

University of Montana

ScholarWorks at University of Montana

Graduate Student Theses, Dissertations, &
Professional Papers

Graduate School

2001

Implementation of parallel-distributed computation under load balancing and fault tolerance

Gang Wu

The University of Montana

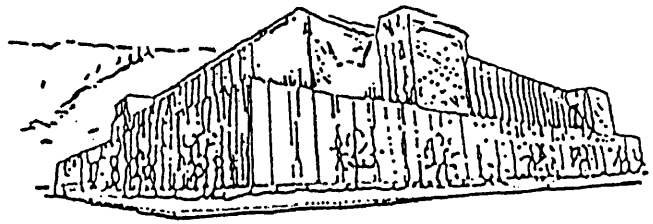
Follow this and additional works at: <https://scholarworks.umt.edu/etd>

Let us know how access to this document benefits you.

Recommended Citation

Wu, Gang, "Implementation of parallel-distributed computation under load balancing and fault tolerance" (2001). *Graduate Student Theses, Dissertations, & Professional Papers*. 5123.
<https://scholarworks.umt.edu/etd/5123>

This Thesis is brought to you for free and open access by the Graduate School at ScholarWorks at University of Montana. It has been accepted for inclusion in Graduate Student Theses, Dissertations, & Professional Papers by an authorized administrator of ScholarWorks at University of Montana. For more information, please contact scholarworks@mso.umt.edu.



Maureen and Mike
MANSFIELD LIBRARY

The University of **MONTANA**

Permission is granted by the author to reproduce this material in its entirety, provided that this material is used for scholarly purposes and is properly cited in published works and reports.

*** Please check "Yes" or "No" and provide signature ***

Yes, I grant permission
No, I do not grant permission

Author's Signature *Gene J. [unclear]*

Date 5 / 2 / 01

Any copying for commercial purposes or financial gain may be undertaken only with the author's explicit consent.

An Implementation of Parallel-Distributed Computation Under Load Balancing and Fault Tolerance

By

Gang Wu

Presented in partial fulfillment of the requirements

for the degree of

Master of Science

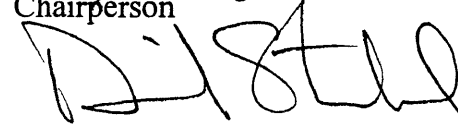
in Computer Science

The University of Montana

May 2001

Approved by:


Chairperson


Dean, Graduate School

5-18-01
Date

UMI Number: EP40587

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.

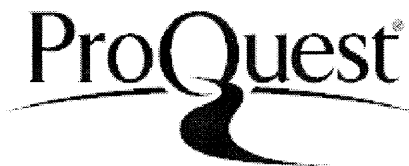


UMI EP40587

Published by ProQuest LLC (2014). Copyright in the Dissertation held by the Author.

Microform Edition © ProQuest LLC.

All rights reserved. This work is protected against unauthorized copying under Title 17, United States Code



ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

An Implementation of Parallel-Distributed Computation Under Load Balancing and Fault Tolerance

Director: Donald J. Morton, Jr. Ph.D.



A parallel-distributed computation of master-slave model for application in biophysical data analysis is designed and developed using Parallel Virtual Machine (PVM) on a set of specified hosts in this paper. To make this PVM routine robust, fault tolerance is implemented; it has the ability to monitor all tasks, if it finds a task fails without sending its result from a slave to master, the task will be spawned by master again. In order to extract the maximum performance, static load balancing and dynamic load balancing are implemented, which makes sure all processors are busy doing useful work most the time.

TABLE OF CONTENTS

ABSTRACT	ii
1. Introduction	1
1.1 Background	1
1.2 Motivation	2
2. A Brief Introduction to PVM	5
2.1 What is PVM	5
2.2 Why PVM	6
2.3 How PVM Works	6
2.4 PVM Libraries	7
2.4.1 Process Control	8
2.4.2 Information	9
2.4.3 Message Passing	10
2.4.4 Dynamic Configuration	11
2.4.5 Signaling	12
3. Design and Implementation of PVM Routine	18
3.1 Design	18
3.2 Implementation	20
4. Correctness and Efficiency of PVM Routine	30
4.1 Test	30
4.2 Results and Comparisons	31
5. Fault Tolerance	35
5.1 Why Fault Tolerance	35
5.2 How to Implement	36
5.3 Tests	38
6. Load Balancing	42
6.1 What is Load Balancing	42
6.2 How to Implement	43
6.3 Discuss	50
7. Conclusions and Future Work	51
References	54
Appendix	56

CHAPTER 1

INTRODUCTION

In this chapter, Dr. Borries Demeler's work is briefly explained, which is a background of this project. The motivation of the project and the outline of this report are described in detail.

1.1 Background

Dr. Borries Demeler is developing an integrated biophysical data analysis software package called UltraScan in the Department of Biochemistry at University of Texas Health Sciences Center, San Antonio. UltraScan software would be extremely valuable for the biophysical characterization of interacting biomolecules and the study of structure/function relationships to more complex biological systems. In order to develop the integrated biophysical data analysis, Dr. Demeler used Dosen't Use Derivatives (DuD) algorithm [1] for fitting complex models and Monte Carlo method [2] to implement rigorous statistical error analyses.

On a single computer, it is computationally expensive to perform the integrated biophysical data analysis in order to be statistically meaningful with Monte Carlo method. However, a major advantage of the integrated data analysis is that it can easily be performed in parallel by modularizing the integrated data analysis into many subprocesses that can be calculated independently of each other. The algorithm can be

structured such that the outcome of a subprocess is not dependent on the outcome of other subprocesses. So, it is possible to perform the calculation of each subprocess in parallel on multiple central processing units that can communicate with each other over a network. A control process is running on one of them, while parallelisms are computed on both the local and other nodes. Since some subprocess can be done earlier than others, in order to save time and make full use of CPUs, once the subprocess is completed in a processing unit, another subprocess will be sent to the processing unit, this procedure will be repeated until all subprocesses are finished.

The algorithm is optimized for maximal usage of all CPUs. Therefore, it is a natural and logical choice to distribute the integrated data analysis over multiple processors. The computing efficiency of the networked cluster can be further improved by manually adjusting the number of nodes and the type of simulations run on each node, depending on their complexity and the particular node's computing power. Without parallel-distributed computation, the integrated biophysical data analysis cannot be completed in a reasonable amount of time

1.2 Motivation

The parallel-distributed computation in Dr. Demeler's software package is an important part, which makes the integrated analysis be done in an acceptable time [3]. The goal of this project is to prototype a parallel-distributed computation method for implementation in Dr. Demeler's code using Parallel Virtual Machine (PVM) [4], which

would display the basic idea that Dr. Demeler will use in his data analysis routine, confirm the correctness and the efficiency of the parallel-distributed computation compared with a single processor to perform the calculation, and improve its performance on both fault tolerance and load balancing [5]. This work will be valuable towards implementing the parallel-distributed computation for Dr. Demeler's integrated data analysis routine.

