Using a Low-Cost PC Cluster for High-Performance Computing

使用低價個人電腦叢集於高性能計算

Chao-Tung Yang
楊朝棟
Dept. of Computer Science and Information Engineering
Tunghai University
181 Taichung-kang Road, Sec. 3
Taichung, 407, Taiwan
東海大學資訊工程與科學系
email: ctyang@mail.thu.edu.tw
Tel: +886-4-23590121 ext. 3279

Chi-Chu Hung
洪基祝
ROCSAT Ground System Section
National Space Program Office
8F, 9 Prosperity 1st Road, Science-based Indu. Park
Hsinchu, 300, Taiwan, R.O.C.
行政院國家太空計畫室
email: om21@nspo.gov.tw
Tel: +886-3-5784208 ext. 1181

Abstract

Clustering computing has become the paradigm of choices for executing large-scale science, engineering, and commercial applications. This is due to their low cost, high performance, high availability of off-the-shelf hardware components and freely accessible software tools that can be used for developing applications. In this paper, we will introduce the clustering system and discuss system architecture, software tools, and applications of this system.

Keywords: Clustering, SMP, Parallel Computing, Speedup, PVM, MPI

摘要

利用傳統電腦來解決複雜的科學及工程問題已不敷使用，唯有藉助平行處理的方式才能達到此一目的。最近，運用叢集式電腦系統搭配 Linux **作業系統與** PVM **或** MPI **訊息傳遞程式庫，來執行高速計算(或平行計算)已經逐漸走到實際可行的階段。即使用者可以花較少的費用與時間就能建置叢集式的平行電腦系統，將平行計算應用到其專業領域，而得到不錯的結果。本論文所陳述的叢集式平行系統架構，軟體工具以及相關應用。**

關鍵詞：叢集式、平行計算、對稱式多處理機系統、訊息傳遞程式庫

1 Introduction

While the use of parallel supercomputers has been one of mainstreams for high-performance computing for the past decade, its popularity is waning. The reasons for this decline are many. They include factors such as being expensive to own and maintain; slow to evolve along with emerging hardware technologies; and difficult to upgrade without totally being replaced by a new system. Extraordinary technological improvements over the past few years in areas such as microprocessors, memory, buses, networks, and software have made it possible to assemble groups of inexpensive personal computers and/or workstations into a cost effective system that functions in concert and posses tremendous processing power rivaling supercomputers. Cluster computing is not new, but in company with other technical capabilities, particularly in the area of networking, this class of machines is becoming a high-performance platform for parallel and distributed applications [1, 2, 10, 11].

Scalable computing clusters, ranging from a cluster of (homogeneous or heterogeneous) PCs or workstations, to SMPs (Symmetric MultiProcessors), as shown in Figure 1 (a), are rapidly becoming the standard platforms for high-performance and large-scale computing. A cluster, as shown in Figure 1 (b), is a group of independent computer systems and thus forms a loosely coupled multiprocessor system. A network is used to provide inter-processor communications. Applications that are distributed across the processors of the cluster use either message passing or network shared memory for communication. Cluster nodes work collectively as a single computing resource and fill the conventional role of using each node as an independent machine. A cluster computing system is a compromise between a massively parallel processing system and a distributed system. An MPP (Massively Parallel Processors) system node typically cannot serve as a standalone computer; a cluster node usually contains its own disk and equipped with a complete operating systems, and therefore, it also can handle interactive jobs. In a distributed system, each node can function only as an individual resource while a cluster system presents itself as a single system to the user.

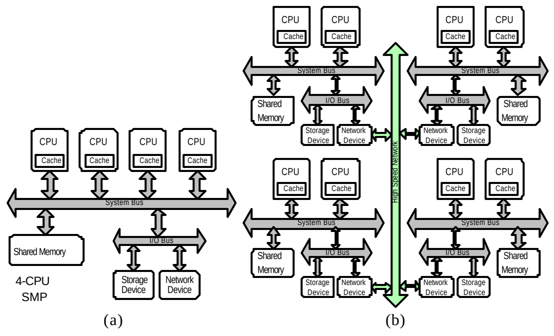


Figure 1: (a) The structure of a typical SMP with four processors. (b) A typical cluster system with eight processors.

The concept of Beowulf clusters originated at the Center of Excellence in Space Data and Information Sciences (CESDIS), located at the NASA Goddard Space Flight Center in Maryland. The goal of building a Beowulf cluster is to create a cost-effective parallel computing system from mass-market commodity, off-the-shelf components to satisfy specific computational requirements in the earth and space sciences community. The first Beowulf cluster was built from 16 Intel DX4TM processors connected by a channel-bonded 10 Mbps Ethernet, and it ran the Linux operating system [10]. It was an instant success, demonstrating the concept of using a commodity cluster as an alternative choice for high-performance computing (HPC). After the success of the first Beowulf cluster, several more were built by CESDIS using several generations and families of processors and network interconnects.

Such a system can provide a cost-effective way to achieve features and benefits (fast and reliable services) that have historically been found only on more expensive proprietary shared memory systems. The main attractiveness of such system is that they are built using affordable, low-cost, commodity hardware (such as Pentium PCs), fast LAN such as Myrinet, and standard software computers such as UNIX, Linux, and Solaris. Applications are often parallelized by using the MPI or PVM message-passing library for inter-processor communications [4, 6]. These systems are scalable, i.e., they can be tuned according to user's available budget and computational needs and allow efficient execution of both demanding sequential and parallel applications. To utilize the resources of a clustering system, a problem had to be algorithmically expressed as comprising a set of concurrently executing sub-problems or tasks [1, 2, 11, 12, 13, 14]. This was achieved via the use of parallel programming, libraries, and/or environments that encapsulate the way in which the various tasks cooperate to provide the solution to the original problem [12, 13, 14].

We conducted and maintained an experimental Linux SMP cluster (SMP PC machines running the Linux operating system), which is available as a computing resource for test users. This cluster is made up of 16 PC-based SMPs. Nodes are connected using Fast Ethernet with a maximum bandwidth of 200/100Mbits with/without channel bonded, through two 3Com 24-port switches. This cluster machine is operated as a unit, sharing networking, file servers, and other peripherals. This cluster is used to run both serial and parallel jobs.

