioning function.

Self-referenced variables will be repeated by some operation stored in the def-chain OP field right after the loop, so they are prefixed by "adppg_" inside the loop to make the distinction. The Array C def-chain in this block is (2, 1, 2, 2). From the OP table, 2 means take the summation of itself. After sending adppg_c from all slaves to the master, Array C repeats summation of adppg_c from all slaves to "collect" its real value. Figure 13 shows the generated code corresponding to the sequential code shown in Fig. 10.

5 Experimental environment and results

5.1 System environment

Our SMPs cluster is a low-cost Beowulf-class system that utilizes a multicomputer architecture for parallel computation. Our cluster consists of 16 PC-based symmetric multiprocessors (SMP) connected, for channel bounding, by two 24-port 100 Mbps Ethernet SuperStackII 3300 XM switches with fast Ethernet interface. There is one server node and fifteen computing nodes. The server node has two Intel Pentium-III 1 GHz (FSB 133 MHz) processors and 768 MBytes of shared local memory. Each Pentium-III has 32K on-chip instruction and data caches (L1 cache), and a 256K on-chip four-way second-level cache with full CPU speed. Each P-III-based computing node has two 1G P-III processors has 512 MBytes of shared local memory. We ran Redhat Linux 7.2 as our operating system and use Lam/MPI implementations for message passing.

Table 14 Matrix multiplication execution times

Processors		Problem size		
		256 × 256	512 × 512	1024 × 1024
Sequential	1	1.494	20.819	170.530
Our system	2	1.456	13.088	102.410
	4	1.382	7.616	55.337
	8	1.166	4.989	31.490
	16	1.355	6.223	25.161
Hand-revised	2	1.515	12.136	101.545
	4	1.326	7.265	56.811
	8	1.136	4.893	31.352
	16	1.195	5.596	22.891

Table 15 Prime number detection execution times

Processors		Problem size		
		1,000,000	10,000,000	100,000,000
Sequential	1	1.178	29.566	780.085
Our system	2	1.128	15.552	393.220
	4	0.845	8.258	202.569
	8	0.719	4.407	101.564
	16	0.646	2.508	51.096
Hand-revised	2	1.128	15.619	393.854
	4	0.837	8.273	202.532
	8	0.724	4.405	101.579
	16	0.645	2.508	51.086

5.2 Applications

Four study cases were assessed for correctness and performance. The first three were: matrix multiplication, prime number detection, and Mandelbrot set. While they are all "independent" communication models, they behave differently from one another. Since they are independent, processors do not have to communicate with one another while computing. The last study case is all-pairs shortest path. It is a "Global" communication model. Each processor must access data from all other processors to update its own data. For each case, we used three program sequencing, versions generated by our system and hand-revised. Experiments were conducted with various numbers of iteration using various numbers of processors. A comparison between using our system and hand-revision is also given.

Execution times for the four applications: matrix multiplication, prime number detection, Mandelbrot set, and all-pairs shortest path, are shown in Tables 14, 15, 16, and 17, respectively. Speedups are shown in Figs. 14 and 15.

Table 16 Mandebrot set execution times

Processors		Problem size		
		Iteration: 1000 Grid: 1024	Iteration: 1000 Grid: 2048	Iteration: 1000 Grid: 4096
Sequential	1	4.945	19.764	79.037
Our system	2	3.425	11.057	41.152
	4	3.728	11.723	42.829
	8	3.289	9.919	35.639
	16	3.709	6.256	21.720
Hand-revised	2	3.070	10.739	43.301
	4	3.304	11.500	44.868
	8	2.924	10.016	38.153
	16	2.114	6.588	26.913

Table 17 All-pairs shortest path execution times

Processors		Problem size		
		512	1024	2048
Sequential	1	5.266	41.422	329.319
Our system	2	3.580	24.230	186.800
	4	2.649	15.233	107.029
	8	2.127	10.676	65.412
	16	2.600	10.079	53.090
Hand-revised	2	3.584	1.716	187.210
	4	2.430	2.866	103.768
	8	1.936	4.216	62.204
	16	2.192	4.684	46.731

5.3 Comparison

Experimental results show that hand-revised optimized codes performed more efficiently than codes generated by our system. There are two main reasons for the difference. First, the master block error-handling mechanism reduces the performance. Second, as is widely known, collective communication generally performs better than point-to-point communication, but in our system, it generates codes using point-to-point behavior. These findings will be taken into consideration for future optimization versions.

