Tunghai Science Vol. 4: 103–125
July, 2002

Using a PC Cluster for High-Performance Computing and Applications

Chao-Tung Yang*

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 in CSIE. 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 by using message-passing libraries, the HPL benchmark is used to demonstrate the performance. The experimental results show that our cluster can obtain 17.38 GFlops/s with channel bonding, when the total number of processor used is 36.

Keywords: PC cluster, parallel computing, SMP, message passing

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, 6, 7, 8, 9, 10, 11, 12].

Scalable computing clusters, ranging from a cluster of (homogeneous or heterogeneous)

* Department of Computer Sciences & Information Engineering, Tunghai University, Taichung 407, TAIWAN

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 shown 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 is 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 is originated at the Center of Excellence in Space Data and Information Sciences (CESDIS), located at the NASA Goddard Space Flight Center in Maryland [6]. 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.

Beowulf is a concept of clustering commodity computers to form a parallel, virtual supercomputer. It is easy to build a unique Beowulf cluster from components that you consider most appropriate for your applications. Such a 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. As the figure illustrates, numerous design choices exist for building a Beowulf cluster. For, example, the blue bold line indicates our cluster configuration from bottom to top. No Beowulf cluster is general enough to satisfy the needs of everyone. We present some considerations as below, derived from CSIE's experience for selecting compute nodes.

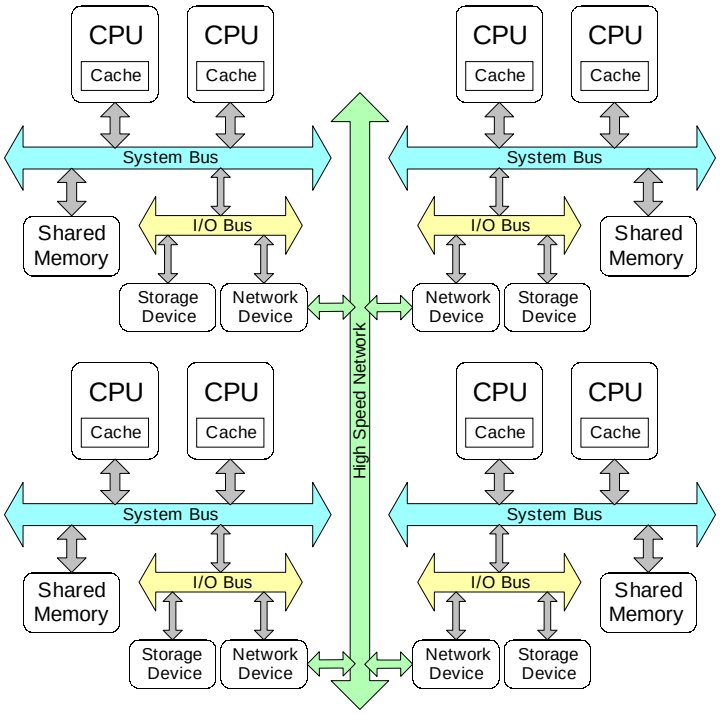

Figure 1: A cluster system by connecting four SMPs.

- Compute node qualities to consider include processor type, size and speed of Level 2 (L2) cache, number of processors per node, speed of front-side bus (FSB), scalability of memory subsystem, and Peripheral Component Interconnect (PCI) bus speed. Some parallel applications are cache-friendly; that is, the problem can be easily accommodated by L2 cache. Compute nodes with large full-speed L2 cache can boost the performance of these applications. On the other hand, applications with random and wide-range memory access patterns will be more likely to benefit from faster processor speed, system bus, and memory subsystem, rather than large L2 cache.
- Interconnects and communication protocols: Unless the application can be nicely partitioned and only modest levels of communication are required, high-speed interconnects unveil the potential performance of a Beowulf cluster. We have observed that high-speed interconnects, such as GigaNet® and Myrinet, not only facilitate higher cluster performance than Fast Ethernet and Gigabit Ethernet, but also achieve a better performance/cost ratio.