In this paper, the system architecture, software tools, and applications of this cluster system will be discussed. The matrix multiplication and parallel ray tracing problems are illustrated and the experimental results are also demonstrated on our Linux SMPs cluster. Eight 2-Celeron-processor SMPs and eight 2-PIII-processor SMPs are connected as a cluster to measure the system performance on speedup. The experimental results show that the highest speedups are 10.85 and 15.22, respectively for matrix multiplication and PVMPOV [5], when the total number of processor is 16 on SMPs cluster. Two benchmarks, LU of NPB and HPL, are also used to demonstrate the performance of our testbed by using LAM/MPI library. The experimental results show that our cluster can obtain 6.433Gflop/s and 5.784Gflop/s for HPL, when the total number of processors used is 16 with and without channel bonding. The results of this study will make theoretical and technical contributions to the design of a high-performance computing system on a Linux SMP Clusters.

2 System Overview

2.1. Hardware

Our SMPs cluster is a low cost Beowulf class supercomputer that utilizes a multi-computer architecture for parallel computations. The Parallel Testbed consists of two PC clusters. One is used for parallel computing, the other is used for high available application. For parallel computation portion, the snapshot of our cluster is shown in Figure 2 that consists of 16 PC-based symmetric multiprocessors (SMP) connected by two 24-port 100Mbps Ethernet SuperStackII 3300 XM switches with Fast Ethernet interface. Its system architecture is shown in Figure 3.

There are one server node and fifteen computing nodes. The server node has two Intel Pentium-III 945MHz (750 over-clock, FSB 126MHz) processors and 768MBytes of shared local memory. Each Pentium-III has 32K on-chip instruction and data caches (L1 cache), a 256K on-chip four-way second-level cache with full speed of CPU. There are two kinds of computing nodes, one is P-III-based, and the other is Celeron-based. Each P-III-based computing node with two 945 P-III processors has 512MBytes of shared local memory. Each Celeron-based computing node with two Celeron processors has 384MBytes of shared local memory. Each Celeron also has 32K on-chip instruction and data caches (L1 cache), a 128K on-chip four-way second-level cache with full speed of CPU. Each individual processor is rated at 495MHz, and the system bus has a clock rate of 110 MHz.

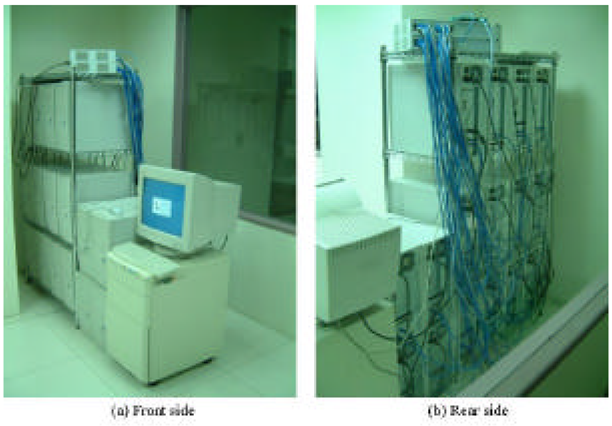


Figure 2: The snapshot of NSPO Parallel Testbed.

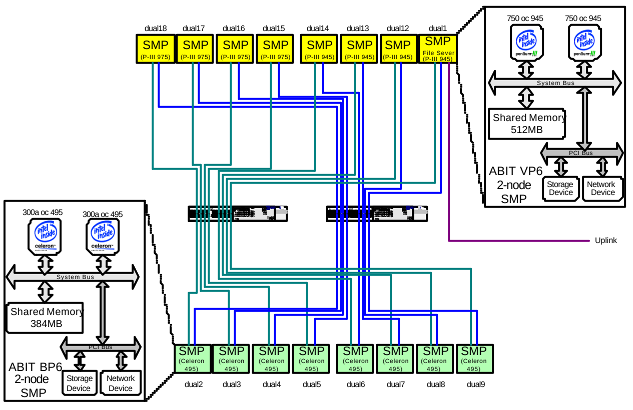


Figure 3: The NSPO Parallel Testbed system architecture.

2.2. System Software

2.2.1. Logical View of Beowulf

A Beowulf cluster uses multi computer architecture, as depicted in Figure 4. It features a parallel computing system that usually consists of one or more master nodes and one or more compute nodes, or cluster nodes, interconnected via widely available network interconnects. All of the nodes in a typical Beowulf 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. Because 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 Beowulf cluster appears as a Massively Parallel Processor (MPP) 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. 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 style 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) and Parallel Virtual Machine (PVM), to exchange information.

Since a Beowulf cluster is an MPP system, it suits applications that can be partitioned into tasks, which can then be executed concurrently by a number of processors. These applications range from high-end, floating-point intensive scientific and engineering problems to commercial data-intensive tasks. Uses of these applications include ocean and climate modeling for prediction of temperature and precipitation, seismic analysis for oil exploration, aerodynamic simulation for motor and aircraft design, and molecular modeling for biomedical research.

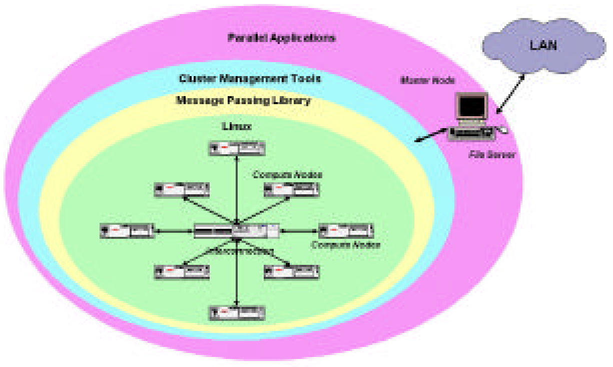


Figure 4: Logic view of a Beowulf cluster.

2.2.2. Linux

Linux is a freely available UNIX-like open operating system that is supported by its users and developers. Now, Linux has become a robust and reliable POSIX compliant operating system. Several companies have built businesses from packaging Linux software into organized distributions; RedHat is an example of such a company. Linux provides the features typically found in standard UNIX such as multi-user access, pre-emptive multi-tasking, demand-paged virtual memory and SMP support. In addition to the Linux kernel, a large amount of application and system software and tools are is also freely available. This makes Linux the preferred operating system for clusters.

