THPTB: A SMP-based PC Cluster with Channel Bonding for High-Performance Computing

Chao-Tung Yang and Yao-Chung Chang

High-Performance Computing Laboratory
Department of Computer Science and Information Engineering
Tunghai University
Taichung, 407, Taiwan, R.O.C.
Tel: +886-4-23590121 ext. 3279
ctyang@mail.thu.edu.tw

Abstract. To use supercomputer for high-performance computing has been growing. Supercomputers that are single big expensive machines with a shared memory and one or more processors meet the professional need. A large-scale processing and storage system that provides high bandwidth at low cost is then their expectation. A cluster is a collection of independent and cheap machines, used together as a supercomputer to provide a solution. In this paper, a SMP-based PC cluster (36 processors), called THPTB (TungHai Parallel TestBed) with channel bonded technique, was proposed and built. The system architecture and benchmark performances of the cluster are also presented in this paper. To take advantage of the parallelism of the SMPs cluster systems, the PVM or MPI library has been used to code a message-passing program on our SMPs cluster. To measure the computing performance, the matrix multiplication program with the different sizes has been performed and parallel ray-tracing program PVMPOV has been used to demonstrate the performance in the SMP cluster by using PVM. The HPL benchmark is also used to demonstrate the performance of our testbed by using LAM/MPI. The experimental results show that our cluster can obtain 17.38 Gflop/s with channel bonding, when the total number of processor used is 36.

1. Introduction

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. 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, 11, 12, 13, 14, 15, 16, 17].

Scalable computing clusters, ranging from a cluster of (homogeneous or heterogeneous) PCs or workstations to SMP (Symmetric MultiProcessors), are rapidly becoming the standard platforms for high-performance and large-scale computing. A cluster is a group of independent computer systems and thus forms a loosely coupled multiprocessor system as show in Figure 1. 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. 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.

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 [11]. The goal of building a Beowulf cluster is to create a cost-effective parallel computing system from commodity components to satisfy specific computational requirements for the earth and space

sciences community. The first Beowulf cluster was built from 16 Intel® DX4™ processors connected by a channel-bonded 10 Mbps Ethernet, and it ran the Linux operating system. 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.

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.

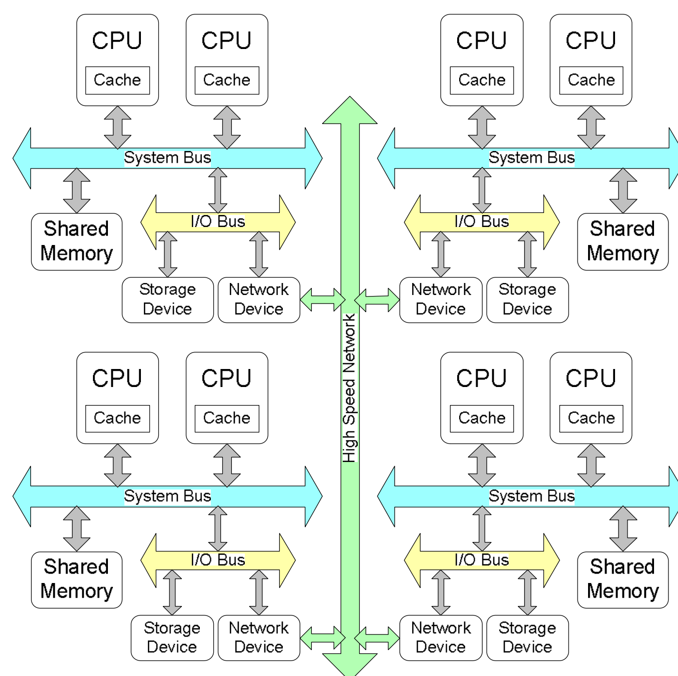

Figure 1: A cluster system by connecting four SMPs.

Beowulf is a concept of clustering commodity computers to form a parallel, virtual supercomputer. It is easy to build a unique Beowulf cluster from available components. Currently, we conducted and maintained an experimental Linux SMP cluster (SMP PC machines running the Linux operating system), named THPTB (TungHai Parallel TestBed), which is served as a computing resource for testing. THPTB is made up of 18 dual Intel® P-III SMP-based PCs. Nodes are connected using Fast Ethernet with a maximum bandwidth of 300Mbits, through three 24-port switches with channel bonded technique. *Channel bonding* is a method where the data in each message gets striped across the multiple network cards installed in each machine [1, 2, 11]. The THPTB is operated as a unit system to share networking, file servers, and other peripherals. The system can provide a cost-effective way to gain features and benefits (fast and reliable services) that have historically been found only on more expensive proprietary shared memory systems. The typical architecture of a cluster is shown in Figure 2. The shaded boxed and the bold lines show the configuration of our THPTB cluster from node (bottom) to parallel applications (top).

In this paper, the system architecture and benchmark performances of the cluster are presented. In order to measure the performance of our cluster, the parallel ray-tracing problem is illustrated and the experimental result is demonstrated on our Linux SMPs cluster. The experimental results show that the highest speedup is 20.25 for PVMPOV [5, 7], when the total numbers of processor is 32 on SMPs cluster. Also, the HPL benchmark [3] is used to demonstrate the performance of our testbed by using LAM/MPI library [4]. The experimental result shows that our cluster can obtain 17.38 GFlops/s when the total numbers of processors used is 36 with 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.

The rest of this paper is organized as follows. Section 2 gives the overview about Beowulf-class clusters and introduces system software including operating system, and two message passing libraries, PVM and MPI respectively. In Section 3, we describe how to setup a PC cluster with channel bonding step by step. Then, the matrix multiplication and PVMPOV are used as an illustration using PVM, respectively. Also, the HPL benchmark [3] is used to demonstrate the performance of our cluster by using MPI. Results from experimental measurements are presented in Section 4. Then, in Section 5 we state the near work that based upon our previous work on Linux SMP. Finally, concluding remarks are given in Section 6.

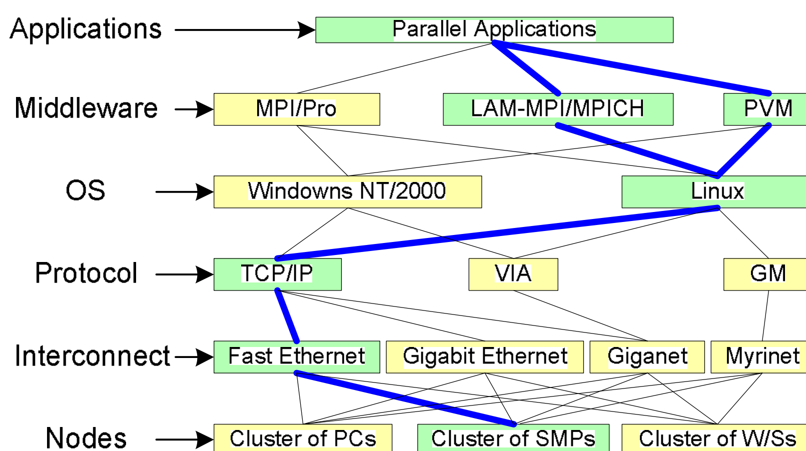

Figure 2: The architecture of cluster systems.

2. System Descriptions

2.1. Logical View of Cluster

A Beowulf cluster uses multicomputer architecture, as depicted in Figure 3. 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.