- Operating systems: The majority of Beowulf clusters are based on Linux. We also found others that took advantage of the multithreading feature of Windows NT and Windows 2000 for building Beowulf-like Windows NT clusters.
- Middleware: Some of the MPI implementations use a polling scheme to achieve ultra-low communication latency; others minimize the CPU cycles used by communication tasks with an interrupt-driven approach. The former is more suitable for applications with a fine-grain communication pattern, while the latter helps the majority of medium-grain and course-grain parallel applications.
- Application development tools: Many numeric-intensive parallel applications can achieve better performance by using proper optimization options of compilers and optimized common subroutines.

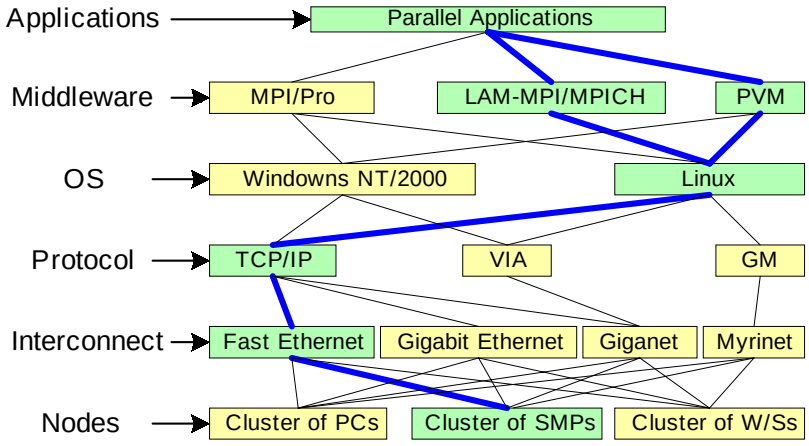

Figure 2: The architecture of cluster systems.

These systems are scalable, i.e., they can be tuned to available budget and computational needs and allow efficient execution of both demanding sequential and parallel applications [1, 2]. Beowulf clusters offer a number of specific benefits:

- Cost-effective: One of the main benefits of a Beowulf cluster is its cost-effectiveness. Beowulf clusters are built from relatively inexpensive commodity components that are widely available.

- Keeps pace with technologies: Since Beowulf clusters only use mass-market components; it is easy to employ the latest technologies to maintain the cluster as a state-of-the-art system.
- Flexible configuration: Users can tailor a configuration that is feasible to them and allocate the budget wisely to meet the performance requirements of their applications. For example, a fine-grain parallel application (which exchange small messages frequently among processors) may motivate users to allocate a larger portion of their budget to high-speed interconnects.
- Scalability: When the processing power requirement increases, the performance and size of a Beowulf cluster can be easily scaled up by adding more compute nodes.
- High availability: Each compute node of a Beowulf cluster is an individual machine. The failure of a compute node will not affect other nodes or the availability of the entire cluster.
- Compatibility and portability: Due to the standardization and wide availability of message passing interface, such as MPI [4] and PVM [5], the majority of parallel applications use these standard middleware. A parallel application using MPI or PVM can be easily ported to a Beowulf cluster. This is why Beowulf-class clusters are rapidly replacing these expensive parallel computers in the low-end to midrange HPC market.

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, 6]. 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.

In this paper, the system architecture and benchmark performances of the cluster are presented. In order to measure the performance of our cluster, the matrix multiplication problem is illustrated and the experimental result is demonstrated on our Linux SMPs cluster. The experimental results show that the highest speedup is 13.03 with channel bonded, when the total numbers of processor is 16 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.

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.

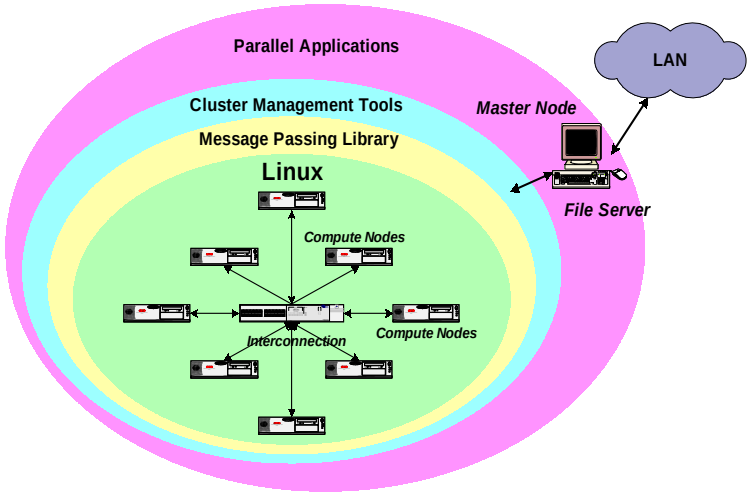

Figure 3: Logical view of Beowulf cluster.