PVM has many advantages, such as portability, scalable parallelism, and robust fault tolerance, it has been available for several years and became a de facto standard. Message Passing Interface (MPI) is another method for parallel computing, primarily concerned with messaging [8]. Because these subprocesses mentioned above are independent, we do not need to exchange information among them, PVM is selected to implement the parallel-distributed computation on the Scinet, which is an intra network with 4 Linux PCs, and Solaris workstations in the Department of Computer Science at the University of Montana, as well as on the intra network (22 processors) of Biochemistry Department at University of Texas Health Sciences Center, San Antonio.

In the following chapters, Chapter 2 will give a brief introduction to PVM, which includes what PVM is, why use PVM, how PVM works, and PVM libraries. Chapter 3 will describe how to design and implement the parallel-distributed computation on a set of specified nodes in the project. Chapter 4 will show how to verify correctness and efficiency of the PVM routine, and results of tests and analyses. Chapter 5 and Chapter 6

will discuss how to improve the PVM routine performance - fault tolerance and load balancing. Finally, conclusions and future work are indicated in Chapter 7.

CHAPTER 2

A BRIEF INTRODUCTION TO PVM

In the last chapter, we described why we did the project and its goal, and why we selected Parallel Virtual Machine. In this chapter, Parallel Virtual Machine is briefly introduced, including what is PVM, why use PVM, how PVM works, and PVM libraries.

2.1 What Is PVM

PVM stands for Parallel Virtual Machine. It is an open source software tool that enables execution of parallel applications across multiple nodes on a network. With this software a user can turn a loose group of machines into a parallel computer. PVM runs on most Linux/Unix machines, and on any network that supports the TCP/IP protocol. PVM can be started from any one machine. If the user supplies a host file with a list of machine names, those machines will be added to the PVM configuration at startup. A virtual console can be brought up on any host in the configuration to monitor the status of PVM. The user can issue commands from the console to add or delete hosts from the PVM, and to list active jobs.

PVM has a simple message-passing interface for exchanging data between different tasks for parallel applications. Each task is identified by a unique *task ID*. The ID of the sender and intended recipient are encoded in the message header, and the message is routed to the appropriate task by the PVM daemons on the source and destination hosts.

2.2 Why PVM

PVM is a handy, low-cost tool for parallel computing and is also supported on massively parallel computers. It can not only turn a loosely scattered, under-utilized set of Unix workstations into a powerful parallel computer, but can also be used to build a supercomputer from scratch with off-the-shelf processors.

Its portability is the key advantage of PVM. A program written in PVM can run on almost any hardware in use today, from PCs to supercomputers. This removes the hardware dependency from the application and reduces the cost for development and future upgrades. An application can also be developed on a desktop system and then moved to a supercomputer for production runs.

PVM has a small set of functions that are intuitive and easy to use. It has been available for several years and gained wide acceptance among technical users, and became a *de facto* standard.

2.3 How PVM Works

PVM consists of two main components – a daemon and library interface routines. A daemon is started on every host in the Virtual Machine. Users' programs need to be linked with the PVM library at compile time. There are three ways to start a new PVM

task: run it like any other Unix process; spawn it from the console; spawn it from another PVM task.

Normally the daemon forks and execs a new process. The process then enrolls in PVM and gets an ID from the daemon. A TCP socket connection [6] is established between the task and the daemon. The new task can query the daemon for information on other tasks and the configuration of the Virtual Machine.

When a task sends a message to another task, the message is usually routed by the local daemon. The daemon decodes the message header and forwards the message to the destination host. The daemon on that host then passes the message along to the intended recipient.

The programming model of PVM is quite simple. A unique ID identifies each task. From the programmer's point of view, it really doesn't matter where the task is running. The PVM console gives the user a global view of the Virtual Machine, commands can be issued there to query the status of any task or to send a signal to a particular task.

2.4 PVM Libraries

There are three PVM libraries: 1) libpvm3.a - Library of C language interface routines. 2) libfpvm3.a - additionally required for Fortran codes. 3) libgpvm3.a - required for use with dynamic groups.

The libraries contain simple subroutine calls that the application programmer may embed in concurrent or parallel application code, and provide ability to 1) initiate and terminate processes, 2) pack, send, receive and broadcast messages, 3) synchronize via barriers, 4) query and dynamically change configuration of the parallel virtual machine. Library routines do not directly communicate with other processes. Instead, they send commands to the local daemon and receive status information back. [Note that some PVM implementations actually allow tasks to communicate directly with each other, through “channels”. By bypassing the daemons in this manner, there is less overhead.] We will briefly talk about some important interface routines in `libpvm3.a`, which include process control routines, information routines, dynamic configuration routines, signaling routines, and message passing routines. Most of them are used in the project.

2.4.1 Process Control

Process control routines are used to control processes, such as killing processes, and spawning processes.

int tid = pvm_mytid(void)

- `pvm_mytid()` returns the `TID` of this process and can be called many times. It enrolls this process into PVM if this is the first PVM call.

int info = pvm_exit(void)

- `pvm_exit()` tells the local pvmd that this process is leaving PVM. This routine does not kill the process, which can continue to perform tasks just like any other UNIX process. Users typically call `pvm_exit()` right before exiting their C programs.

int numt = pvm_spawn(char *task, char **argv, int flag, char *where, int ntask, int *tids)

- `pvm_spawn()` starts up `ntask` copies of an executable file `task` on the virtual machine. `argv` is a pointer to an array of arguments to `task` with the end of the array specified by `NULL`. If `task` takes no arguments, then `argv` is `NULL`. The `flag` argument is used to specify options, such as `PvmTaskDefault` and `PVMTaskHost`. In this project, `PvmTaskHost` is used. On return, `numt` is set to the number of tasks successfully spawned or an error code if no tasks could be started.

int info = pvm_kill(int tid)

- `pvm_kill()` kills some other PVM task identified by `TID`. This routine is not designed to kill the calling task, which should be accomplished by calling `pvm_exit()` followed by `exit()`.

2.4.2 Information

Information routines provide message about the virtual machine and the PVM tasks running on the virtual machine.

int tid = pvm_parent(void)

- `pvm_parent()` returns the TID of the process that spawned this task or the value of `PvmNoParent` if not created by `pvm_spawn()`.