The idea of the Linux cluster is to maximize the performance-to-cost ratio of computing by using low-cost commodity components and free-source Linux and GNU software to assemble a parallel and distributed computing system. Software support includes the standard Linux/GNU environment, including compilers, debuggers, editors, and standard numerical libraries. Coordination and communication among the processing nodes so they can truly work together is an obvious key requirement of parallel-processing clusters. In order to accommodate this coordination, developers have created software to carry out the coordination and hardware to send and receive the coordinating messages. Messaging architectures such as MPI or Message Passing Interface, and PVM or Parallel Virtual Machine, allow the programmer to ensure that control and data messages take place as needed during operation. Several approaches to message passing are discussed below.

2.2.3. PVM

PVM, or Parallel Virtual Machine, started out as a project at the Oak Ridge National Laboratory and was developed further at the University of Tennessee [6]. PVM is a complete distributed computing system, allowing programs to span several machines across a network. PVM utilizes a Message Passing model that allows developers to distribute programs across a variety of machine architectures and across several data formats. PVM essentially collects the network's workstations into a single virtual machine. PVM allows a network of heterogeneous computers to be used as a single computational resource called the parallel virtual machine. As we have seen, PVM is a very flexible parallel processing environment. It therefore supports almost all models of parallel programming, including the commonly used all-peers and master-slave paradigms. The chief reasons for reasons for the richness in the models it provides a framework for are its support for dynamic cluster change and dynamic spawning of processes.

2.2.4. MPI

MPI is a message-passing application programmer interface with protocol and semantic specifications for how its features must behave in any implementation (such as a message buffering and message delivery progress requirement). MPI includes point-to-point message passing and collective (global) operations. These are all scoped to a user-specified group of processes [4]. In addition, MPI supplies abstractions for processes at two levels. At the first level, processes are named according to the rank of the group in which the communication is being performed. At the second level, virtual topologies allow for graph or Cartesian naming of processes that help relate in a convenient, efficient way the application semantics to the message passing semantics. Communicators, which house groups and communication context (scoping) information, provide an important measure of safety that is necessary and useful for building up library-oriented parallel code. MPI provides a substantial set of libraries for the writing, debugging, and performance testing of distributed programs. Our system currently offers LAM/MPI, a portable implementation of the MPI standard developed cooperatively by Notre Dame University. LAM (Local Area Multicomputer) is an MPI programming environment and development system and includes a visualization tool that allows a user to examine the state of the machine allocated to their job as well as provides a means of studying message flows between nodes.

3 Applications and Performance

3.1. Performance of Channel Bonding

We use a simple program to measure the round ripe time of messages of various lengths between two nodes of our clusters. The program uses the MPI_Send and MPI_Recv library routines for sending and receiving messages. The latency and bandwidth are two important parameters that characterize a network. The latency measures the overhead associated with sending or receiving a message and is often measures as half the round trip time for a small message. Figure 5 shows the round trip time (ms) versus message size (Byte) achieved by MPI library. In order to reduce sampling errors, the message is send back and forth 1,000 times and the average of these round trip times is taken. There are three cases are considered: One measuring the round trip time between two processors in the same SMP; one measuring the time between two processors in the different SMP; and one measuring the time between two processors in the different SMP with channel bonded. Case one incurs a cost of 27.2124 microseconds for a 524288-byte message; whereas the same messages take 106.756 ms and 58.9251 ms for case 2 and 3, respectively. In general, channel bonding can improve the communication bandwidth and reduce the communication time for larger messages.

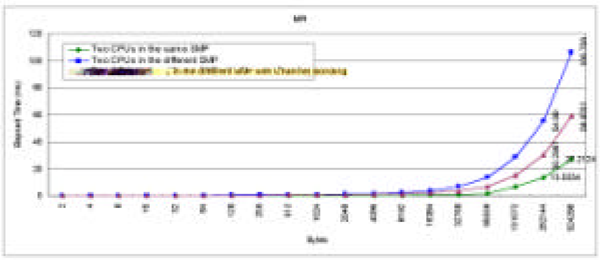


Figure 5: The performance of channel bonding by using MPI.

3.2 Matrix Multiplication

The matrix operation derives a resultant matrix by multiplying two input matrices, a and b, where matrix a is a matrix of N rows by P columns and matrix b is of P rows by M columns. The resultant matrix c is of N rows by M columns. The serial realization of this operation is quite straightforward as listed in the following:

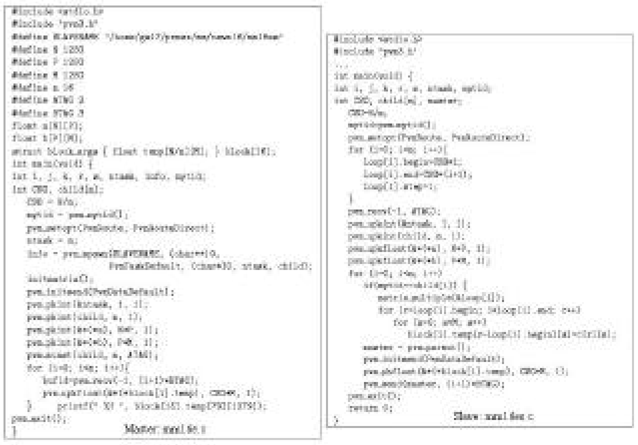
```
for(k=0; k<M; k++)
    for(i=0; i<N; i++){
        c[i][k]=0.0;
        for(j=0; j<P; j++)
            c[i][k]+=a[i][j]*b[j][k];
    }
```

Its algorithm requires n3 multiplications and n3 additions, leading to a sequential time complexity of O(n3). Let's consider what we need to change in order to use PVM. The first activity is to partition the problem so each slave node can perform on its own assignment in parallel. For matrix multiplication, the smallest sensible unit of work is the computation of one element in the result matrix. It is possible to divide the work into even smaller chunks, but any finer division would not be beneficial because of the number of processor is not enough to process, i.e., n2 processors are needed.

The matrix multiplication algorithm is implemented in PVM using the master-slave paradigm as shown in

Figure 6. The master task is named master_mm_pvm, and the slave task is named slave_mm_pvm. The master reads in the input data, which includes the number of slaves to be spawned, nTasks. After registering with PVM and receiving a taskid or tid, it spawns nTasks instances of the slave program slave_mm_pvm and then distributes the input graph information to each of them. As a result of the spawn function, the master obtains the tids from each of the slaves. Since each slave needs to work on a distinct subset of the set of matrix elements, they need to be assigned instance IDs in the range (0, ..., nTask-1). The tids assigned to them by the PVM library do not lie in this range, so the master needs to assign the instance IDs to the slave nodes and send that information along with the input matrix. Each slave also need to know the total number of slaves in the program, and this information is passed on to them by the master process as an argument to the spawn function since, unlike the instance IDs, this number is the same for all nTasks slaves.