As indicated by the experimental results above, we compared our Directive-based MPI Code Generator (DMCG) to hand-revised optimization. The comparison is summarized in Table 18.

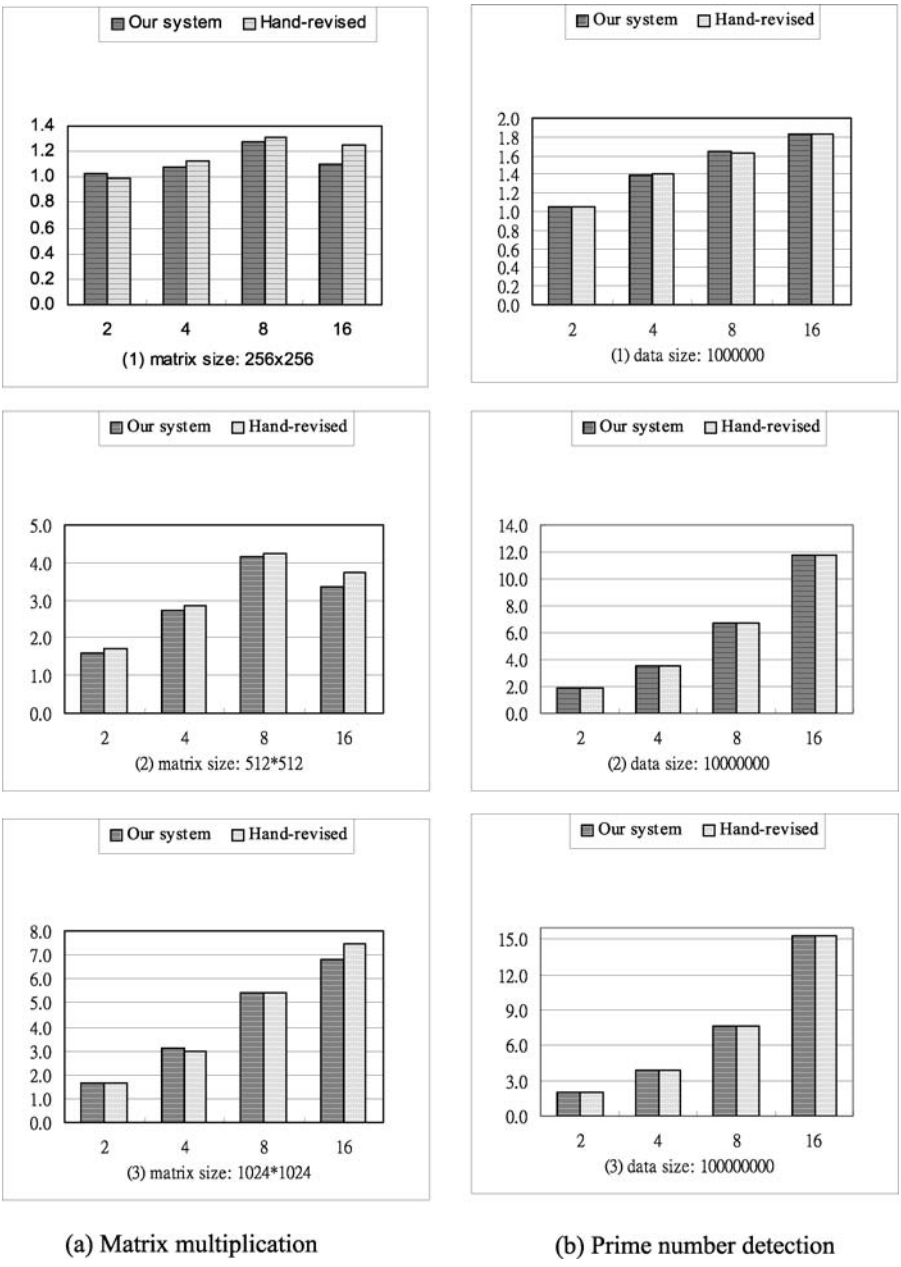


Fig. 14 Matrix multiplication and prime number detection speedups

6 Conclusions and future work

We developed the Directive-based MPI Code Generator (DMCG) for translating sequential C source code into parallel code using C language with MPI. We provided