*int info = pvm_config(int *nhost, int *narch, struct pvmhostinfo **hostp)*

- `pvm_config()` returns information about the virtual machine including the number of hosts, `nhost`, and the number of different data formats, `narch`. `hostp` is a pointer to a user declared array of `pvmhostinfo` structures. The array should be of size at least `nhost`. On return, each `pvmhostinfo` structure contains the pvm TID, host name, name of the architecture, and relative CPU speed for that host in the configuration. To do load balancing, we need to know how many hosts are used in a virtual machine, so `pvm_config()` is used to return the number of hosts.

2.4.3 Message Passing

Message passing routines are used to initialize a send buffer, pack a message into a buffer, send, receive and unpack the message. Sending a message consists of three steps in PVM.

First, a send buffer must be initialized by a call to `pvm_initsend()` or `pvm_mkbuf()`.

Second, the message must be “packed” into this buffer using any number and combination of `pvm_pk*()` routines.

Third, the completed message is sent to another process by calling the `pvm_send()` routine or multicast with the `pvm_mcast()` routine.

A message is received by calling either a blocking or nonblocking receive routine and then “unpacking” each of the packed items from the receive buffer. The receive routines can be set to accept *any* message, or any message from a specified source, or any message with a specified message tag, or only messages with a given message tag from a given source. There is also a probe function that returns whether a message has arrived, but does not actually receive it.

If required, other receive contexts can be handled by PVM 3. The routine `pvm_recvf()` allows users to define their own receive contexts that will be used by the subsequent PVM receive routines. The `pvm_recv()` is blocking, which means the routine waits until a message matching the user specified `tid` and `meshtag` values arrives at the local `pvmd`. If the message has already arrived, then `pvm_recv()` returns immediately with the message.

2.4.4 Dynamic Configuration

We can use dynamic configuration routines to add or delete hosts in the virtual machine.

*int info = pvm_addhosts(char **hosts, int nhost, int *infos)*

*int info = pvm_delhosts(char **hosta, int nhost, int *infos)*

- The routines add or delete a set of `hosts` in the virtual machine. The `info` is returned as the number of hosts successfully added. The argument `infos` is an array of length `nhost` that contains the status code for each individual host being added or deleted. This allows the user to check whether only one of a set of hosts caused a problem rather than trying to add or delete the entire set of hosts again.
- These routines are sometimes used to set up a virtual machine, but more often they are used to increase the flexibility and fault tolerance of a large application. These routines allow an application to increase the available computing power (adding hosts) if it determines the problem is getting harder to solve. Another use would be to increase the fault tolerance of an application by having it detect the failure of a host and adding in a replacement host.

2.4.5 Signaling

Signaling routines can provide information about events, such as if a task exits, if a host is deleted, and if a host is added.

int info = pvm_notify(int what, int msgtag, int cnt, int tids)

- `pvm_notify` requests PVM to notify the caller on detecting certain events.

The present options are as follows:

- PvmTaskExit - one notify message for each tid requested. The message body contains a single tid of exited task.
- PvmHostDelete - one notify message for each tid requested. The message body contains a single pvmd tid of exited pvmd.
- PvmHostAdd - *cnt* notify messages are sent, one each time the local pvmd's host table is updated. The message body contains an integer length followed by a list of pvmd tids of new pvmds.

In response to a notify request, some number of messages are sent by PVM back to the calling task. The messages are tagged with the user-supplied msgtag. The tids array specifies who to monitor when using TaskExit or HostDelete. The array contains nothing when using HostAdd. If required, the routines `pvm_config` and `pvm_tasks` can be used to obtain task and pvmd tids.

Parameter *tids* is a *cnt* length array of task or pvmd tids. It specifies who to monitor when using TaskExit or HostDelete. The array is not used with the PvmHostAdd option. Specifying *cnt* of -1 turns on PvmHostAdd messages until a future notify, a count of zero disables them. Other number will be the times to notify.

If `pvm_notify()` is not successful, *info* will be less than zero. *Info* is an integer status code.

If the host on which task A is running fails, and task B has asked to be notified if task A exits, then task B will be notified even though the exit was caused indirectly by the host failure.

Example of `pvm_notify()` [4]:

```

/*
    Failure notification example
    Demonstrates how to tell when a task exits
*/

/* defines and prototypes for the PVM library */
#include <pvm3.h>

/* Maximum number of children this program will spawn */
#define MAXNCHILD    20

/* Tag to use for the task done message */
#define TASKDIED      11

int main(int argc, char* argv[])
{

    /* number of tasks to spawn, use 3 as the default */
    int ntask = 3;

    /* return code from pvm calls */
    int info;

```

```
/* my task id */
int mytid;

/* my parents task id */
int myparent;

/* children task id array */
int child[MAXNCHILD];
int i, deadtid;
int tid;
char *argv[5];

/* find out my task id number */
mytid = pvm_mytid();

/* check for error */
if (mytid < 0) {

    /* print out the error */
    pvm_perror(argv[0]);

    /* exit the program */
    return -1;
}

/* find my parent's task id number */
myparent = pvm_parent();
```

```
/* exit if there is some error other than PvmNoParent */
if ((myparent < 0) && (myparent != PvmNoParent)) {
    pvm_perror(argv[0]);
    pvm_exit();
    return -1;
}

/* if i don't have a parent then i am the parent */
if (myparent == PvmNoParent) {

    /* find out how many tasks to spawn */
    if (argc == 2) ntask = atoi(argv[1]);

    /* make sure ntask is legal */
    if ((ntask < 1) || (ntask > MAXNCHILD)) { pvm_exit(); return 0;}

    /* spawn the child tasks */
    info = pvm_spawn(argv[0], (char**)0, PvmTaskDebug, (char*)0,
        ntask, child);

    /* make sure spawn succeeded */
    if (info != ntask) { pvm_exit(); return -1; }

    /* print the tids */
```

```

for (i = 0; i < ntask; i++) printf("t%x\t",child[i]);

    putchar('\n');

/* ask for notification when child exits */
info = pvm_notify(PvmTaskExit, TASKDIED, ntask, child);
if (info < 0) { pvm_perror("notify"); pvm_exit(); return -1; }

/* reap the middle child */
info = pvm_kill(child[ntask/2]);
if (info < 0) { pvm_perror("kill"); pvm_exit(); return -1; }

/* wait for the notification */
info = pvm_recv(-1, TASKDIED);
if (info < 0) { pvm_perror("recv"); pvm_exit(); return -1; }
info = pvm_upkint(&deadtid, 1, 1);
if (info < 0) pvm_perror("calling pvm_upkint");

/* should be the middle child */
printf("Task t%x has exited.\n", deadtid);
printf("Task t%x is middle child.\n", child[ntask/2]);
pvm_exit();
return 0;
}

/* i'm a child */
sleep(63);
pvm_exit();
return 0;

```

CHAPTER 3

DESIGN AND IMPLEMENTATION OF PVM ROUTINE

In chapter 2, we briefly introduced PVM, and explained why we use PVM. In this chapter, we will discuss how to design the parallel-distributed computation of the project and how to implement it using PVM.