To send the input data and instance ID information, the master process packs these into the active send buffer, and then invokes the send function. It then waits to receive partial results from each of the slaves. The slaves register with the PVM environment, and then wait for input data from the master, using a wildcard in the receive function to receive a message from any source. Once a message is received, each slave determines the master's tid from the received message buffer properties. Alternatively, the slaves could have determined the master's tid by calling the pvm_parent function, which they could have used as the source in their receive function. Upon receiving the message from the master that contains the input matrix, a slave unpacks this data from the active receive buffer. Each slave then works on its input partition, and send its partial results to the master when it is done. Then the master collects these partial results into an output matrix and outputs the results. In the slave program, we keep the basic structure of the sequential program intact. But now the routine to multiply the two matrices, the main program of slave_mm_pvm does not do the actual work itself, only performs the loop partition for each individual portion. Instead, the slave program calls a function matrix_multiple to perform real matrix multiplication. Each individual slave then performs a portion of the matrix multiplication as shown in Figure 7.



```
#include <stdio.h>
#include "pvm3.h"
#define SLAVENAME "/home/[illegible]/pvm3/mm/[illegible]/mslave"
#define N 1200
#define P 1200
#define M 1200
#define n 16
#define ATAG 2
#define BTAG 3
float a[N][P];
float b[P][M];
struct block_args { float temp[N/n][M]; } block[16];
int main(void) {
int i, j, k, r, m, ntask, info, mytid;
int CHU, child[n];
   CHU = N/n;
   mytid = pvm_mytid();
   pvm_setopt(PvmRoute, PvmRouteDirect);
   ntask = n;
   info = pvm_spawn(SLAVENAME, (char**)0,
                    PvmTaskDefault, (char*)0, ntask, child);
   initmatrix();
   pvm_initsend(PvmDataDefault);
   pvm_pkint(&ntask, 1, 1);
   pvm_pkint(child, n, 1);
   pvm_pkint(&a[0][0], N*P, 1);
   pvm_pkint(&b[0][0], P*M, 1);
   pvm_mcast(child, n, ATAG);
   for (i=0; i<n; i++){
      bufid=pvm_recv(-1, (i+1)+BTAG);
      pvm_upkfloat(&b[0]+block[i].temp), CHU*M, 1);
   }     printf(" %f ", block[%d].temp[%d][%d]);
pvm_exit();
}                    Master: mm16.c
```

```
#include <stdio.h>
#include "pvm3.h"
...
int main(void) {
int i, j, k, r, m, ntask, mytid;
int CHU, child[n], master;
   CHU=N/n;
   mytid=pvm_mytid();
   pvm_setopt(PvmRoute, PvmRouteDirect);
   for (i=0; i<n; i++){
      Loop[i].begin=CHU*i;
      Loop[i].end=CHU*(i+1);
      Loop[i].step=1;
   }
   pvm_recv(-1, ATAG);
   pvm_upkint(&ntask, 1, 1);
   pvm_upkint(child, n, 1);
   pvm_upkfloat(&a[0][0], N*P, 1);
   pvm_upkfloat(&b[0][0], P*M, 1);
   for (i=0; i<n; i++)
      if(mytid==child[i]) {
         matrix_multiple(&Loop[i]);
         for (r=Loop[i].begin; r<Loop[i].end; r++)
            for (m=0; m<M; m++)
               block[i].temp[r-Loop[i].begin][m]=c[r][m];
      master = pvm_parent();
      pvm_initsend(PvmDataDefault);
      pvm_pkfloat(&b[0]+block[i].temp), CHU*M, 1);
      pvm_send(master, (i+1)+BTAG);
   pvm_exit();
   return 0;
}                    Slave: mm16s.c
```

Figure 6: The master_mm_pvm.c and slave_mm_pvm.c programs

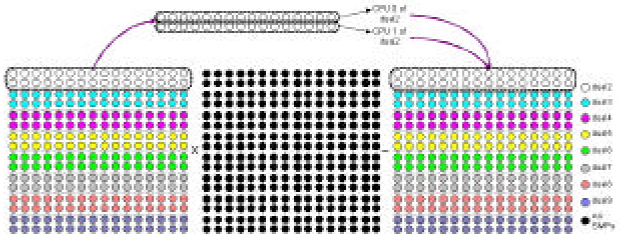


Figure 7: The block partition for MM.

The matrix multiplication was run with forking of different numbers of tasks to demonstrate the speedup. The problem sizes were 256X256, 512X512, 768X768, 1024X1024, and 2048X2048 in our experiments. It is well known, the speedup can be defined as Ts/Tp, where Ts is the execution time using serial program, and Tp is the execution time using multiprocessor. The execution times on 1, 2, 4, 8, and 16, were listed in Figure 8, respectively. In Figure 9, the corresponding speedup is increased for different problem sizes by varying the number of slave programs. Since matrix multiplication was a uniform workload application, the high speedups were gained about 1.76 (256X256) using 2 processors of one SMP, and 3.41 (256X256) using two SMPs. The highest speedup was obtained about 10.85 (2048X2048) by using our SMP cluster with 16 processors. We also found that the speedups were closed when creating two slave programs on one dual processor machine and two slaves program on two SMPs respectively by using PVM library. With channel-bonded technique, the highest speedup was measured about 11.71 shown in Figure 10.

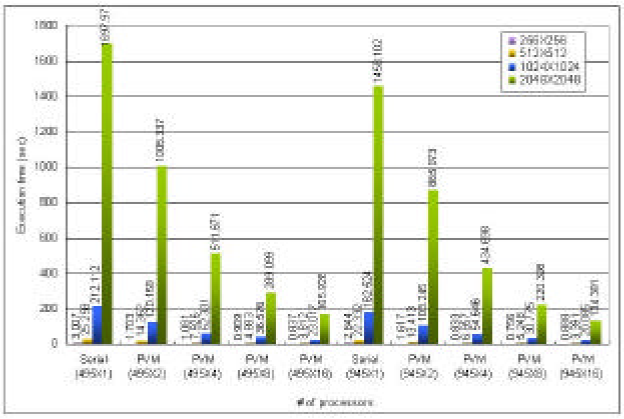


Figure 8: Execution times of MM with different number of tasks.