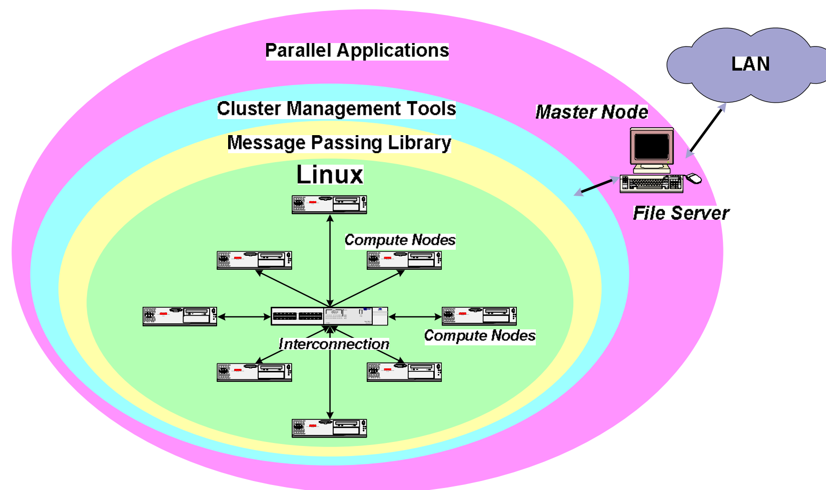

Figure 3: Logical view of Beowulf cluster.

2.2. Hardware

For our project we decided to build a cluster from scratch using standard PC parts. The acronym COTS, for commodity off-the-shelf technology, is often used to describe this approach. The main constraint was that we needed a number of PCs and a lot of memory to use in our computation. Communications and I/O are not a big issue for us since the PCs spend most of their time doing computations, and the amount of information exchanged between PCs is always comparatively small. Therefore, our particular application would not benefit significantly from the use of a high-performance network, such as Gigabit Ethernet or Myrinet. Instead, we used standard 100Mbps Fast Ethernet and channel bonded technique to achieve 300 Mbps. Due to the budget is limited; we used dual-processor motherboards to reduce the number of boxes to 18, thus minimizing the space needed for storage (and the footprint of the cluster). This structure impacts performance because two processors share the memory bus (which causes bus contention but reduces the hardware cost) since only one case, motherboard, hard drive, etc., are needed for two processors. We ruled out the option of rack-mounting the nodes, essentially to reduce cost, but chose to use standard mid-tower cases on shelves, as illustrated in Figure 4. This approach is sometimes given the name LOBOS ("lots of boxes on shelves").

Our SMP cluster, called THPTB (TungHai Parallel TestBed), is a low cost Beowulf-type class supercomputer that utilizes multi-computer architecture for parallel computations. THPTB consists of 18 PC-based symmetric multiprocessors (SMP) connected by three 24-port 100Mbps Ethernet switches with Fast Ethernet interface. Its system architecture is shown in Figure 5. There are one server node and 17 computing nodes. The server node has two Intel Pentium-III 1050MHz (1GHz over-clock, FSB 140MHz) processors and 1GBytes 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 kind (dual2 ~ dual10) is dual P-III 1GHz with 768MB shared-memory, and the other kind (dual11 ~ dual18) is dual P-III 950MHz (866MHz over-clock, FSB: 146MHz) with 512MB shared local memory.

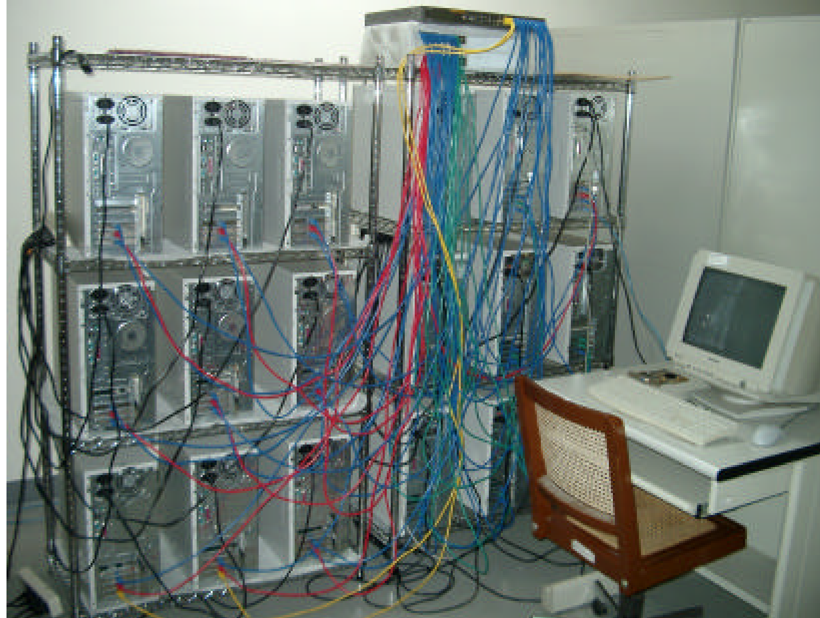

Figure 4: THPTB cluster snapshot.

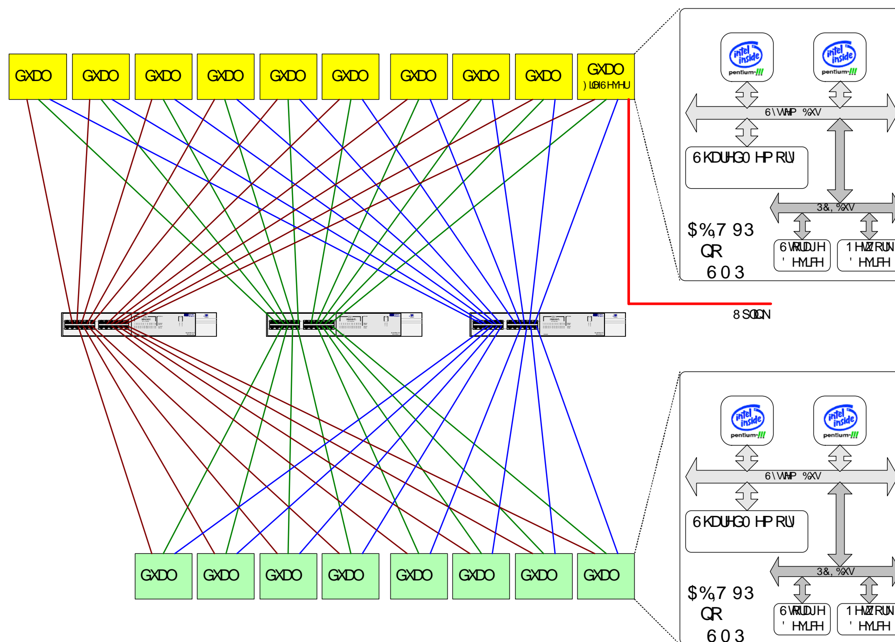

Figure 5: The network topological view of THPTB cluster.

2.3. Operating System

Linux is a robust, free 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 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 is a 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.

2.4. Message Passing Libraries

To use clusters of Intel architected PC machines for High Performance Computing applications, you must run the applications in parallel across multiple machines. Parallel processing requires that the code running on two or more processor nodes communicate and cooperating with each other. The message passing model of communication is typically used by programs running on a set of discrete computing systems (each with its own memory) which are linked together by means of a communication network. A cluster is such a loosely coupled distributed memory system.

2.4.1. Parallel Virtual Machine (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. 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.