3.1 Design

As we know, the integrated data analysis can be modularized into many subprocesses, and these subprocesses can be calculated independently of each other. In order to do similar subprocesses, we need to design a similar application as a rapid prototype, and implement it using PVM, as well as test if it works correctly and efficiently.

First, an ideal prototype should be selected and designed, which should have the following features:

1. It takes much longer time to complete the task if a single computer is used to do the task;
2. The task can be divided into many subtasks that can be done independent from each other, and the outcome of a subtask is not dependent on the outcomes of others;

3. The final result of the task can be defined by these subtasks' outcomes.

After considering carefully and testing repeatedly, I chose to compute the number of primes within a given range, for example, calculating the number of primes from 1 to 1000000. Generally, it consumes much time to list primes for a large range. The following code provided by Dr. Wilson in his C++ course (CS205) is used to check if a number is a prime[9]:

```
#include <iostream.h>

#include <math.h>

main() {

    int i, div, prime, count=0;

    unsigned int x = 1, num = 1;

    // computer prime numbers

    while (num <= 10000000) {

        prime = 1;

        div = 3;

        while (div <= sqrt(num)) {

            if ((num % div) == 0) {

                prime = 0;

                break;

            }

            else div += 2;

        }

        if (prime == 1) {
```

```

        cout << num << endl;

        count++;

    }

    num += 2;

}

cout << "count = " << count << endl;

return 0;

}

```

Second, a PVM algorithm should be designed to implement how to compute the number of primes for a natural number sequence. The sequence is divided into many subsequences, and all subsequences have the same length. Master process spawns slave processes on specified hosts, when one of slave processes finishes its task, which means it finds the last prime in a subsequence, it sends message and results to master process, then master process spawns a new process to this host. This procedure will be repeated until all slave processes that master spawns and sends finish their tasks. In Chapter 6, a flowchart (Figure 6.2) is shown, including load balancing and fault tolerance. Fault tolerance and load balancing will be discussed in Chapter 5 and Chapter 6, respectively.

3.2 Implementation

For a natural number sequence, how many primes are there from 1 to 10000000? The sequence 1 ~ 10,000,000 is divided into 10 subsequences, such as 1 ~ 1000000, 1000001

~ 2000000, 2000001 ~ 3000000, 3000001 ~ 4000000, 4000001 ~ 5000000, 50000001 ~ 6000000, 6000001 ~ 7000000, 7000001 ~ 8000000, 8000001 ~ 9000000, and 9000001 ~ 10000000, the total prime number is the sum of the results of the subsequences. By using PVM, 10 slave processes need to be spawned by master process. Only four hosts could be used on Scinet when I used it; they were p1, p5, p6, and Scinet. So, concurrently four slave processes worked. My prime routine at beginning produced four slave processes and sent data to p1, p5, p6, and Scinet respectively, then it used `pvm_nrecv()` to wait for information from slave processes, each slave process returned its host name, subsequence's scope, elapsed time, and the number of primes. Once the master received information from a host, it would spawn a new slave process and send new data to the host. It would repeat this procedure until 10 slave processes returned their results. Figure 2.1 displays how the master-slave model works and the functions are used. The PVM prime routine consists of `pmaster.C` and `pslave.C`. A result of running it is in Appendix A.

`pmaster.C` – Master process, spawns 10 subprocesses and sends them to slave, and receives message from slave. The following code is used in `pmaster.C`:

1) Spawn processes and send data to slaves

```
for (i = 1; i < Spawn_Number; i++) {
    // spawn task copies
    info = pvm_spawn("pslave", (char**) 0, PvmTaskHost, myhost, 1, &tid[i]);
    if(info < 0) pvm_perror("After pvm_spawn()");
    // initialize buffer, pack, and send
```

```
info = pvm_initsend(PvmDataRow);
```

```
if(info < 0) pvm_perror("After master pvm_initsend()");
```

```
number = i*NUM/Subprocess_Number; // NUM = Sequence_Number
```