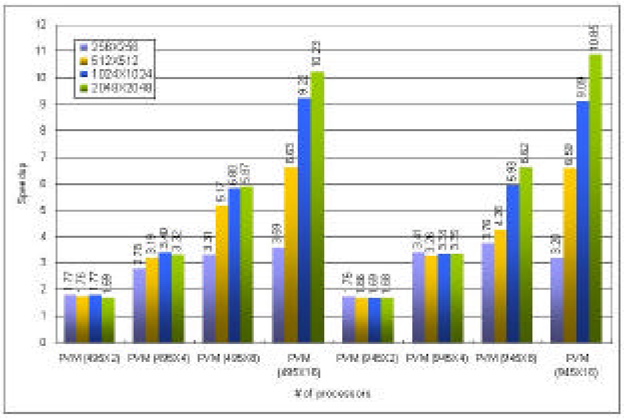


Figure 9: Speedups of MM with different number of tasks.

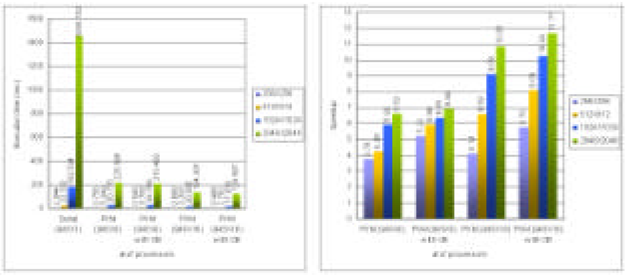


Figure 10: The execution times and the corresponding speedups of MM by using channel bonded.

3.3. PVMPOV

Rendering is a technique for generating a graphical image from a mathematical model of a two or three-dimensional object or scene. A common method of rendering is ray tracing. Ray tracing is a technique used in computer graphics to create realistic images by calculating the paths taken by rays of light entering the observer's eye at different angles. Ray tracing is an ideal application for parallel processing since there are many pixels, each of whose values are independent and can be calculated in parallel. The Persistence of Vision Ray Tracer (POV-Ray) is an all-round 3-dimensional ray tracing software package [5]. It takes input information and simulates the way light interacts with the objects defined to create 3D pictures and animations. In addition to the ray tracing process, newer versions of POV can also use a variant of the process known as radiosity (sophisticated lighting) to add greater realism to scenes, particularly those that use diffuse light POVRay can simulate many atmospheric and volumetric effects (such as smoke and haze).

Given a number of computers and a demanding POVRay scene to render, there are a number of techniques to distribute the rendering among the available resources. If one is rendering an animation then obviously each computer can render a subset of the total number of frames. The frames can be sent to each computer in contiguous chunks or in an interleaved order, in either case a preview (every Nth frame) of the animation can generally be viewed as the frames are being computed. POVRay is a multi-platform, freeware ray tracer. Many people have modified its source code to produce special "unofficial" versions. One of these unofficial versions is PVMPOV, which enables POVRay to run on a Linux cluster.

PVMPOV has the ability to distribute a rendering across multiple heterogeneous systems. Parallel execution is only active if the user gives the "+N" option to PVMPOV. Otherwise, PVMPOV behaves the same as regular POV-Ray and runs a single task only on the local machine. Using the PVM code, there is one master and many slave tasks. The master has the responsibility of dividing the image up into small blocks, which are assigned to the slaves. When the slaves have finished rendering the blocks, they are sent back to the master, which combines them to form the final image. The master does not render anything by itself, although there is usually a slave running on the same machine as the master, since the master doesn't use very much CPU power.

If one or more slaves fail, it is usually possible for PVMPOV to complete the rendering. PVMPOV starts the slaves at a reduced priority by default, to avoid annoying the users on the other machines. The slave tasks will also automatically time out if the master fails, to avoid having lots of lingering slave tasks if you kill the master. PVMPOV can also work on a single machine, like the regular POV-Ray, if so desired. The code is designed to keep the available slaves busy, regardless of system loading and network bandwidth. We have run PVMPOV on our 16-Celeron and 16-PIII processors testbed and have had amazing results, respectively. With the cluster configured, runs the following commands to begin the ray tracing and generates the image files as shown in Figure 11:

```
$pvmpov +iskyvase.pov +w640 +h480 +nw32 +nh32 +nt16 -L/home/gs17/pvmpov3_1g_1/povray31/include
$pvmpov +ifish13.pov +w640 +h480 +nw32 +nh32 +nt16 -L/home/gs17/pvmpov3_1g_1/povray31/include
$pvmpov +ipawns.pov +w640 +h480 +nw32 +nh32 +nt16 -L/home/gs17/pvmpov3_1g_1/povray31/include
$pvmpov +iEstudio.pov +w640 +h 480 +nw32 +nh32 +nt16 -L/home/gs17/pvmpov3_1g_1/povray31/include
```

This is the benchmark option command-line with the exception of the +nw and +nh switches, which are specific to PVMPOV and define the size of image each of the slaves, will be working on. The +nt switch is specific to the number of tasks will be running. For example, +nt16 will start 16 tasks, one for each processor. The messages on the screen should show that slaves were successfully started. When completed, PVMPOV will display the slave statistics as well as the total render time. In case of Skyvase model, by using single Celeron processor mode of a dual processor machine for processing 1600X1280 image, the render time was 256 seconds. Using out Celeron-based SMP cluster (16 processors) further reduced the time to 26 seconds. The execution times for the different POVray model (Skyvase, Fish13, Pawns, and Estudio) on Celeron SMPs and P-III SMP clusters were shown in Figure 12, respectively. The corresponding speedups of different problem size by varying the number of task (option: +nt) was shown in Figure 13. The highest speedups were obtained about 15.22 and 13.45 (1600X1280) for Pawns model by using our Celeron SMPs cluster with 16 processors and P-III SMPs cluster with 16 processors, respectively.