A typical PVM consists of a (possibly heterogeneous) mix of machines on the network, one being the "master" host and the rest being "worker" or "slave" hosts. These various hosts communicate by message passing. The PVM is started at the command line of the master which in turn can spawn workers to achieve the desired configuration of hosts for the PVM. This configuration can be established initially via a configuration file. Alternatively, the virtual machine can be configured from the PVM command line (master's console) or during run time from within the application program.

A solution to a large task, suitable for parallelization, is divided into modules to be spawned by the master and distributed as appropriate among the workers. PVM consists of two software components, a resident daemon (**pvmd**) and the PVM library (**libpvm**). These must be available on each machine that is a part of the virtual machine. The first component, pvmd, is the message-passing interface between the application program on each local machine and the network connecting it to the rest of the PVM. The second component, libpvm, provides the local application program with the necessary message-passing functionality, so that it can communicate with the other hosts. These library calls trigger corresponding activity by the local pvmd which deals with the details of transmitting the message. The message is intercepted by the local pvmd of the target node and made available to that machine's application module via the related library call from within that program.

2.4.2. Message Passing Interfacing (MPI)

MPI is a message-passing library standard that was published in May 1994. The "standard" of MPI is based on the consensus of the participants in the MPI Forum, organized by over 40 organizations. Participants included vendors, researchers, academics, software library developers and users. MPI offers portability, standardization, performance, and functionality.

The advantage for the user is that MPI is standardized on many levels. For example, since the syntax is standardized, you can rely on your MPI code to execute under any MPI implementation running on your architecture. Since the functional behavior of MPI calls is also standardized, your MPI calls should behave the same regardless of the implementation. This guarantees the portability of your parallel programs. Performance, however, may vary between different implementations.

MPI includes point-to-point message passing and collective (global) operations. These are all scoped to a user-specified group of processes. MPI provides a substantial set of libraries for the writing, debugging, and performance testing of distributed programs. Our system currently uses

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. System Setup

3.1. Setup Hardware

Here are the main steps in setting up the hardware:

1. Moved the machines from various labs in the campus to the parallel processing lab, mounted them on the rack, names and numbered them.
2. Setup the three 24-port network switches and connected each port to Ethernet adapters of the machines one by one.
3. A single monitor, keyboard and mouse were connected to one of the machines that have 1GB RAM, 40GB extra hard disk and a high-end graphics card.
4. Powered all the components.

3.2. Setup Software

All the nodes in the cluster run Linux Red Hat 7.2. Linux corresponds perfectly to the demands of our application; we require very high reliability because the machine is being used by researchers who need their jobs to run without having to worry about nodes crashing. We have not had a single system crash since the machine was built seven months ago. In terms of network configuration, the 18 nodes are on a private network of 192.168.1.X addresses. For security reasons, the cluster is not connected to the outside world expect the master node. Here are the steps, which made the reality come true:

1. OS installation: RedHat Linux 7.2 was installed on all the machines by connecting all the peripherals such as monitor, mouse and keyboard. Most of the hardware was automatically detected, so main focus was on partitioning the drive and choosing the relevant packages to be installed. It is very important to choose partition sizes which are correct for the need because it might be very difficult to change this at a later stage when the cluster will be in the functional mode. Following is the list of partitions:
   - The `/` partition is about 2GB. This `/` partition contains `/bin`, `/boot`, `/dev`, `/lib`, `/root`, `/sbin`, `/var` and `/home` directories with their contents.
   - The `/usr` partition is about 1.5GB. This `/usr` partition were created by keeping in mind that most additional rpm's will install in `/usr`.
   - The `swap` partition: Swapping is really bad for the performance of the system. Unfortunately there might be a time when the machine is computing a very large job and just don't have enough memory. Since the machines have 512MB RAM, it was realized that a 512MB of swap partition was a good idea.
   - The `/export/home` partition: Rest part of the disk. This partition was used as users home on individual machines. If needed it can be NFS mounted for additional user space. One of the machines had 8.4GB of addition hard drive that was used for common home area for users.
2. Network Configuration: During OS installation IP addresses and nodes names were assigned. Following are the nodes names of Parallel Testbed

```
dual1 (192.168.1.1)
dual2 (192.168.1.2)
dual3 (192.168.1.3)
dual4 (192.168.1.4)
dual5 (192.168.1.5)
dual5 (192.168.1.5)
dual6 (192.168.1.6)
dual7 (192.168.1.7)
dual8 (192.168.1.8)
```

```
dual9 (192.168.1.9)
dual10 (192.168.1.10)
dual11 (192.168.1.11)
dual12 (192.168.1.12)
dual13 (192.168.1.13)
dual14 (192.168.1.14)
dual15 (192.168.1.15)
dual16 (192.168.1.16)
dual17 (192.168.1.17)
dual18 (192.168.1.18)
```

- For the configuration, the following files were modified by superuser: `/etc/sysconfig/network`, `/etc/sysconfig/network-scripts/ifcfg-eth0` and `/etc/sysconfig/network-scripts/ifcfg-eth3`. Here are these three files

```
/etc/sysconfig/network
NETWORKING=yes
HOSTNAME=dual1
NISDOMAIN=dual

/etc/sysconfig/network-scripts/ifcfg-eth0
DEVICE=eth0
ONBOOT=yes
BOOTPROTO=static
IPADDR=192.168.1.1
NETMASK=255.255.255.0

/etc/sysconfig/network-scripts/ifcfg-eth1
DEVICE=eth1
ONBOOT=yes

/etc/sysconfig/network-scripts/ifcfg-eth2
DEVICE=eth2
ONBOOT=yes

/etc/sysconfig/network-scripts/ifcfg-eth3
DEVICE=eth3
ONBOOT=yes
BOOTPROTO=static
IPADDR=140.128.101.190
NETMASK=255.255.255.0
GATEWAY=140.128.101.250
```

- `/etc/hosts.equiv`: In order to allow remote shells (**`rsh`**) from any node to any other in the cluster, for all users, we should relax the security, and list all hosts in `/etc/hosts.equiv`.

```
/etc/hosts.equiv
dual1
dual2
dual3
dual4
dual5
dual6
dual7
dual8
```

```
dual9
dual10
dual11
dual12
dual13
dual14
dual15
dual16
dual17
dual18
```