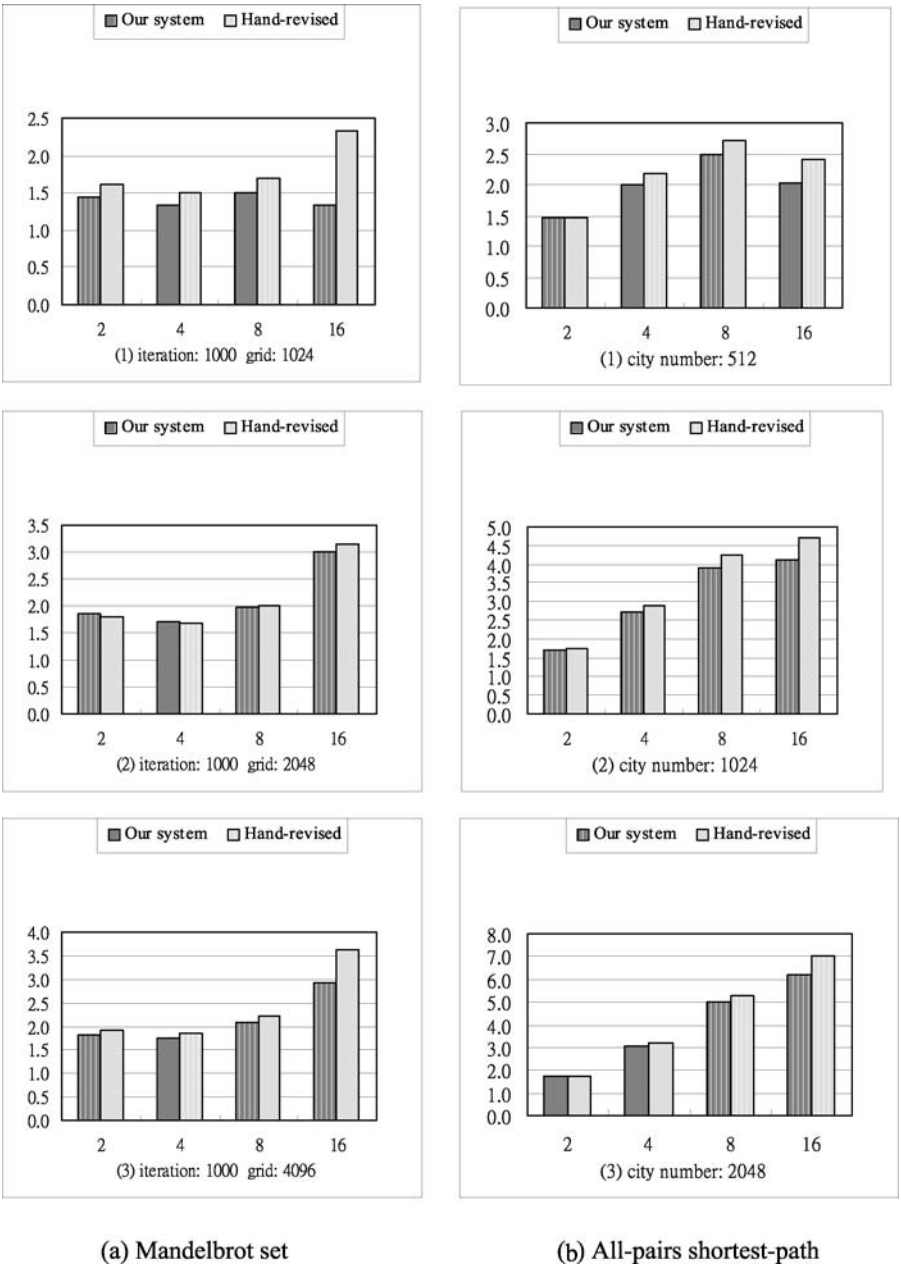


Fig. 15 Mandelbrot set and all-pairs shortest-path speedups

a useful learning tool to aid beginners to parallel programming with MPI. With our system, users can generate parallel codes from sequential codes and can look closely at the relationship between sequential and parallel coding. They can also learn how