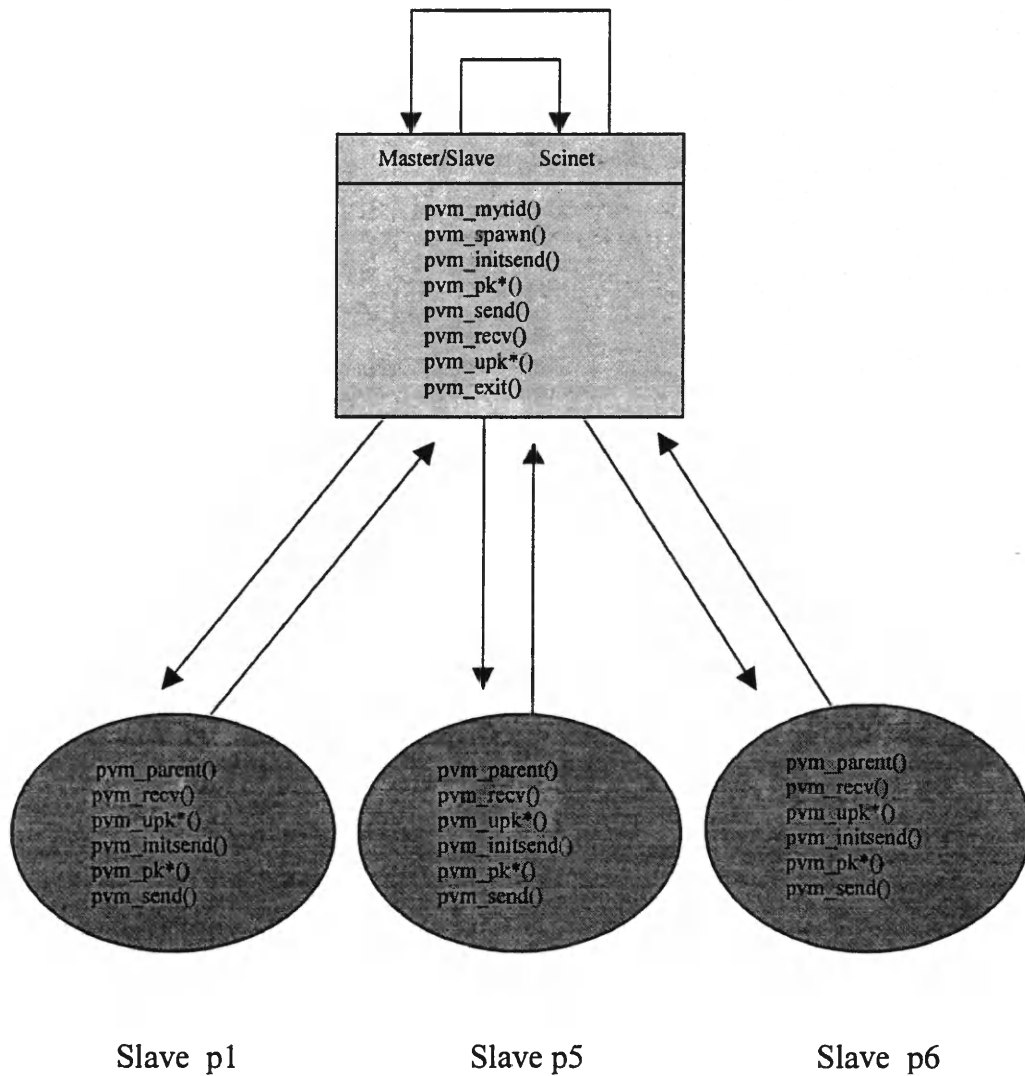


Figure 2.1 A Schematic Diagram of Master-Slave Model and their PVM Functions

```

info = pvm_pkint(&number, 1, 1);
if(info < 0) pvm_perror("After master pvm_pkint()");

```

```

info = pvm_send(tid[i], msgtag);
if(info < 0) pvm_perror("After master pvm_send()");

```

```

}

```

2) Receive results from slaves and send new tasks to slaves

// master wait for message from a slave and send a new job

```

for(i = Spawn_Number; i <= NTASKS + Spawn_Number - 1 ; i++) {

```

```

do {

```

```

    info = pvm_nrecv(-1, msgtag); // waiting and receive

```

```

    if(info < 0) pvm_perror("After master pvm_nrecv()");

```

```

}while(info == 0);

```

```

if(info > 0) {

```

```

    info = pvm_upkint(&slave_id, 1, 1); // slave_id – task id

```

```

    if (info < 0) pvm_perror("After master pvm_upkint() for slave_id");

```

```

    info = pvm_upkint(&sum, 1, 1); // sum – the number of primes

```

```

    if (info < 0) pvm_perror("After master pvm_upkint() for sum");

```

```

    info = pvm_upkint(&start, 1, 1); // start – the first number of subsequence

```

```

if (info < 0) pvm_perror("After master pvm_upkint() for start");

info = pvm_upkint(&end, 1, 1); // end – the last number of subsequence
if (info < 0) pvm_perror("After master pvm_upkint() for end");

info = pvm_upkint(&spend, 1, 1); // spend – time is used
if (info < 0) pvm_perror("After master pvm_upkint() for spend");

info = pvm_upkstr(&myhost[0]); // myhost – node is used
if (info < 0) pvm_perror("After master pvm_upkstr() for myhost");

if(i > NTASKS) continue; // receive rest tasks
else { // spawn new copy
    info = pvm_spawn("pslave", (char**) 0, PvmTaskHost, myhost, 1, &tid[I]);
    if(info < 0) pvm_perror("After pvm_spawn()");

    info = pvm_initsend(PvmDataRow);
    if(info < 0) pvm_perror("After master pvm_initsend()");

    number = i*NUM/Subprocess_Number;

    info = pvm_pkint(&number, 1, 1);

```

```

        if(info < 0) pvm_perror("After master pvm_pkint()");

        info = pvm_send(tid[i], msgtag);

        if(info < 0) pvm_perror("After master pvm_send()");

    }

} // end if

} // end for

```

pslave.C – Slave process, receives message from master, finds primes and returns results to master. Its code is as following:

```

// I'm the slave process-receive and unpack what master sent
if(myptid != PvmNoParent) {

    info = pvm_rcv(myptid, msgtag);

    if(info < 0) pvm_perror ("After slave pvm_rcv()");

    info = pvm_upkint(&list, 1, 1); // list equals to number
    if(info < 0) pvm_perror("After slave pvm_upkint()");

    // I'm the slave - calculate sum, then send to master

    t1 = time(NULL);

    sum = 0;

```

```
p = list - Sequence_Number/Subprocess_Number -1;  
list1 = list - Sequence_Number/Subprocess_Number -1;
```

```
while (p <= list) {  
    prime = 1;  
    div = 3;  
  
    while (div <= sqrt(p)) {  
        if ((p % div) == 0) {  
            prime = 0;  
            break;  
        }  
        else div += 2;  
    }  
  
    if (prime == 1) {  
        sum++;  
    }  
  
    p += 2;  
}
```

```
t2 = time(NULL) - t1;
```

```
info = pvm_initsend(PvmDataRaw);
```

```
if(info < 0) pvm_perror("After slave pvm_initsend()");
```



```

// pack mytid so master will know who message is from

info = pvm_pkint(&mytid, 1, 1);

if(info < 0) pvm_perror("After slave pvm_pkint() for mytid");

info = pvm_pkint(&sum, 1, 1);

if(info < 0) pvm_perror("After slave pvm_pkint() for sum");

info = pvm_pkint(&list1, 1, 1);

// list1 = list Sequence_Number/Subprocess_Number -1

if(info < 0) pvm_perror("After slave pvm_pkint() for list1");

info = pvm_pkint(&list, 1, 1); // list = number

if(info < 0) pvm_perror("After slave pvm_pkint() for list");

info = pvm_pkint(&t2, 1, 1); // t2 is used time

if(info < 0) pvm_perror("After slave pvm_pkint() for t2");

info = pvm_pkstr(&myhost[0]);

if(info < 0) pvm_perror("After slave pvm_pkstr() for myhost");

info = pvm_send(myptid, msgtag);

if(info < 0) pvm_perror("After slave pvm_send()");

```

```
}

```

Following PVM functions are used in the PVM routine:

`pvm_nrecv(int tid, int msgtag)` – Checks for nonblocking message with label msgtag;

The routine `pvm_nrecv` checks to see whether a message with label msgtag has arrived from tid. If tid = -1 and msgtag is defined by the user, then `pvm_nrecv` will accept a message from any process that has a matching msgtag.

`pvm_upkint(int *ip, int nitem, int stride)` – Unpacks the active message buffer into arrays of integer data type;

`pvm_upkstr(char *sp)` - Unpacks the active message buffer into arrays of char data type;

`pvm_barrier(char *group, int count)` – Blocks the calling process until all processes in a group have called it;

and `pvm_perror(char *msg)` – Prints the error status of the last PVM call.

Hosts can be chosen and added and, when tasks are done, the hosts are deleted automatically, and the routine exits PVM.

Using `pvm_nrecv()` function is a key part of the PVM routine. `pvm_nrecv()` is little different from `pvm_recv()`.

The `pvm_nrecv()` is non-blocking in the sense that it always returns immediately either with the message or with the information that the message has not arrived at the local pvmd. It can be called many times to check whether a given message has arrived yet. If the requested message has not arrived, then `pvm_nrecv()` immediately returns with 0.