3. Network File System configuration: We have used fully local OS install configuration for this parallel testbed cluster. In this setup all the participating machines have their own disks with operating system and swap locally and only `/home` and `/usr/local` off the servers. We use Network File System (NFS) for mounting `/home` and `/usr/local` partitions. The advantage of this setup is no NFS traffic and the disadvantage of complicated installation and maintenance is over taken by writing shell scripts such as **`rcmd`** (remote command) and **`rsync`**, which could update all file systems. Following were the steps used to configure NFS:
   - We selected dual1 as **server node** and exported the `/home` partition on additional 30GB disk and `/usr/local` partition by modifying the `/etc/exports` as:
     `/etc/exports` look like:
     ```
     /home        192.168.1.*(rw,no_root_squash)
     /usr/local   192.168.1.*(ro,no_root_squash)
     ```
   - We use automounter (**`autofs`**) for mounting the exported /home partition on the **client nodes**. Fist we installed rpm of `autofs` from the RedHat Linux 7.2 CD. The `autofs` automatically mounts the various user partitions on demand. Automounter gives better NFS performance. For configuring `autofs`, modified `/etc/auto.master` and `/etc/auto.home` files by adding following lines:
     ```
     /etc/auto.master:
     /home   /etc/auto.home
     /etc/auto.home:
     * dual1:/home/&
     ```
   - For mounting `/usr/local` file system on **client nodes**, we used traditional NFS mount, i.e. by appending `/etc/fstab` files with the following additional line:
     ```
     dual1:/usr/local   /usr/local nfs   defaults   1 2
     ```
4. Network Information System configuration: The Network Information Service (NIS) is an administrative database that provides central control and automatic dissemination of important administrative files. NIS converts several standard UNIX files into databases that can be queried over the network. The databases are called NIS maps. Following are the main steps for configuring NIS:
5. Configuration of NIS **master server**: We selected NIS domain name “dual” by issuing the command **`authconfig`** and the options that we choose are as follows
   ```
   [*] Use NIS
   Domain: dual
   ```
   - Then we initiated the yp services on server node:
     ```
     #/etc/init.d/ypserv start
     #/etc/init.d/yppasswd start
     ```
   - It asked for the NIS server name. We named the NIS server as dual1 and followed the following steps:
     ```
     #cd /var/yp
     #make
     ```
   - To start the NIS services and bind the **client nodes** with NIS server we start **`ypbind`** as
     ```
     #/etc/rc.d/init.d/ypbind start
     ```
   - These steps configured the NIS. To check the NIS services, try these

```
#ypcat passwd
#ypmatch username passwd
```

6. BIOS configuration: For booting machines without monitor, keyword and mouse, BIOS was configured on all the machines. We connected the monitor, mouse and keyboard to the nodes and configured the BIOS for no halt in the absence of keyboard, mouse and monitor.
7. Message passing libraries installation: Massage Passing Interface (MPI) installation was automatic. During the selection of packages, we selected clustering tools and it installed in `/usr/bin/mpicc` and `/usr/bin/mpif77`. PVM installation: Parallel Virtual Machine (PVM) installation was automatic. During the selection of packages, we selected clustering tools and it installed in `/usr/share/pvm3`. Later we moved pvm3 directory to `/usr/local/pvm3`.
8. XPVM installation: X-windows version of PVM also installed automatically like PVM.
9. Scripts for single system image: We have written few scripts such as **`rcmd`**, **`rsync`** etc., for monitoring the machine from console on command line.

3.3. Channel Bonding

This provides a low cost parallel computing system, but one that is much more unbalanced than traditional MPP systems that use similar processors but have communication systems that are an order of magnitude faster. This limits the types of applications that are suitable for PC clusters. Faster networking such as Gigabit Ethernet can be used to connect the PCs, but this usually doubles the cost of the cluster and only provides a little better performance since the internal memory bandwidth limits the flow of data to the PCI bus. Channel bonding as shown in Figure 6, is a method where the data in each message gets striped across the multiple network cards installed in each machine. Figure 5 and Figure 7 show our PC cluster connected with 3 network cards per machine and three Fast Ethernet switches. The channel bonding can be set by the following steps for each machine:

```
#!/bin/bash

#hwadds=`/sbin/ifconfig | grep eth0 |  awk '{print $5}'`
hwadds=`/sbin/ifconfig | /bin/grep eth0 | /bin/sed -e 's/eth0
Link encap:Ethernet  HWaddr //'  `
echo $hwadds

ipadds=`/sbin/ifconfig eth0 | /bin/grep "inet addr" | /bin/sed -e
's/inet addr://' | /bin/sed -e 's/  Bcast:192.168.1.255  Ma
sk:255.255.255.0//' `
echo $ipadds

/sbin/insmod bonding
/sbin/ifconfig eth0 down
/sbin/ifconfig eth1 down
/sbin/ifconfig eth2 down
/sbin/ifconfig bond0 $ipadds hw ether $hwadds netmask 255.255.255.0
broadcast 192.168.1.255 up
/sbin/ifenslave bond0 eth0
/sbin/ifenslave bond0 eth1
/sbin/ifenslave bond0 eth2
#mount -n -t proc proc /proc
#echo 0x00ff > /proc/sys/kernel/real-root-dev
```

These tests were done on two dual 1050 MHz Pentium-III SMPs. 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 8 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 four cases are considered: One measuring the round trip time between two processors in the same SMP (case 4); one measuring the time between two processors in the different SMP by using one NIC card (case 1); and the others measuring the time between two processors in the different SMP with channel bonded by using two (case 2) and three (case 3) NICs, respectively. Case one incurs a cost of 21.2 microseconds for a 524288-byte message; whereas the same messages take 102.2 ms, 58.9 ms, and 44.5 ms for cases 1, 2, and 3, respectively.

The graph below demonstrates that channel bonding 2 Fast Ethernet cards per PC doubles the communication rate while only adding about 10% to the overall cost of the cluster. Gigabit Ethernet is limited by the main memory bus in the PCs, and provides only 3 times the rate of Fast Ethernet but 10 times costs per network card in the cluster. Adding a third or fourth Fast Ethernet card to each machine produces little benefit, again due to the limited main memory bandwidth. In general, channel bonding can improve the communication bandwidth and reduce the communication time for larger messages.