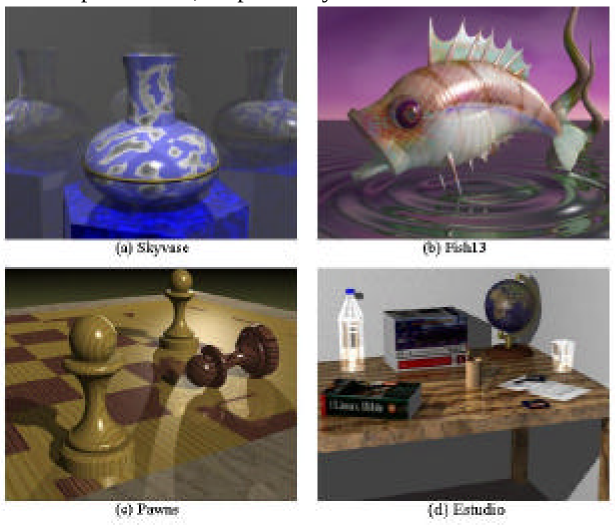


Figure 11: Four diagrams were generated by PVMPOV.

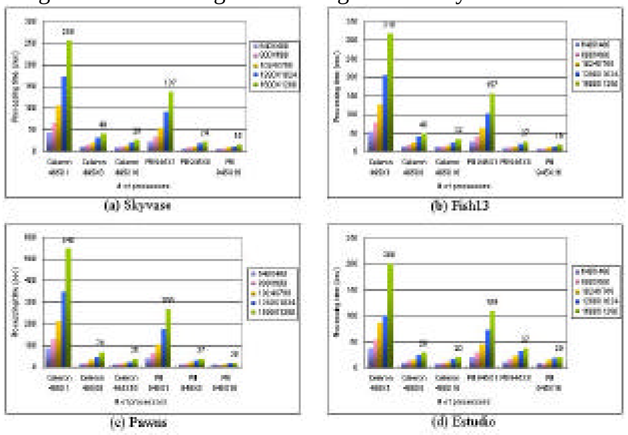


Figure 12: Execution times of PVMPOV diagram.

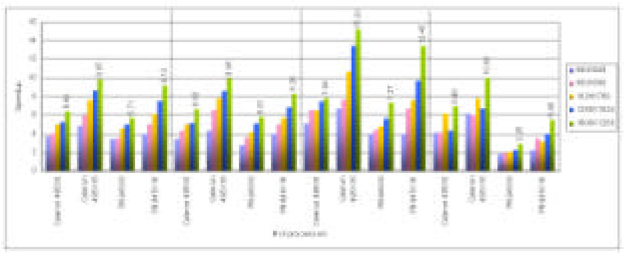


Figure 13: Speedups of PVMPOV diagrams

3.4. NAS Parallel Benchmark

The NAS Parallel Benchmark (NPB) is a set of 8 programs designed to help evaluate the performance of parallel supercomputers. The benchmarks, which are derived from computational fluid dynamics (CFD) applications, consist of five kernels and three pseudo-applications. NPB 2.3 is MPI-based source-code implementations written and distributed by NAS. They are intended to run with little or no tuning, and approximate the performance a typical user can expect to obtain for a portable parallel program. The LU benchmark is based on the NX reference implementation from 1991. This code requires a power-of-two number of processors. A 2-D partitioning of the grid onto processors occurs by halving the grid repeatedly in the first two dimensions, alternately x and then y, until all power-of-two processors are assigned, resulting in vertical pencil-like grid partitions on the individual processors. This ordering of point based operations constituting the SSOR procedure proceeds on diagonals which progressively sweep from one corner on a given z plane to the opposite corner of the same z plane, thereupon proceeding to the next z plane. Communication of partition boundary data occurs after completion of computation on all diagonals that contact an adjacent partition. This constitutes a diagonal pipelining method and is called a "wavefront" method. It results in relatively large number of small communications of 5 words each.

The one NAS benchmark that we chose to present here is LU. For the LU benchmark, the sizes were class A and B. The execution time of LU was shown in Figure 14 (a). The performance numbers for 16 processors as reported in Figure 14 (b) by the LU benchmark were 715.06 MFLOPS and 778.62 MFLOPS for class A and class B, respectively. As a measure of scalability, we selected parallel speedup, as classically calculated (that is, as the ratio between the serial time Ts and the parallel time Tp for the execution of the benchmark, Ts/Tp). The serial time was obtained by running the benchmarks on one processor. The speedup of LU benchmark is reported in Figure 14 (c).

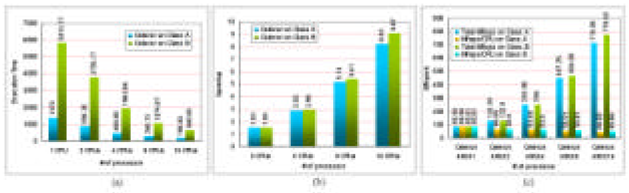


Figure 14: (a) Execution time of LU. (b) Speedup of LU using 16 processors. (c) Total Mflop/s obtained using 16 processors.

3.5. High Performance Linpack (HPL)

HPL is a software package that solves a (random) dense linear system in double precision (64 bits) arithmetic on distributed-memory computers [3]. It can thus be regarded as a portable as well as freely available implementation of the High Performance Computing Linpack Benchmark. The HPL software package requires the availability on your system of an implementation of the Message Passing Interface MPI (1.1 compliant). An implementation of either the Basic Linear Algebra Subprograms BLAS or the Vector Signal Image Processing Library VSIPL is also needed. Machine-specific as well as generic implementations of MPI, the BLAS and VSIPL are available for a large variety of systems.

This software package solves a linear system of order n: Ax=b by first computing the LU factorization with row partial pivoting of the n-by-n+1 coefficient matrix [A b]=[[L, U] y]. Since the lower triangular factor L is applied to b as the factorization progresses, the solution x is obtained by solving the upper triangular system Ux=y. The lower triangular matrix L is left unpivoted and the array of pivots is not returned. The data is distributed onto a two-dimensional P-by-Q grid of processes according to the block-cyclic scheme to ensure "good" load balance as well as the scalability of the algorithm. The n-by-n+1 coefficient matrix is first logically partitioned into NB-by-NB blocks, which are cyclically "dealt" onto the P-by-Q process grid. This is done in both dimensions of the matrix. The right-looking variant has been chosen for the main loop of the LU factorization. This means that at each iteration of the loop a panel of NB columns is factorized, and the trailing submatrix is updated. Note that this computation is thus logically partitioned with the same block size NB that was used for the data distribution.