`Pvm_recv()` is blocking, which means the routine waits until a message matching the user specified tid and mesgtag values arrives at the local pvmd. If the message has already arrived, then `pvm_recv()` returns immediately with the message.

The performance of both `pvm_nrecv()` and `pvm_recv()` is tested in the PVM prime routine.

CHAPTER 4

CORRECTNESS AND EFFICIENCY OF PVM ROUTINE

In chapter 3, we designed and implemented a parallel-distributed computation using PVM. In this chapter, we indicate how we tested the parallel-distributed computation's correctness and efficiency, and show results of tests and comparisons.

4.1 Tests

In order to confirm the PVM prime routine correct and efficient, we need to compare it to a non-PVM routine that does the same task, and measure the time of running the routines. Of course, first, we must guarantee the non-PVM routine works correctly. A simple way is to test the non-PVM routine with a small range, for example, 1 to 20. So, we are sure the non-PVM works very well. However, the best way is to prove its algorithm, but it is not necessary for us to do this for the project; its algorithm has been proven[9].

Firstly, a single computer, Scinet, was used to calculate the number of primes between 1 and 10000000 with the non-PVM routine, it took around 1600 seconds, and 664579 primes are found. Two non-PVM routines with different algorithms returned the number 664579. So, there are 664579 primes from 1 to 10000000.

Next, we tested the PVM prime routine with two hosts, three hosts, and four hosts respectively. For PVM prime routine with 4 hosts, it took about 564 seconds and got the same result – there are 664579 primes between 1 and 10000000. Also `pvm_nrecv()` was compared with `pvm_recv()`.

4.2 Results and Comparisons

The outcomes of testing with `pvm_nrecv()` are listed as following:

Host number	Host name	Time used	Prime Number
1	Scinet	1564 seconds	664579
2	Scinet, p5	1127 seconds	664579
3	Scinet, p5, p6	707 seconds	664579
4	Scinet, p1, p5, p6	564 seconds	664579

The outcomes of testing with `pvm_recv()` are listed as following:

Host number	Host name	Time used	Prime Number
1	Scinet	1564 seconds	664579
2	Scinet, p5	787 seconds	664579
3	Scinet, p5, p6	577 seconds	664579
4	Scinet, p1, p5, p6	465 seconds	664579

Figure 3.1 shows the relation between elapsed time and the number of processors for the PVM routine. Obviously, the PVM prime routine works well, and is much faster than

the no PVM routine. The performance of the PVM routine is fine; if 4 processors are used, it saves more than 1000 seconds relative to single-processor execution – about 18

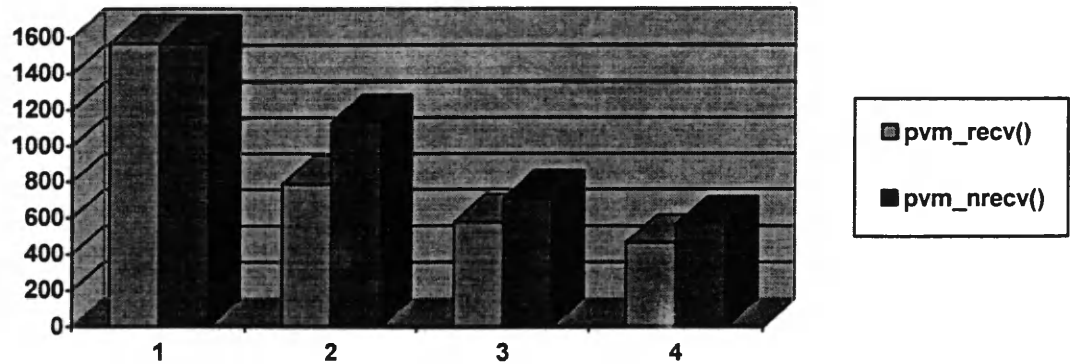


Figure 3.1 A diagram of relation between elapsed time and the number of processors

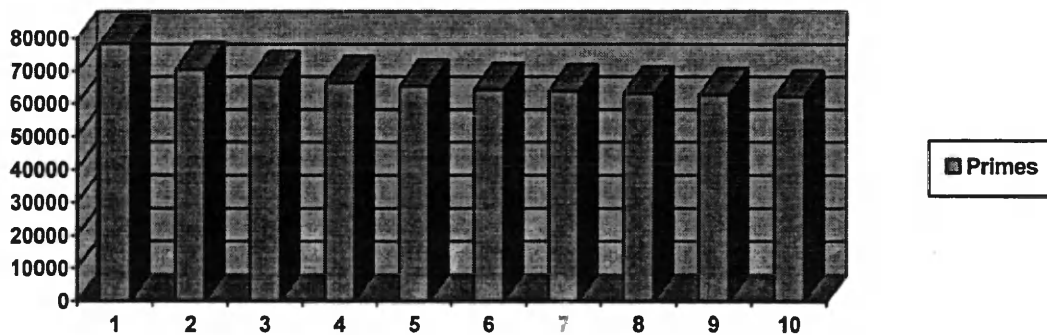


Figure 3.2 The number of primes in the ten subsequences

minutes, which means it saves 70.3% time. The PVM routine with `pvm_nrecv()` is 2.77 times faster than the non PVM routine, with `pvm_recv()` is 3.36 times.

The number of primes in subsequences is also listed as following:

The number of primes	The range of subsequence	The number of subsequence
78498 primes	1 - 1000000	1
70435 primes	1000001 - 2000000	2
67883 primes	2000001 - 3000000	3
66330 primes	3000001 - 4000000	4
65367 primes	4000001 - 5000000	5
64336 primes	5000001 - 6000000	6
63799 primes	6000001 - 7000000	7
63129 primes	7000001 - 8000000	8
62712 primes	8000001 - 9000000	9
62090 primes	9000001 - 10000000	10

Note 1: The performance of each node on Scinet system is not high. When Sun Solaris in CS Department was used to check the prime code, it only took about 279 seconds to list 664579 primes.

Note 2: With the increment of the number of subsequence, the number of primes decreases (see Figure 3.2). However, it had been proved that there are infinite primes in a

natural number sequence. The largest known prime is $2^{6972593} - 1$. It has 2098960 digits[11].

The `pvm_recv()` and `pvm_nrecv()` work very similar, but the performance of `pvm_recv()` is better than that of `pvm_nrecv()`. One reason may be that `pvm_nrecv()` is non-blocking, it may be called a lot of times in a loop to check if a given message has arrived; the other reason may be that their algorithms are different. Anyway, we will use `pvm_recv()` instead of `pvm_nrecv()` in the project.