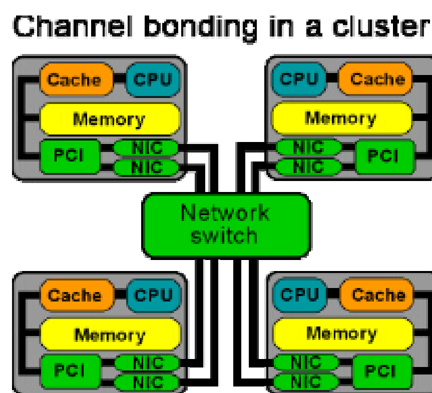

Figure 6: The hardware diagram of channel bonding in a cluster

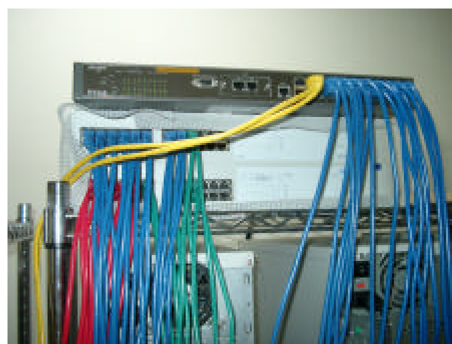

Figure 7: Apply channel bonding technique by using three switches

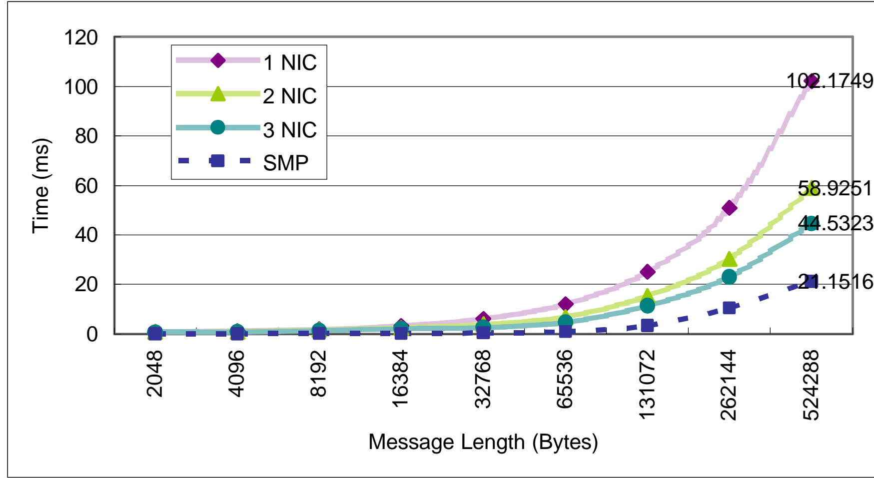

Figure 8: Performance of channel bonding on cluster

4. Performance Evaluation

4.1. Matrix Multiplications

The biggest price we had to pay for the use of a PC cluster was the conversion of an existing serial code to a parallel code based on the message-passing philosophy. The main difficulty with the message-passing philosophy is that one needs to ensure that a control node (or master node) is distributing the workload evenly between all the other nodes (the compute nodes). Because all the nodes have to synchronize at each time step, each PC should finish its calculations in about the same amount of time. If the load is uneven (or if the load balancing is poor), the PCs are going to synchronize on the slowest node, leading to a worst-case scenario. Another obstacle is the possibility of communication patterns that can deadlock. A typical example is if PC A is waiting to receive information from PC B, while B is also waiting to receive information from A. To avoid deadlocking, one needs to use a master/slave programming methodology.

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 n^3 multiplications and n^3 additions, leading to a sequential time complexity of $O(n^3)$. 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., n^2 processors are needed.

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

The matrix multiplication was run with forking of different numbers of tasks to demonstrate the speedup. The problem sizes were 256×256, 512×512, 1024×1024, and 2048×2048 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 by using sixteen processors with and without channel bonding were listed in Figure 9, respectively. In Figure 10, the corresponding speedup is increased for different problem sizes by varying the number of slave programs. With channel-bonded technique by using 3 NICs, the higher speedup was measured about 13.03 than 8.32 without channel bonding.

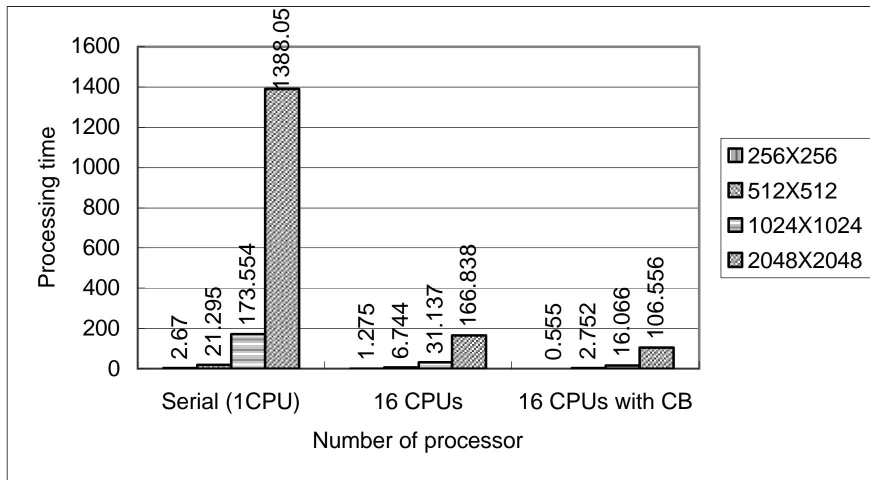

Figure 9: The processing time of matrix multiplication with and without channel bonding (3 NIC)

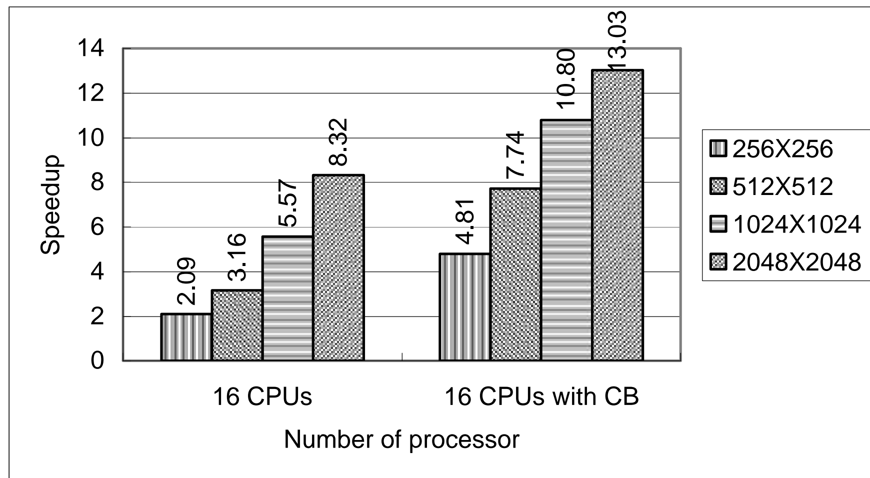

Figure 10: The speedup of matrix multiplication with and without channel bonding (3 NIC)

4.2. High Performance Linpack

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.

The benchmark used in the LINPACK Benchmark is to solve a dense system of linear equations. For the TOP500, we used that version of the benchmark that allows the user to scale the size of the problem and to optimize the software in order to achieve the best performance for a given machine. This performance does not reflect the overall performance of a given system, as no single number ever can. It does, however, reflect the performance of a dedicated system for solving a dense system of linear equations. Since the problem is very regular, the performance achieved is quite high, and the performance numbers give a good correction of peak performance. In an attempt to obtain uniformity across all computers in performance reporting, the algorithm used in solving the system of equations in the benchmark procedure must confirm to the standard operation count for LU factorization with partial pivoting. In particular, the operation count for the algorithm must be 2/3 $n^3 + O(n^2)$ floating point operations. This excludes the use of a fast matrix multiply algorithm like "Strassan's Method".

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 512 MB of memory on each, this corresponds to 4 GB 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 11. Our P-III SMP cluster can achieve 17.38Gflop/s for the problem size 32000×32000 with channel bonded by using 36 P-III processors. In Figure 12, we can find that more system speed can be obtained when channel bonding is used.

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: 1×4, 1×8 and 2×4. For example, in Figure 13, we can found that the case of 4×4 always got more computational speed than both cases of 2×8 and 8×2 by using a 16-processor cluster.

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. Form Figure 14 and Figure 15, we can find the cluster gained more system speed when the NB is increased form 32 to 96.

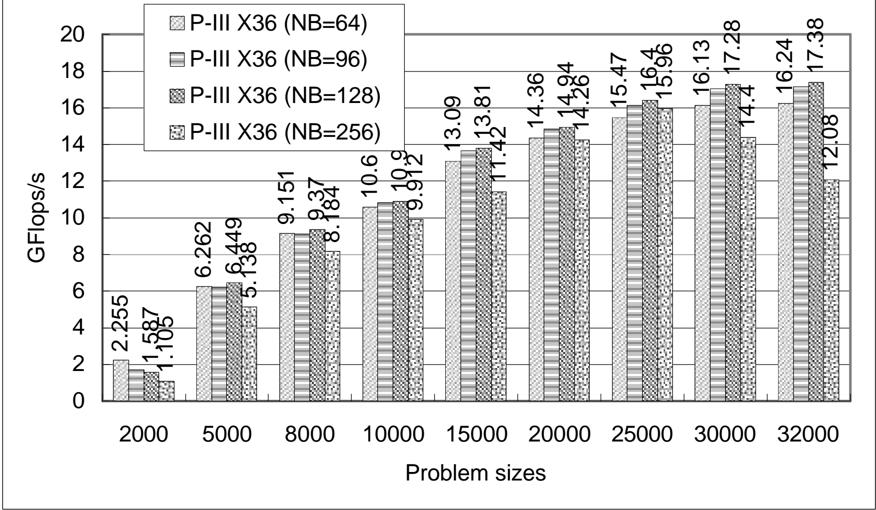

Figure 11: The system performance gained from THPTB.