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 limited budget; 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). 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.

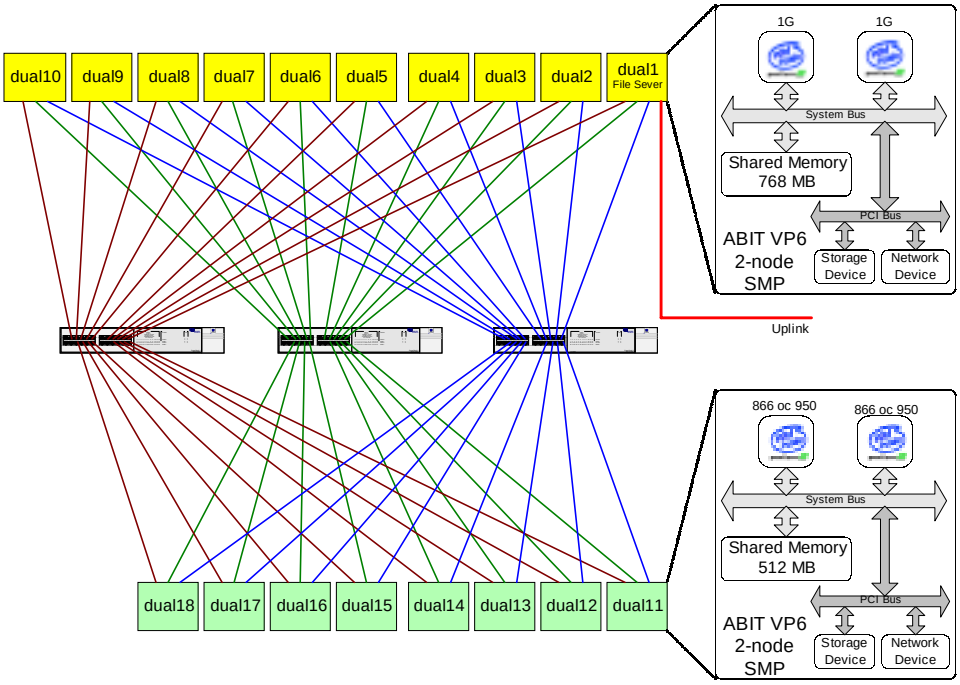

Figure 5: The network topological view of THPTB cluster.

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 [5]. 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 [4]. 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. Performance Evaluation

3.1 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 $\boldsymbol{O}(n^3)$. 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 6, respectively. In Figure 7, 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 8.58 without channel bonding.

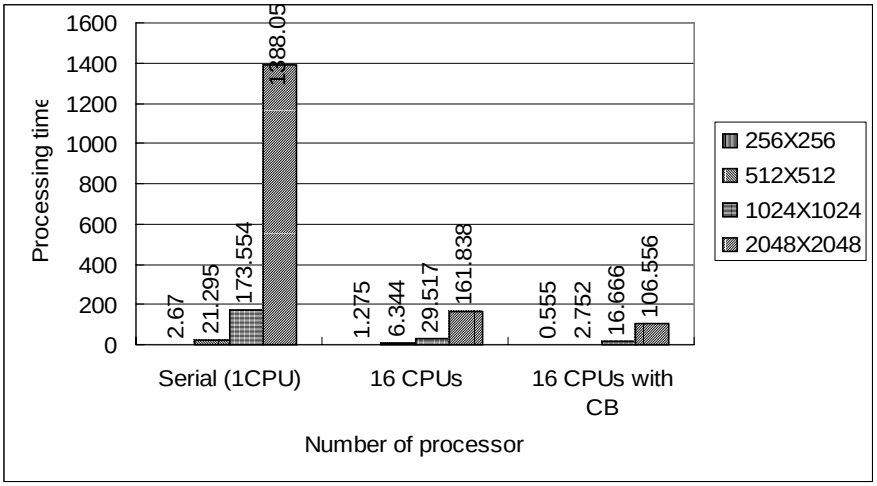

Figure 6: The processing time of matrix multiplication with and without channel bonding (3 NIC)