CHAPTER 5

FAULT TOLERANCE

In the previous chapters, we briefly introduced PVM, designed and implemented the PVM routine, and verified its correctness and efficiency. In this chapter and the next chapter, we will show how to improve its performance. Fault tolerance is an important issue of Parallel Virtual Machine [5]. Why we need to consider fault tolerance, and how to implement it are discussed in this chapter in detail.

5.1 Why Fault Tolerance

Generally, fault tolerance is an extensive issue; we have to deal with fault tolerance in many applications. We need program and network to be robust. For PVM master-slave model, fault tolerance is very important. We have to consider fault tolerance.

Since master sends tasks to slaves and receives the results from slaves, if a slave task crashes for some reason, master could not receive its result, so, the entire task could not be done. To make the program fault-tolerant, the master has to monitor the tasks that exited/failed without sending the result back. The master creates some new tasks to do the work of those tasks.

5.2 How to Implement

PVM is able to withstand host and network failures. It doesn't automatically recover an application after a crash, but it does provide polling and notification primitives to allow fault-tolerant applications to be built. The virtual machine is dynamically reconfigurable. PVM provides `pvm_notify()` routine to notify the caller on detecting certain events. Here is a brief description of `pvm_notify()`.

pvm_notify - Request notification of PVM event such as host failure.

C int info = pvm_notify(int what, int msgtag, int cnt, int *tids)

what -- Type of event to trigger the notification. Presently one of:

<i>Value</i>	<i>Meaning</i>
<i>PvmTaskExit</i>	<i>Task exits or is killed</i>
<i>PvmHostDelete</i>	<i>Host is deleted or crashes</i>
<i>PvmHostAdd</i>	<i>New host is added</i>

msgtag -- Message tag to be used in notification.

cnt -- For *PvmTaskExit* and *PvmHostDelete*, specifies the length of the *tids*

array. For *PvmHostAdd*, specifies the number of times to notify.

tids -- For *PvmTaskExit* and *PvmHostDelete*, an array of length *cnt* of task or

pvm TIDs to be notified about. The array is not used with the

PvmHostAdd option.

info -- Integer status code returned by the routine. Values less than zero

indicate an error.

Since we are interested in finding out when a task fails, we call `pvm_notify()` after spawning the tasks. The `pvm_notify()` call tells PVM to send the calling task a message when certain tasks exit/fail.

Example Code:

```
for (i= 0; i < Spawn_Number; i++) {

    // spawn process on a fixed host

    info = pvm_spawn("pslave", (char**) 0, PvmTaskHost, hostname[i], 1, &tid[i+1]);

    if(info < 0) pvm_perror("After pvm_spawn()");

    // ask for notification when a task exits

    if(info == 1) status = pvm_notify(PvmTaskExit, TASKEXIT, 1, &tid[i+1]);

    if(status < 0) {pvm_perror("notify"); pvm_exit(); }

    // initialize buffer, pack, and send

    info = pvm_initsend(PvmDataRaw);

    if(info < 0) pvm_perror("After master pvm_initsend()");
```

```

    number = (i+1)*NUM/Subprocess_Number;

    info = pvm_pkint(&number, 1, 1);

    if(info < 0) pvm_perror("After master pvm_pkint()");

    info = pvm_send(tid[i+1], msgtag);

    if(info < 0) pvm_perror("After master pvm_send() 1111");

}

```

Normally we could encounter two situations: one is that a task fails/exits, the other is that a host crashes or it is deleted from a network. If a task exited before sending back the message, we need to create another task to do the same job. If a host has been deleted/suspended from a network, we need to check to see if the tasks running on it has been finished. If not, we should create new slave tasks to do the work on some other hosts.

5.3 Tests

In order to confirm the fault-tolerance of my program, the `pvm_kill()` was used to kill a process and `pvm_delhosts()` to delete a host during the period when the program was executing. To delete a host using `pvm_delhosts()` is similar to remove a node from a network by hand. Of course, the program was also tested by unplugging hosts' Ethernet cards.

Here is an example of code:

```

// Spawn process on a fixed host

info = pvm_spawn("slave", (char**)0, PvmTaskHost, hostname[i], 1, &tid[i+1]);

    if(info < 0) pvm_perror("After pvm_spawn()");

// Ask for notification when a task exits

if(info == 1) status = pvm_notify(PvmTaskExit, TASKEXIT, 1, &tid[i+1]);

    if(status < 0) pvm_perror("notify");

// Ask for notification when a host is deleted

info = pvm_notify(PvmHostDelete, HOSTDELETE, 1, &tid[i+1]);

    if(info < 0) pvm_perror("PvmHostDelete");

// Delete a host from PVM

info = pvm_delhosts(&hostname[i+1], 1, &infos[i+1]);

    if(info != 1) pvm_perror("After pvm_delhosts()");

// Kill a task

info = pvm_kill(tid[i+2]);

    if(info < 0) pvm_perror("kill tid[i+2]");

```

After killing tasks and deleting a host, the `pvm_notify()` routine detected unlucky events and notified master about the events, so the master would continue to send the

failed tasks to slaves again depending on the task IDs. We also test fault tolerance with abort() function. The result of running the program displayed that it has the ability to handle potential faults. Here is example:

.....

/ 4 tasks start up and distributed across processors */*

Tasks 1048578, 262149, 786434, 1310722 were sent.

Unlucky scapegoat was deleted...

Again unlucky Task 262149 is killed...

Task 262149 failed and send it again! / Since task is gone, master creates a new task */*

/ Looking for a host with low load */*

load average: 0 larch / larch, and reimel ... are machines' names in CS Department*/*

load average: 1.11 reimel

load average: 0 garnet

load average: 0.11 stillwater

load average: 8.63 junio

select load average: 0 larch to send the task 262149 again ...

Task 786434 failed and send it again! / Since task is gone, master creates a new task */*

/ Looking for a host with low load */*

load average: 0 larch

load average: 1.11 reimel

load average: 0.02 garnet

load average: 0.12 stillwater

load average: 8.66 jun0

select load average: 0 larch to send the task 786434 again ...

ninepipe is added again!!! / Since host is deleted, master adds a new host */*

My master: Received 78498 primes between 1 and 1000000 from task 1048578,

it took 9 second. Its host is garnet.

/ Looking for a host with low load */*

load average: 0.23 larch

load average: 1.09 reibel

load average: 0.14 garnet

load average: 0.25 stillwater

load average: 8.72 jun0

load average: 0.5 ninepipe

.....

CHAPTER 6

LOAD BALANCING

In the last chapter, we discussed fault tolerance. In this chapter, we discuss another important issue of PVM – Load Balancing [5], why we have to do load balancing, and how to implement load balancing.

6.1 What is Load Balancing

To extract the maximum performance from the parallel applications, load balancing is very important. Making sure that each host is doing its fair share of work and that all processors are busy doing useful work most the time. There are two kinds of load balancing – *static* load balancing and *dynamic* load balancing.