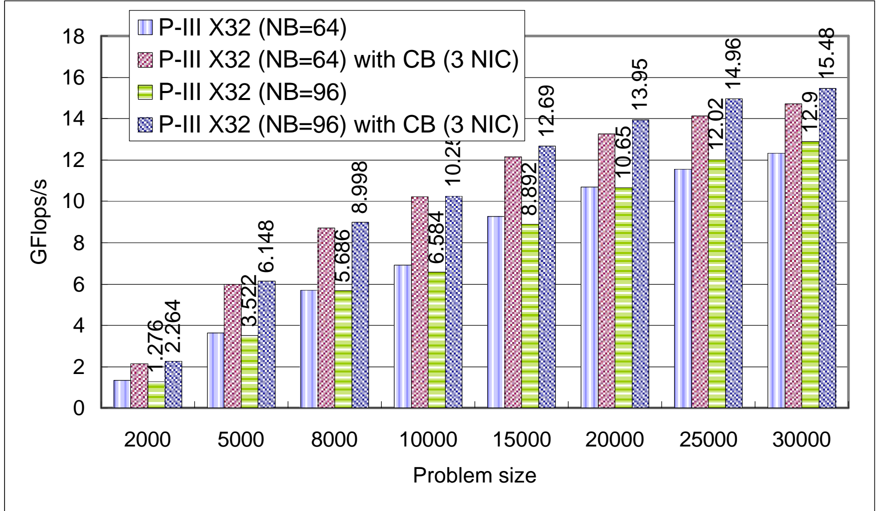

Figure 12: High performance results gained form channel bonding.