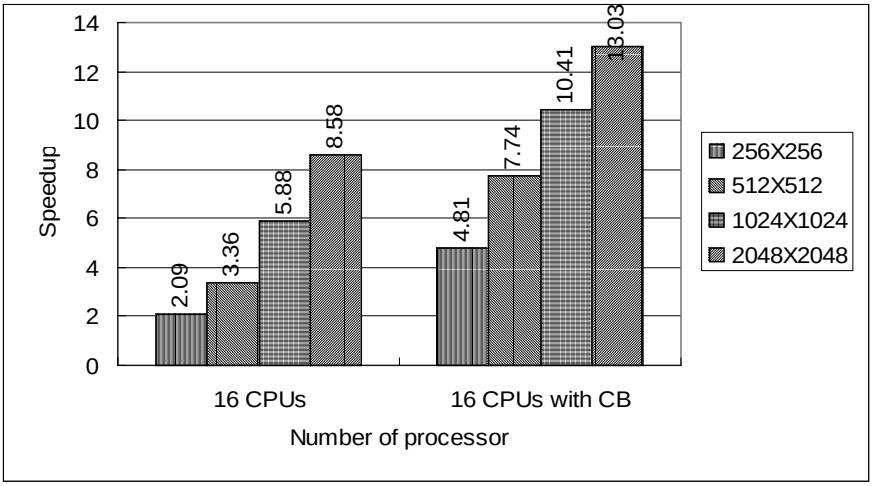

Figure 7: The speedup of matrix multiplication with and without channel bonding (3 NIC)

3.2 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 are 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 8. Our P-III SMP cluster can achieve 17.38Gflop/s for the problem size 32000×32000 with channel bonded (3 NIC) by using 36 P-III processors. In Figure 9, we can find that more system speed can be obtained when channel bonding is used.

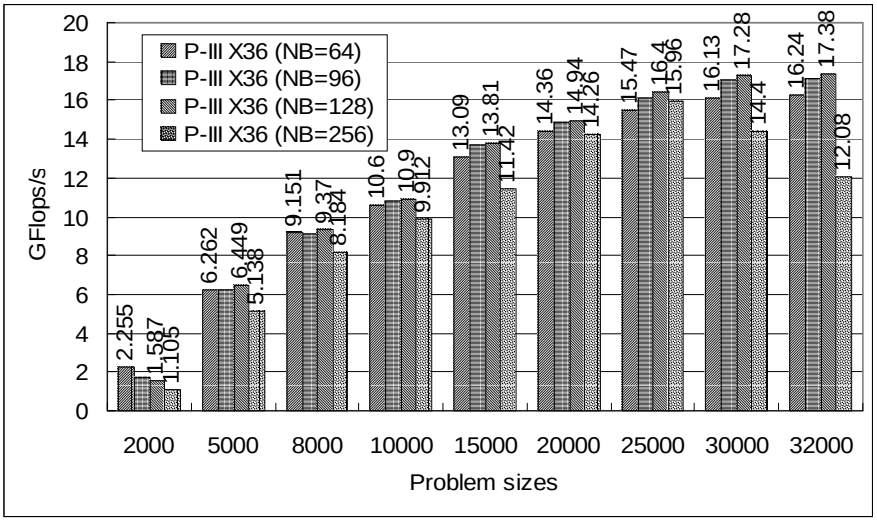

Figure 8: The system performance gained from THPTB with channel bonding.