Table 18 Comparison of our system and hand-revised approach

Approach	Time/Effort	Performance	Applicability
Our system	Annotation required: parallelism directives and scheduling parameter methods for performance	Depends completely on program communication model; it is excellent if model is independent and user tunes the code well	Cannot handle structure, pointer, indirect array reference, and loop-carried dependence
Hand-revised	Requires extensive code modification; time-consuming and error-prone	Excellent when implementation is adjusted to problems and optimized to parallel environments	Applicable to any code

to implement loop scheduling. Since the generated parallel codes perform nearly as well as optimized codes, it is a good tool for speeding up solution steps and porting current applications to parallel architectures with MPI implementation.

In the near future, we will reimplement our system, using the Stanford University Intermediate Format (SUIF) [29] as a platform, to make it more flexible and suitable for collaborative development. Of course, since it is a learning tool for beginners to parallel programming, the graphical interface will be clarified and made friendlier. It will show the relationship between sequential code and parallel code, as well as loop transformation processing.

Other future projects relate to performance. One is introducing dynamic scheduling into our system to give users more tuning options in adjusting generated codes to their environments (homogeneous and heterogeneous). It will be a long road to realizing this. The other concerns loop and communication optimization. There are many approaches to loop optimization, e.g., redundancy elimination, loop reordering, loop fusion, etc. These approaches can be added to the loop preparation phase, as described in the design approach. For communication optimization, we will improve send/receive behaviors for various communication models and use the technology described in [30] to reconstruct communication behavior.

References

1. Sterling TL, Salmon J, Becker DJ, Savarese DF (1999) How to build a Beowulf: a guide to the implementation and application of PC clusters, 2nd edn. MIT Press, Cambridge
2. Wilkinson B, Allen M (1999) Parallel programming: techniques and applications using networked workstations and parallel computers. Prentice Hall, New York
3. Buyya R (1999) High performance cluster computing: architectures and systems, vol. 1. Prentice Hall, New York
4. Message passing interface forum. http://www.mpi-forum.org/
5. PVM—parallel virtual machine. http://www.epm.ornl.gov/pvm/
6. TOP500 supercomputer sites. http://www.top500.org
7. Wolfe M (1996) Parallelizing compilers. ACM Comput Surv 28(1):261–262
8. Wolfe M (1996) High performance compilers for parallel computing. Addison-Wesley, Reading
9. Boulet P, Darte A, Silber G-A, Vivien F (1998) Loop parallelization algorithms: from parallelism extraction to code generation. Parallel Comput 24:421–444
10. Banerjee U (1988) An introduction to a formal theory of dependence analysis. J Supercomput 2(2):133–149