The HPL package provides a testing and timing program to quantify the accuracy of the obtained solution as well as the time it took to compute it. The best performance achievable by this software on your system depends on a large variety of factors. Nonetheless, with some restrictive assumptions on the interconnection network, the algorithm described here and its attached implementation are scalable in the sense that their parallel efficiency is maintained constant with respect to the per processor memory usage. In order to find out the best performance of your system, the largest problem size fitting in memory is what you should aim for. The amount of memory used by HPL is essentially the size of the coefficient matrix. For example, if you have 8 nodes with 512MB of memory on each, this corresponds to 4GB total, i.e., 500M double precision (8 Bytes) elements. The square root of that number is 22360. One definitely needs to leave some memory for the OS as well as for other things, so a problem size of 20000 is likely to fit. As a rule of thumb, 80 % of the total amount of memory is a good guess. If the problem size you pick is too large, swapping will occur, and the performance will drop. If multiple processes are spawn on each node (say you have 2 processors per node), what counts is the available amount of memory to each process. The performance achieved by this software package on our cluster is shown in Figure 15. We compare the system performance obtained from our cluster with P-III 550X16 data that generated by University of Tennessee (UT) from the HPL web site. Our P-III SMP cluster can achieve 6.433Gflop/s for the problem size 20000X20000 with channel bonded. Also, Our cluster can achieve 6.669Gflop/s by using 16 P-III and 16 Celeron processors.

HPL uses the block size NB for the data distribution as well as for the computational granularity. From a data distribution point of view, the smallest NB, the better the load balance. You definitely want to stay away from very large values of NB. From a computation point of view, a too small value of NB may limit the computational performance by a large factor because almost no data reuse will occur in the highest level of the memory hierarchy. The number of messages will also increase. Efficient matrix-multiply routines are often internally blocked. Small multiples of this blocking factor is likely to be good block sizes for HPL. The bottom line is that "good" block sizes are almost always in the [32, 256] interval. The best values depend on the computation/communication performance ratio of your system. This depends on the physical interconnection network you have. In other words, P and Q should be approximately equal, with Q slightly larger than P. If you are running on a simple Ethernet network, there is only one wire through which all the messages are exchanged. On such a network, the performance and scalability of HPL is strongly limited and very flat process grids are likely to be the best choices: 1X4, 1X8, and 2X4. For example, in Figure 16 we can found that the case of 4X4 always got more computational speed than the case of 2X8 by using a 16-processor cluster.

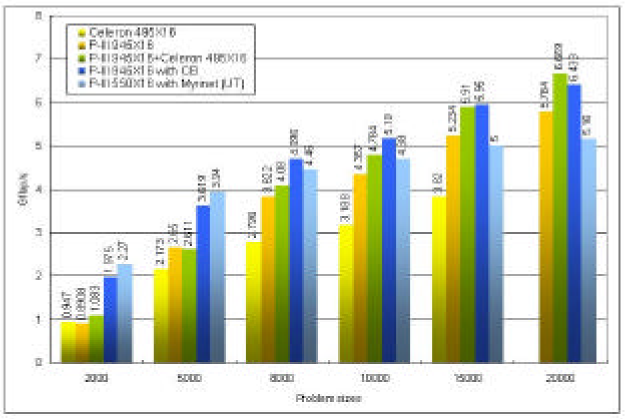


Figure 15: The system performance comparison of our cluster with HPL web site data.

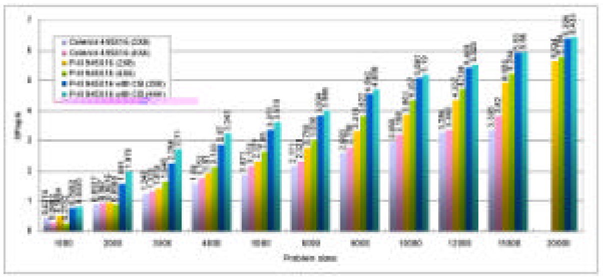


Figure 16: The performance of HPL with the different case of PXQ

4 Parallel Programming for SMP Clusters

Architectures of parallel systems are broadly divided into shared-memory and distributed-memory models. While multithreaded programming is used for parallelism on shared-memory systems, the typical programming model on distributed-memory systems is message passing. SMP clusters have a mixed configuration of shared-memory and distributed-memory architectures. One way to program SMP clusters is to use an all-message-passing model. This approach uses message passing even for intra-node communication. It simplifies parallel programming for SMP clusters but might lose the advantage of shared memory in an SMP node. Another way is with the all-shared-memory model, using a software distributed-shared-memory (DSM) system such as TreadMarks. This model, however, needs complicated runtime management to maintain consistency of the shared data between nodes.

We will use a hybrid-programming model of shared and distributed memory to take advantage of locality in each SMP node. Intra-node computations use multithreaded programming, and inter-node programming is based on message passing and remote memory operations. Consider data-parallel programs. We can easily phase the partitioning of target data such as matrices and vectors. First, we partition and distribute the data between nodes and then partition and assign the distributed data to the threads in each node. Data decomposition and distribution and inter-node communications are the same as in distributed-memory programming. Data allocation to the threads and local computation are the same as in multithreaded programming on shared-memory systems. Hybrid programming is a type of distributed programming, in that computation in each node uses multiple threads. Although some data-parallel operations such as reduction and scan need more complicated steps in hybrid programming, we can easily implement hybrid programming by combining both shared and distributed programming for data-parallel programs.

The matrix multiplication was run with forking of different numbers of tasks to demonstrate the speedup as shown in Figure 17. The problem sizes were 256X256, 512X512, 1024X1024, and 1280X1280, in our experiments. We found that the execution time of combining thread and PVM model always small than pure PVM model. The hybrid-programming model of shared and distributed memory can take advantage of locality in each SMP node. It is believed that hybrid-programming model is the most obvious approach to help programmer to take advantage of clustering symmetric multiprocessors (SMP) parallelism.