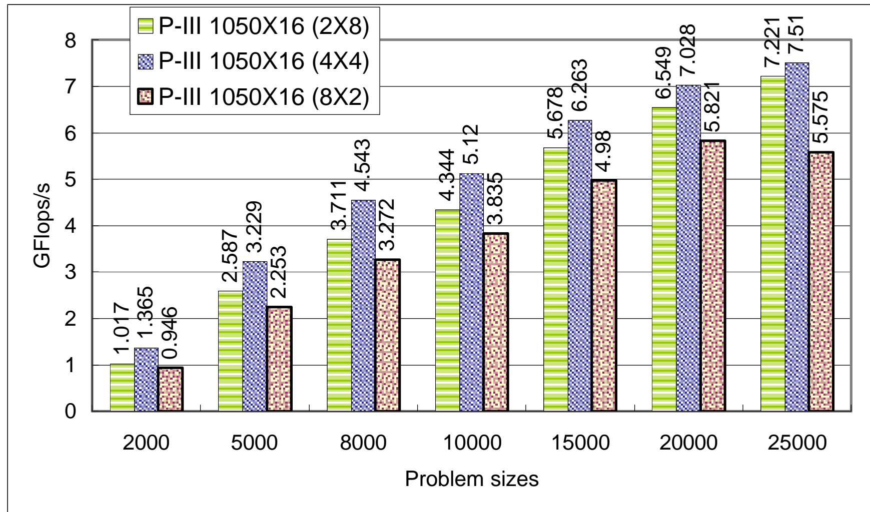

Figure 13: The performance of HPL with the different case of P×Q

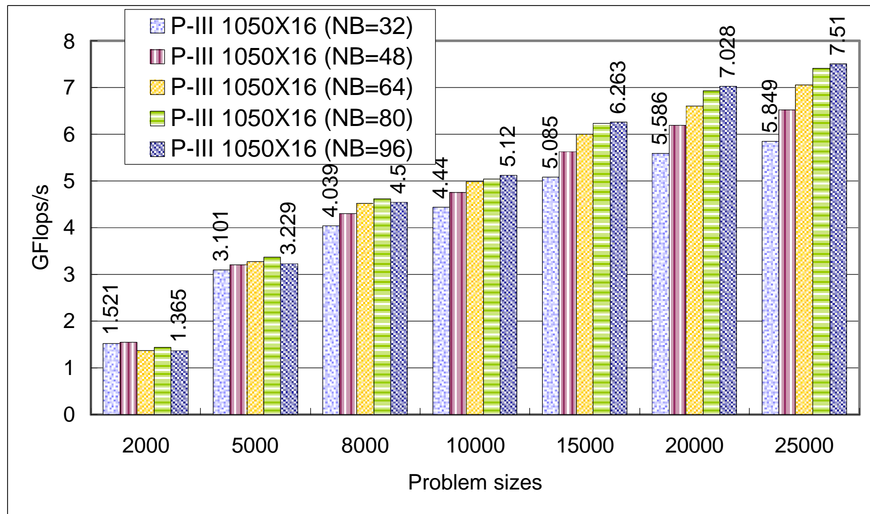

Figure 14: The performance of different sizes of NB on 16 processors

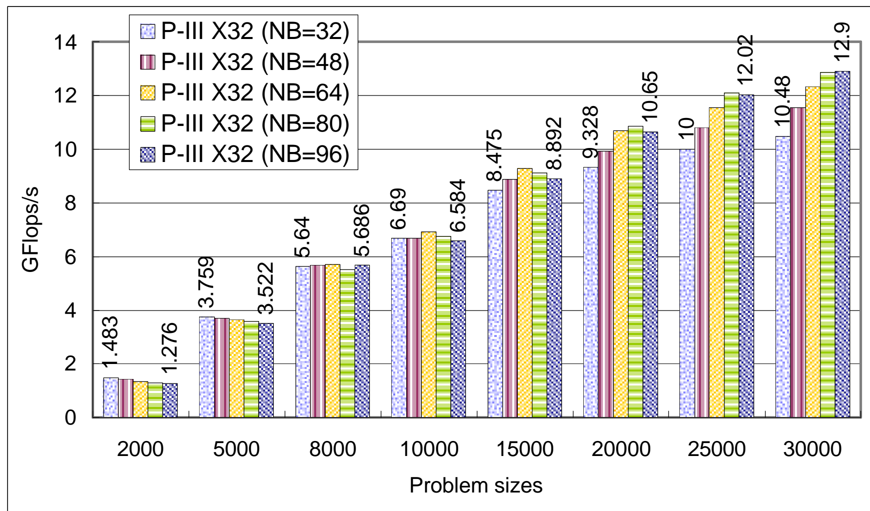

Figure 15: The performance of different sizes of NB on 32 processors

4.3. PVMPOV for Parallel Rendering

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 16.

```
$pvmpov +iskyvase.pov +w640 +h480 +nt32 -
L/home/ct/pvmpov3_1g_2/povray31/include
$pvmpov +ifish13.pov +w640 +h480 +nt32 -
L/home/ct/pvmpov3_1g_2/povray31/include
$pvmpov +ipawns.pov +w640 +h480 +nt32 -
L/home/ct/pvmpov3_1g_2/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, the parameter "`+nt32`" will start 32 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 P-III processor mode of a dual processor machine for processing 1600×1280 image, the render time was 141 seconds. Using out THPTB SMP cluster (32 processors) further reduced the time to 9 seconds. The execution times for the different POVray model (Skyvase, Pawns, and Fish13) on THPTB SMP clusters were shown in Figure 17, respectively. The corresponding speedups of different problem size by varying the number of task (option: `+nt`) was shown in Figure 18. The highest speedup was obtained about 20.25 (1600×1280) for Pawns model by using 32 processors.

Figure 16: Three models including skyvase, pawns, and fish13

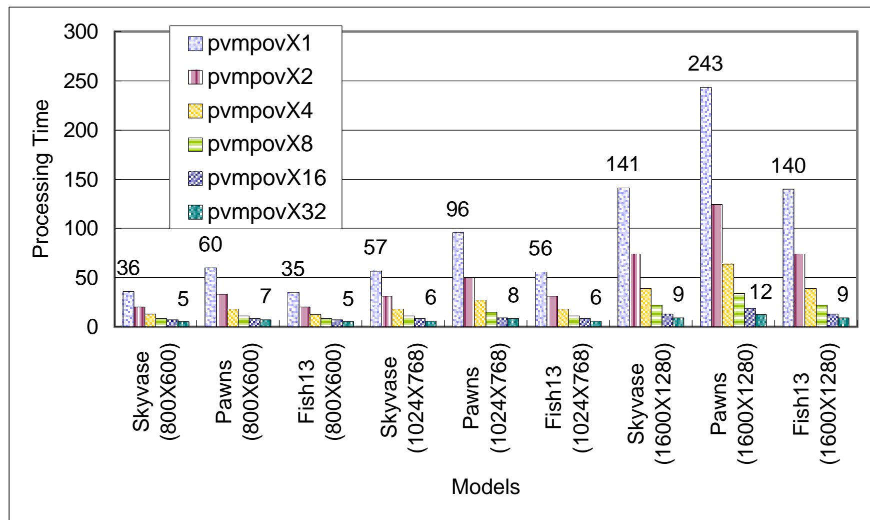

Figure 17: The processing time of three models by varying the different image sizes

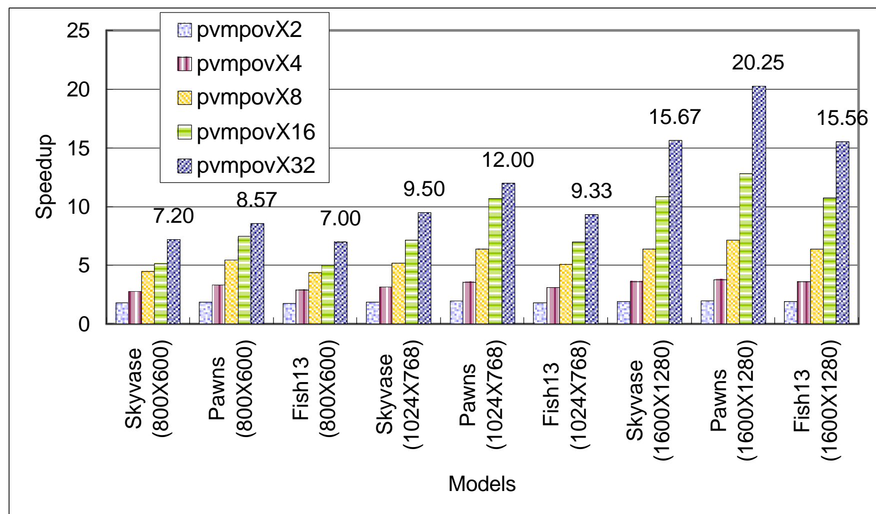

Figure 18: The speedup of three models by varying the different image sizes

5. Related Research Topics on PC Clusters

5.1. Automatic Directive-based Parallel Program Generator

Message-passing programming support may be the most obvious approach to help programmers to take advantage of parallelism on cluster. 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 directive-based parallel program generator (ADPPG) for parallelizing compiler to produce parallel object codes is shown in **Figure 19**.

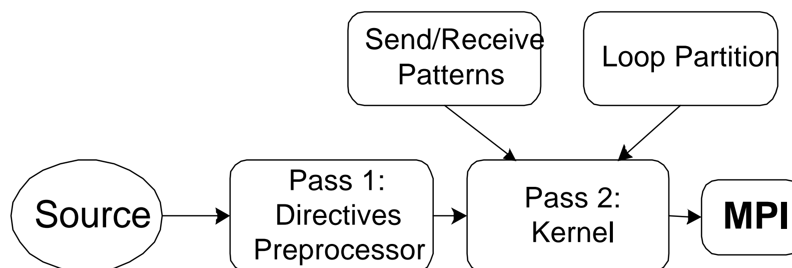

Figure 19: System structure of ADPPG

First, the automatic directive-based parallel program generator (ADPPG) 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 MPI function 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 message passing program is then compiled and linked with MPI message passing library, 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.

5.2. Dynamic Scheduling for Heterogeneous Cluster

Distributed Computing Systems are a viable and less expensive alternative to parallel computers. However, a serious difficulty in concurrent programming of a distributed system is how to deal with scheduling and load balancing of such a system which may consist of heterogeneous computers. Distributed scheduling schemes suitable for parallel loops with independent iterations on heterogeneous computer clusters have been designed in the past. In this work we consider a class of Self-Scheduling schemes for parallel loops with independent iterations which have been applied to multiprocessor systems. We extend this type of schemes to heterogeneous distributed systems. We will present tests that the distributed versions of these schemes maintain load balanced execution on heterogeneous systems in the near future.

5.3. Parallel Programming on 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.

6. Applications on PC Clusters

6.1. Remote Sensing Data Processing

There are a growing number of people who want to use remotely sensed data and GIS data. The different applications that they want to required increasing amounts of spatial, temporal, and spectral resolution. Some users, for example, are satisfied with a single image a day, while others require many images an hour. The ROCSAT-2 is the second space program initiated by National Space Program Office (NSPO) of National Science Council (NSC), the Republic of China. The ROCSAT-2 Satellite is a three-axis stabilized satellite to be launched by a small expendable launch vehicle into a sun-synchronous orbit. The primary goals of this mission are remote sensing applications for natural disaster evaluation, agriculture application, urban planning, environmental monitoring, and ocean surveillance over Taiwan area and its surrounding oceans.

The Image Processing System (IPS) refers to the Contractor-furnished hardware and software that provide the full capabilities for the reception, archival, cataloging, user query, and processing of the remote sensing image data. The IPS will be used to receive, process, and archive the bit sync remote sensing image data from the X-band Antenna System (XAS) of NSPO. The XAS is dedicated for receiving the high-rate link of the earth remote sensing data from ROCSAT-2 satellite, and has the capability of receiving downlink data rate up to 320Mbps. It will also be expanded to receive data from other remote sensing satellites. Remote sensing data comes to the IPS via either a satellite link or some other high-speed network and is placed into mass storage. Users can then process the data through some of interface.

The purpose of image rectification and restoration is to correct image for distortions or degradations that stem from the image acquisition process in geometry and radiance. The measured radiance is influenced by factors, such as changes in scene illumination, atmospheric conditions, viewing geometry, and detector response characteristics. The sources of geometric distortions include perspective, earth curvature, earth rotation, orbit inclination, atmospheric refraction, and relief displacement; aspect overlap, and variations in the altitude, attitude and velocity of the sensor platform. The parallel version of geometric correction consists of atmospheric correction, sensor response characteristics correction, aspect ration correction, earth rotation skew correction, image orientation to North-South, and correction of panoramic effects [8, 9, 10]. Consequently to produce a geometrically correct image either the vertical dimension has to be expanded by this amount or the horizontal dimension must be compressed. To correct for the effect of earth rotation, it is necessary to implement a shift of pixels to the left that is dependent upon the particular line of pixels measured with respect to the top of the image. It is an inconvenience to have an image that is not oriented vertically in a north-south direction. For example, to correct orientation distortion, it will be recalled that the Landsat-5 orbits in particular are inclined to the north-south line by about 9 degrees [9]. To correct panoramic effects, the far pixels should be compressed. Finally, ground controlled points for geo-coding are applied for precise correction. This cluster will be used to perform remote sensing data processing to save the processing time in the future.

6.2. Bioinformatics

Some document surveys the computational strategies followed to parallelize the most used software in the bioinformatics area. The studied algorithms are computationally expensive and their computational patterns range from regular, such as database searching applications, to very irregularly structured patterns (phylogenetic trees). Fine- and coarse-grained parallel strategies are discussed for these very diverse sets of applications. We will outline computational issues related to parallelism, physical machine models, parallel programming approaches, and scheduling strategies