11. Wolfe M (1989) More iteration space tiling. In: Proceedings of supercomputing, pp 655–664
12. Bacon DF et al (1994) Compiler transformations for high-performance computing. ACM Comput Surv 26(4):245–320
13. Yang CT, Tseng SS, Fan YW, Tsai TK, Hsieh MH, Wu CT (2001) Using knowledge-based systems for research on portable parallelizing compilers. Concurr Comput Pract Exper 13:181–208
14. Hummel SF, Schonberg E, Flynn LE (1992) Factoring: a method for scheduling parallel loops. Commun ACM 35(8):90–101
15. Kruskal CP, Weiss A (1985) Allocating independent subtasks on parallel processors. IEEE Trans Softw Eng 11(10):1001–1016
16. Polychronopoulos CD, Kuck DJ (1987) Guided self-scheduling: a practical self-scheduling scheme for parallel supercomputers. IEEE Trans Comput 36(12):1425–1439
17. Tzen TH, Ni LM (1993) Trapezoid self-scheduling: a practical scheduling scheme for parallel compilers. IEEE Trans Parallel Distrib Syst 4(1):87–98
18. Tang P, Yew PC (1986) Processor self-scheduling for multiple-nested parallel loops. In: International conference on parallel processing, pp 528–535
19. Li H, Tandri S, Stumm M, Sevcik KC (1993) Locality and loop scheduling on NUMA multiprocessors. In: International conference on parallel processing, vol II, pp 140–147
20. LAM/MPI parallel computing. http://www.lam-mpi.org/
21. MPICH—a portable implementation of MPI. http://www-unix.mcs.anl.gov/mpi/mpich/
22. MPI software technology. http://www.mpi-softtech.com/
23. McGarvey B, Cicconetti R, Bushyager N, Dalton E, Tentzeris M (2001) Beowulf cluster design for scientific PDE models. In: Proceedings of the 2001 annual Linux showcase, Oakland, CA, November 2001
24. Sedgewick R (1992) Algorithms in C++. Addison-Wesley, Reading, pp 476–478
25. Gorlatch S (2002) Message passing without send–receive. Future Gener Comput Syst 18:797–805
26. Luecke GR, Raffin B, Coyle JJ (1999) The performance of the MPI collective communication routines for large messages on the Cray T3E600, the Cray Origin 2000, and the IBM SP. J Perform Eval Model Comput Syst, July 1999
27. Beletsky V, Bagaterenco A, Chemeris A (1995) A package for automatic parallelization of serial C-programs for distributed systems. In: Proceedings of the conference on programming models for massively parallel computers, pp 184–188
28. Zhang F, D'Hollander EH (1994) Extracting the parallelism in programs with unstructured control statements. In: Proceedings of international conference on parallel and distributed systems. IEEE, New York, pp 264–270
29. The Stanford SUIF compiler group. http://suif.stanford.edu
30. Di Martino B, Mazzeo A, Mazzoccaa N, Villano U (2001) Parallel program analysis and restructuring by detection of point-to-point interaction patterns and their transformation into collective communication constructs. Sci Comput Program 40:235–263
31. Allen JR, Kennedy K (1987) Automatic translation of Fortran programs to vector form. ACM Trans Program Lang Syst 9(4):491–542

Chao-Tung Yang is a professor of computer science at Tunghai University in Taiwan. He received a B.S. degree in computer science from Tunghai University, Taichung, Taiwan, in 1990, and the M.S. degree in computer and information science from the National Chiao Tung University, Hsinchu, Taiwan, in 1992. He received the Ph.D. degree in computer and information science from National Chiao Tung University in July 1996. He won the 1996 Acer Dragon Award for an outstanding Ph.D. dissertation. He has worked as an associate researcher for ground operations in the ROCSAT Ground System Section (RGS) of the National Space Program Office (NSPO) in Hsinchu Science-based Industrial Park since 1996. In August 2001, he joined the faculty of the Department of Computer Science at Tunghai University. He got the excellent research award by Tunghai University in 2007. In 2007 and 2008, he got the Golden Penguin Award by the Industrial Development Bureau, Ministry of Economic Affairs, Taiwan. His researches have been sponsored by Taiwan agencies National Science Council (NSC), National Center for High Performance

Computing (NCHC), and the Ministry of Education. His present research interests are in grid and cluster computing, parallel and high-performance computing, and internet-based applications. He is both a member of the IEEE Computer Society and ACM.

Kuan-Chou Lai received his M.S. degree in computer science and information engineering from the National Cheng Kung University in 1991, and the Ph.D. degree in computer science and information engineering from the National Chiao Tung University in 1996. Currently, he is an associate professor in the Department of Computer and Information Science and the director of the Computer and Network Center at the National Taichung University. His research interests include parallel processing, heterogeneous computing, system architecture, P2P, grid computing, and multimedia systems. He is a member of the IEEE and the IEEE Computer Society.