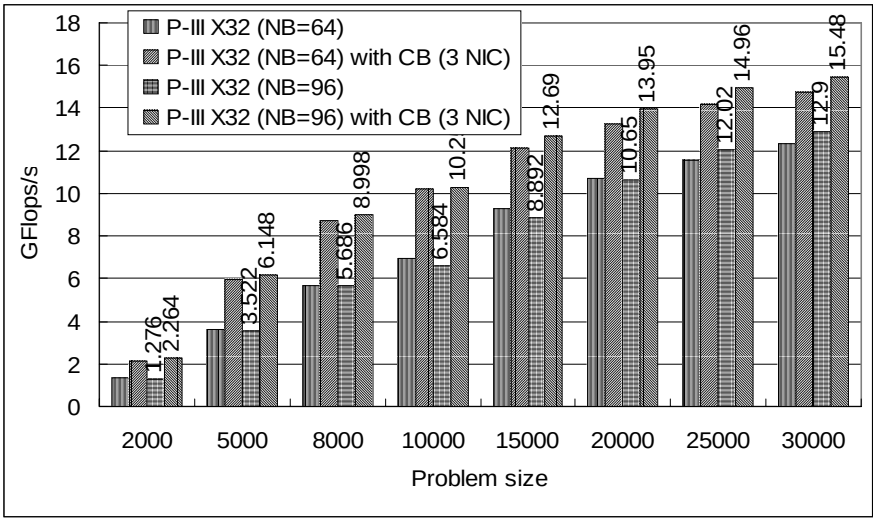

Figure 9: High performance results gained form channel bonding.

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 10, 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 11 and Figure 12, we can find the cluster gained more system speed when the NB is increased form 32 to 96.