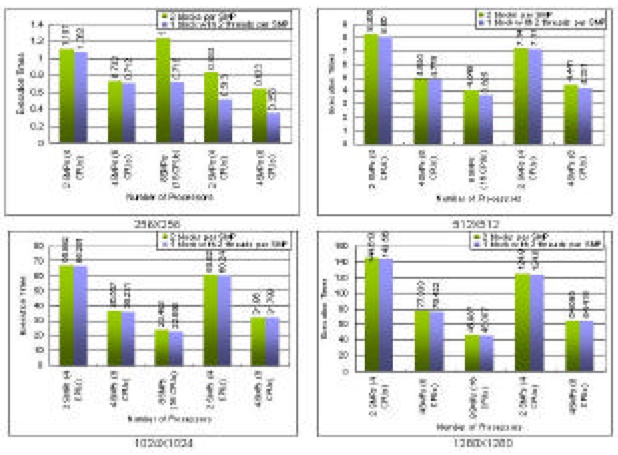


Figure 17: Performance comparisons between pure message passing and hybrid with different problem size.

5 Near Work: Automatic Translator for Parallel Programming

PVM programming support may be the most obvious approach to help programmers to take advantage of parallelism by the operating systems. Therefore, we propose a new model of parallelizing compiler for exploiting potential power of multiprocessors and gaining performance benefit on cluster systems [14]. The portable automatic parallel program generator (APPG) for parallelizing compiler to produce parallel object codes is shown in Figure 18.

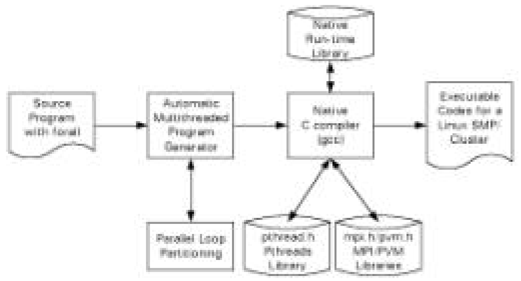

Native Run-time Library
Source Program with forall
Automatic Multithreaded Program Generator
Native C compiler (gcc)
Executable Codes for a Linux SMP Cluster
Parallel Loop Partitioning
pthread.h Pthreads Library
mpi.h/pvm.h MPI/PVM Libraries

Figure 18: The system structure of APPG

First, the automatic parallel program generator (APPG) takes the C source program as input, and then generates the output in which the parallel loops (doall) are translated into sub-tasks by replacing them with multithreaded codes. Our AMPG will use some loop-partitioning algorithms, e.g., Chunk Self-Scheduling (CSS), Factoring, and Trapezoid Self-Scheduling (TSS) to partition a doall loop. Second, The resulting multithreaded program is then compiled and linked with the pthreads run-time libraries or message passing library, such as PVM or MPI, by using the native C compiler, e.g., GNU C compiler. Then, the generated parallel object codes can be scheduled and executed in parallel on the multiprocessors or cluster system to achieve high performance. Based upon this model, we will implement a parallelizing compiler to help programmers take advantage of multithreaded parallelism and message passing on SMP clusters, running Linux.

6 Conclusion and Future Work

Scalable computing clusters, ranging from a cluster of (homogeneous or heterogeneous) PCs or workstations, to SMPs, are rapidly becoming the standard platforms for high-performance and large-scale computing. It is believed that message-passing programming is the most obvious approach to help programmer to take advantage of clustering symmetric multiprocessors (SMP) parallelism. In this paper, the topics including system architecture, software tools, and applications of our cluster system were discussed. In order to take advantage of a cluster system, we presented the basic programming techniques by using Linux/PVM to implement a PVM-based matrix multiplication program. Also, a real application PVMPOV by using parallel ray-tracing techniques was examined. The experimental results show that the highest speedups are 10.85 and 15.22 respectively for matrix multiplication and PVMPOV, when the total number of processors is 16, by creating 16 tasks on SMPs cluster. Furthermore, two benchmarks, NAS and HPL are used to demonstrate the performance of our parallel testbed by using MPI. The experimental results show that our cluster can obtain 6.433Gflops for HPL programs with channel bonded, when the total number of processors used is 16. The results of this study will make theoretical and technical contributions to the design of a message passing program on a Linux SMP clusters. In the near future, we will propose a new model of parallelizing compiler for exploiting potential power of multiprocessor systems and gaining performance benefit on PC-based SMP cluster systems. Also, we will expend the parallel computing research to remote sensing applications.

References

1. R. Buyya, *High Performance Cluster Computing: System and Architectures*, Vol. 1, Prentice Hall PTR, NJ, 1999.
2. R. Buyya, *High Performance Cluster Computing: Programming and Applications*, Vol. 2, Prentice Hall PTR, NJ, 1999.
3. http://www.netlib.org/benchmark/hpl, *HPL – A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers*.
4. http://www.lam-mpi.org, *LAM/MPI Parallel Computing*.
5. http://www.haveland.com/povbench, *POVBENCH – The Official Home Page*.
6. http://www.epm.ornl.gov/pvm/, *PVM – Parallel Virtual Machine*.
7. Lie, W. N., *Distributed Computing Systems for Satellite Image Processing*, Technical Report, EE, National Chung Cheng University, 2001.
8. Lillesand, Thomas M. and Kiefer, Ralph W., *Remote Sensing and Image Interpretation*, Third Edition, John Wiley & Sons, 1994.
9. Richards, John A., *Remote Sensing Digital Image Analysis: An Introduction*, Springer-Verlag, 1999.
10. T. L. Sterling, J. Salmon, D. J. Backer, and D. F. Savarese, *How to Build a Beowulf: A Guide to the Implementation and Application of PC Clusters*, 2nd Printing, MIT Press, Cambridge, Massachusetts, USA, 1999.
11. B. Wilkinson and M. Allen, *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*, Prentice Hall PTR, NJ, 1999.
12. M. Wolfe, *High-Performance Compilers for Parallel Computing*, Addison-Wesley Publishing, NY, 1996.
13. C. T. Yang, S. S. Tseng, M. C. Hsiao, and S. H. Kao, "A Portable parallelizing compiler with loop partitioning," *Proc. of the NSC ROC(A)*, Vol. 23, No. 6, 1999, pp. 751-765.
14. Chao-Tung Yang, Shian-Shyong Tseng, Yun-Woei Fan, Ting-Ku Tsai, Ming-Hui Hsieh, and Cheng-Tien Wu, "Using Knowledge-based Systems for research on portable parallelizing compilers," *Concurrency and Computation: Practice and Experience*, vol. 13, pp. 181-208, 2001.