for a broad range of computer architectures. In particular, it deals with shared, distributed, and shared/distributed memory architectures.

Much of the sequence analysis involves comparing a query sequence or a pattern to a reference database. In general, the time required to complete such a search is directly proportional to the size of the database. One method of the database search algorithm optimization, as applied to the distributed cluster environment and leading to the overall improvement of the performance will be discussed in our future work.

6.3. High Availability Web Servers

Failover clustering allows Network Administrators to significantly improve quality of service levels for practically every TCP/IP based network service, such as Web, Mail, News, and FTP. Unlike distributed processing clusters (Beowulf Clusters), high-availability clusters seamlessly and transparently integrate existing stand-alone, non-cluster aware applications together into a single virtual network, providing the architectural framework necessary to allow the network to continuously and effortlessly grow to meet performance and reliability demands.

Load balancing clusters integrate multiple systems that share the load of incoming requests in an equitably distributed manner. The systems do not work together on a single process, but rather handle incoming requests independent of one another. This type of cluster is especially suited to ISP and e-commerce applications that require real-time resolution of many incoming requests.

Additionally, in order for a cluster to be scalable, it must ensure that each server is fully utilized. The standard technique for accomplishing this is load-balancing. The basic idea behind load balancing is that by distributing the load proportionally among all the servers in the cluster, the servers can each run at full capacity, while all requests receive the lowest possible response time. In a web server scenario, load-balancing refers to the technique of routing user requests over a certain number of networked computers, so as to keep the average usage of any system's resource approximately the same within that network that acts as a functional unit.

Requests for service between nodes in the cluster are achieved using dedicated hardware switches and specialized load balancing software, such as Resonate Central Dispatch. Load balancing software services include:

- Recognizes failure detection, automatic failover & reintegration into the cluster upon recovery
- Service monitoring for web, email, file transfer, e-commerce, and TCP/IP applications
- Data replication for automatic distribution of web and file content
- Load balancing solutions for highly responsive applications
- Distributed operation & centralized management

Currently, we setup a PC cluster for high availability and load balancing usage. Our LVS PC cluster offers three types of load-balancing cluster solutions as show in. Each utilizes the highest-quality, performance-optimized system of load balancing hardware, software and components.

- Least Connections: This technique routes requests to the server that is currently handling the smallest number of requests/connections. For example, if Server 1 is currently handling 50 requests, and Server 2 is currently handling 25 requests/connections, the next request/connection will be automatically routed to Server 2, since that server currently has the least number of active connections/requests.
- Round Robin: This technique routes requests to the next available server on a rotating basis. For example, incoming requests/connections are routed to Server 1, then Server 2, and finally Server 3 before starting again with Server 1.
- Weighted Fair: This technique routes requests to servers based upon their current request load and their performance capabilities. For example, If Server 1 is four times faster at handling requests than Server 2; the administrator factors this difference by routing 4 times as many performance loads to Server 1 than Server 2.

6.4. High Performance File Systems

In recent years the disparity between I/O performance and processor performance has led to I/O bottlenecks in many applications, especially those using large data sets. A popular approach for alleviating this kind of bottleneck is the use of parallel file systems. Parallel Virtual File System

(PVFS) developed at Clemson University that supports the UNIX I/O interface and allows existing UNIX I/O programs to use PVFS files without recompiling. The familiar UNIX file tools (ls, cp, rm, etc.) will all operate on PVFS files and directories as well. This is accomplished via a Linux kernel module which is provided as a separate package. Currently, we install, integrate, and test PVFS on our cluster by using six machines.

PVFS includes scripts and test applications are included to help with configuration, testing for correct operation, and performance evaluation. PVFS stripes file data across multiple disks in different nodes in a cluster. By spreading out file data in this manner, larger files can be created, potential bandwidth is increased, and network bottlenecks are minimized. PVFS provides the following:

- Compatibility with existing binaries
- Ease of installation
- User-controlled striping of files across nodes
- Multiple interfaces, including a MPI-IO interface via ROMIO

7. Conclusions

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, a SMP-based PC cluster (36 processors), called THPTB (TungHai Parallel TestBed) with channel bonded technique, was proposed and built. The system architecture and benchmark performances of the cluster are also presented in this paper. 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 13.03 for matrix multiplication, when the total number of processors is 16 with channel bonded, by creating 16 tasks on SMPs cluster. Furthermore, the benchmark, HPL is used to demonstrate the performance of our parallel testbed by using MPI. The experimental results show that our cluster can obtain 17.38 GFlops/s for HPL programs with channel bonded, when the total number of processors used is 36. 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.

8. 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://parlweb.parl.clemson.edu/pvfs, *Parallel Virtual File System*.

[7] http://www.epm.ornl.gov/pvm, *PVM – Parallel Virtual Machine*.

[8] Lie, W. N., *Distributed Computing Systems for Satellite Image Processing*, Technical Report, EE, National Chung Cheng University, 2001.

[9] Lillesand, Thomas M. and Kiefer, Ralph W., *Remote Sensing and Image Interpretation*, Third Edition, John Wiley & Sons, 1994.

[10]Richards, John A., *Remote Sensing Digital Image Analysis: An Introduction*, Springer-Verlag, 1999.

[11]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.

[12]B. Wilkinson and M. Allen, *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*, Prentice Hall PTR, NJ, 1999.

[13]M. Wolfe, High-Performance Compilers for Parallel Computing, Addison-Wesley Publishing, NY, 1996.

[14]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.

[15]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.

[16]Chao-Tung Yang, Chi-Chu Hung, and Chia-Cheng Soong, "Parallel Computing on Low-Cost PC-Based SMPs Clusters," *Proc. of the 2001 International Conference on Parallel and Distributed Computing, Applications, and Techniques* (*PDCAT 2001*), Taipei, Taiwan, pp 149-156, July 2001.

[17]Chao-Tung Yang and Chi-Chu Hung, "High-Performance Computing on Low-Cost PC-Based SMPs Clusters," *Proc. of the 2001 National Computer Symposium* (*NCS 2001*), Taipei, Taiwan, pp 149-156, Dec. 2001.