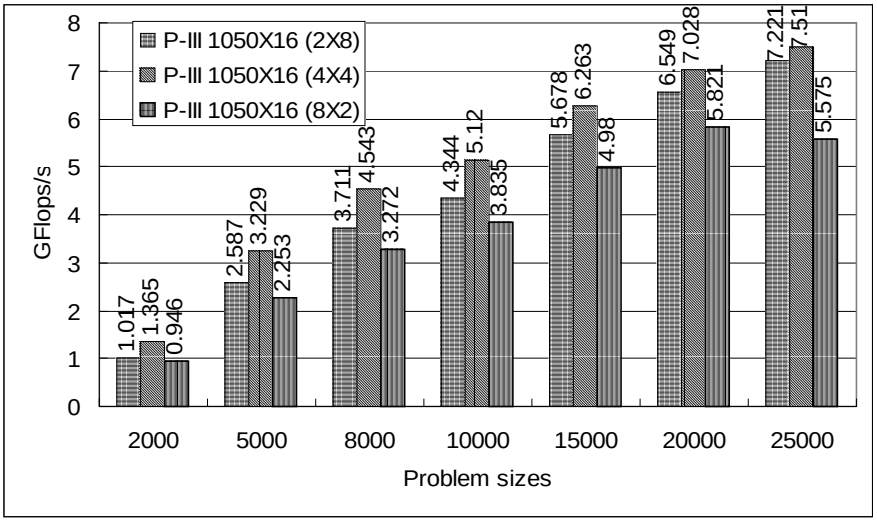

Figure 10: The performance of HPL with the different case of P×Q

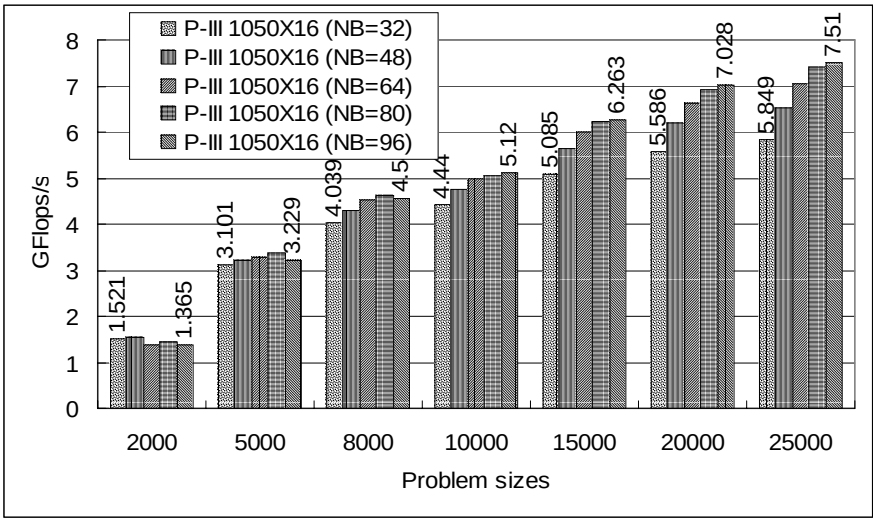

Figure 11: The performance of different sizes of NB on 16 processors

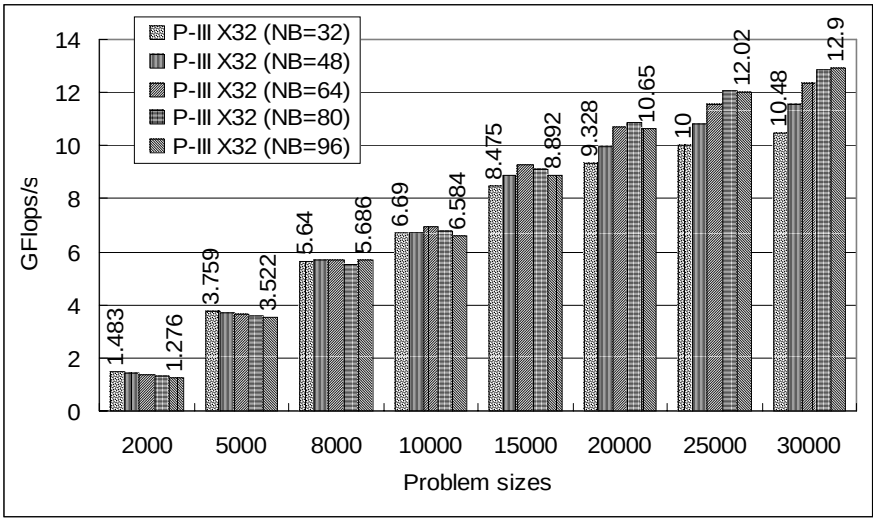

Figure 12: The performance of different sizes of NB on 32 processors

4 Related Research Topics on PC Clusters

4.1. 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 13.

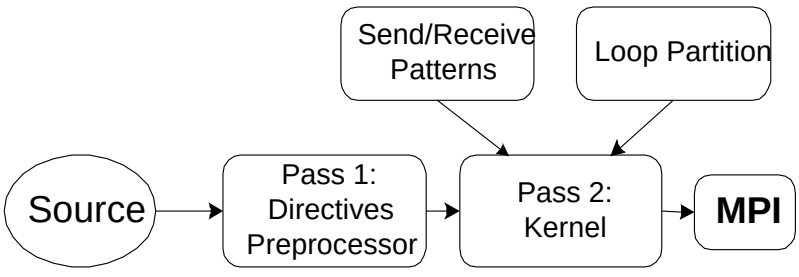

Figure 13: System structure of ADPPG

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 ADPPG 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.

4.2. 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.

4.3. 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.

5 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. 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 LAM/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.

References

[1] Buyya, R. (1999) *High Performance Cluster Computing: System and Architectures*, Vol. 1, Prentice Hall PTR, NJ.

[2] Buyya, R. (1999) *High Performance Cluster Computing: System and Architectures*, Vol. 2, Prentice Hall PTR, NJ.

[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.epm.ornl.gov/pvm, *PVM – Parallel Virtual Machine*.

[6] Sterling, T.L., Salmon, J., Backer, D.J., and Savarese, D.F.(1999) *How to Build a Beowulf: A Guide to the Implementation and Application of PC Clusters*, 2nd Printing, MIT Press, Cambridge, Massachusetts, USA.

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

[8] Wolfe, M. (1996) *High-Performance Compilers for Parallel Computing*, Addison-Wesley Publishing, NY.

[9] Yang, C.-T., Tseng, S.-S., Hsiao, M.-C., and Kao, S.-H. (1999) "A Portable parallelizing compiler with loop partitioning," *Proc. of the NSC ROC(A)* **23**(6), pp. 751-765.

[10] Yang, C.-T., Tseng, S.-S., Fan, Y.-W., Tasi, T.-K., Hsieh, M.-H., and Wu, C.-T. (2001) "Using Knowledge-based Systems for research on portable parallelizing compilers," *Concurrency and Computation: Practice and Experience* **13**, pp. 181-208.

[11] Yang, C.-T., Hung, C.-C., and Soong, C.-C. (2001) "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.

[12] Yang, C.-T., and Hung, C.-C. (2001) "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..

使用個人電腦叢集於高效能計算與應用

楊朝棟*

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。本論文陳述本實驗室所建構的一套以 18 台對稱式多處理機系統(36 顆處理器)的個人電腦叢集，並簡介叢集式平行系統架構，軟體工具以及相關應用。

關鍵詞：個人電腦叢集、平行計算、對稱式多處理機系統、訊息傳遞程式庫。

* 高效能計算實驗室，東海大學資訊工程與科學系