The simplest method is *static* load balancing. In this method, the problem is divided up and tasks assigned to processors only once. The partitioning may occur before the job starts, or as an early step in the application. The size and number of tasks can be varied depending on the processing power of a given machine. On a lightly loaded network, this scheme can be quite effective.

When computational loads vary, a more sophisticated *dynamic* method of load balancing is required. It is important to keep all nodes busy all the time. This is typically

implemented as a master/slave program where the master manages a set of tasks. It sends jobs to slaves as slaves become idle.

6.2 How to Implement

My program works with both static load balancing and dynamic balancing. Before spawning task processes to slaves, the master does static load balancing, sorts hosts depending on the load average on the hosts, and selects the hosts with lower load to send tasks.

During execution, either the master receives the result of a task or the master receives message of a task fail, then the master does dynamic load balancing, selecting the host with the lowest load on the virtual machines to send next task.

To obtain load information on the parallel virtual machines, the master spawns slave processes to all hosts, each slave process uses system call – `system()` to run `uptime` command for its host and generate a file including load information, then reads the file, extracts the load information, and sends it back to master. Dr. Borries Demeler put forward a different way [10] to obtain load information using `popen()` without generating a file and reading it.

Here is example code:

```

FILE *str;

char *test[80];

int i = 0;

str = popen("/usr/bin/uptime", "r");
while((fscanf(str, "%s", test)) > 0) {
    printf("%d: %s\n", i, test);
    i++;
}

```

The uptime command prints the current time, the length of time the system has been up, and the average number of jobs in the run queue over the last 1, 5 and 15 minutes. In the program, the load average in the last 1 minute is used.

After getting uptime information, the strtok() function [7] is used to extract load average message.

Here is example code:

```

char *a1, *a2, *a3, buf, bufout[100], *tok;

FILE *out;

int i = 0, start;

while(!feof(out)) {

    // get all string from a file

```

```
fgets(bufout, 100, out);

// strtok breaks string into "token" by separated spaces
tok = strtok(bufout, " ");

while(tok !=NULL) {

    // compare token with string "average:"
    if(strcmp(tok, "average:") == 0)

    // get the position for load average information
        start = i;

    if(i== start + 1)
        a1 = tok;

    if(i== start + 2)
        a2 = tok;

    if(i== start + 3)
        a3 = tok;

    tok = strtok(NULL, " ");

    i++;
}
}
```

Figure 6.1 indicates the steps of load balancing. Step 1: The master spawns processes to the slaves for obtaining load average for each node. Step 2: The master sorts nodes depending on load average obtained in each node. Step 3: The master spawns processes and sends data to the slaves (nodes) that have much lower load average, and receives the information from the slaves. Step 4: Once the master receives message, either a task is finished or a task crash, it performs Step1 and selects the lowest load node. Step 5 The master sends either next task or failed task to the lowest load nodes. Step4 and Step 5 are repeated until all tasks are done. Figure 6.2 is a brief flow chart of load balancing and program. The `init()` is used to initialize program, read host file, and add hosts; `find_balance()` is used to obtain load average for each host; `collect_info()` collects load information; `sort()` is used to sort load average and its hostname; `spawnjob()` spawns a task copy; and `minload()` finds the minimum load average.

The result of executing program showed both static load balancing and dynamic load balancing work very well. Here is example:

.....

The hosts in the virtual machine:

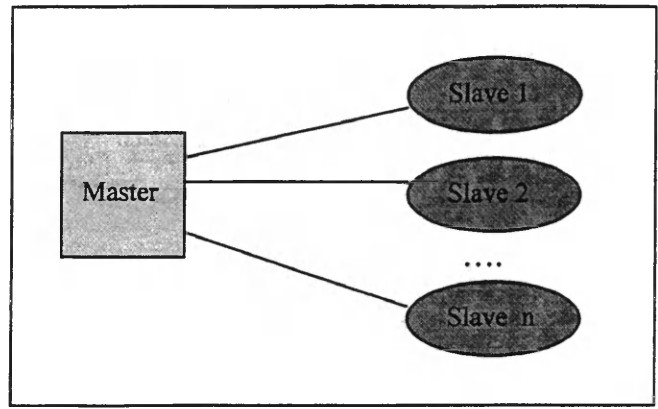
larch, stillwater, juno, scapegoat, garnet, bannack,

load average: 0.5 stillwater

load average: 0.55 larch

load average: 0.68 juno

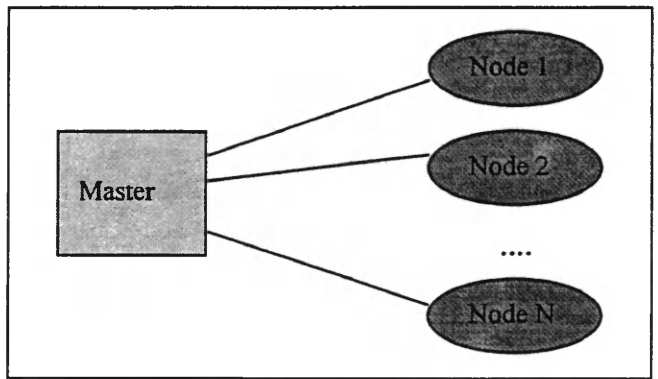
load average: 0.59 scapegoat



Step 1

Master
Sorts
Nodes

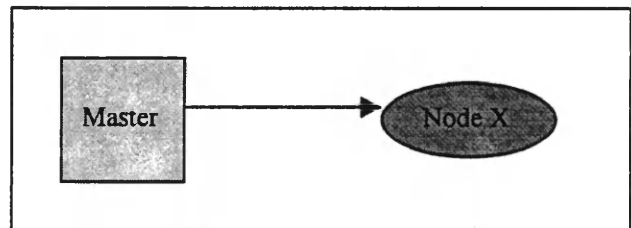
Step 2



Step 3

Master
Selects
Node

Step 4



Step 5

Figure 6.1 Steps of Load Balancing

load average: 1.09 garnet

load average: 0.87 bannack

After sorting:

Stillwater, larch, scapegoat, juno, bannack, garnet

Tasks 524290, 262149, 1048578, 786434 were sent.

Unlucky scapegoat was deleted...

Again unlucky Task 262149 is killed...

Task 262149 failed and send it again!

load average: 0.57 larch

load average: 0.89 garnet

load average: 0.89 bannack

load average: 0.73 stillwater

load average: 0.92 juno

select load average: 0.57 larch to send the task 262149 again ...

Task 1048578 failed and send it again!

load average: 0.83 stillwater

load average: 0.98 garnet

load average: 0.83 bannack

load average: 0.81 larch

load average: 1.13 juno

.....

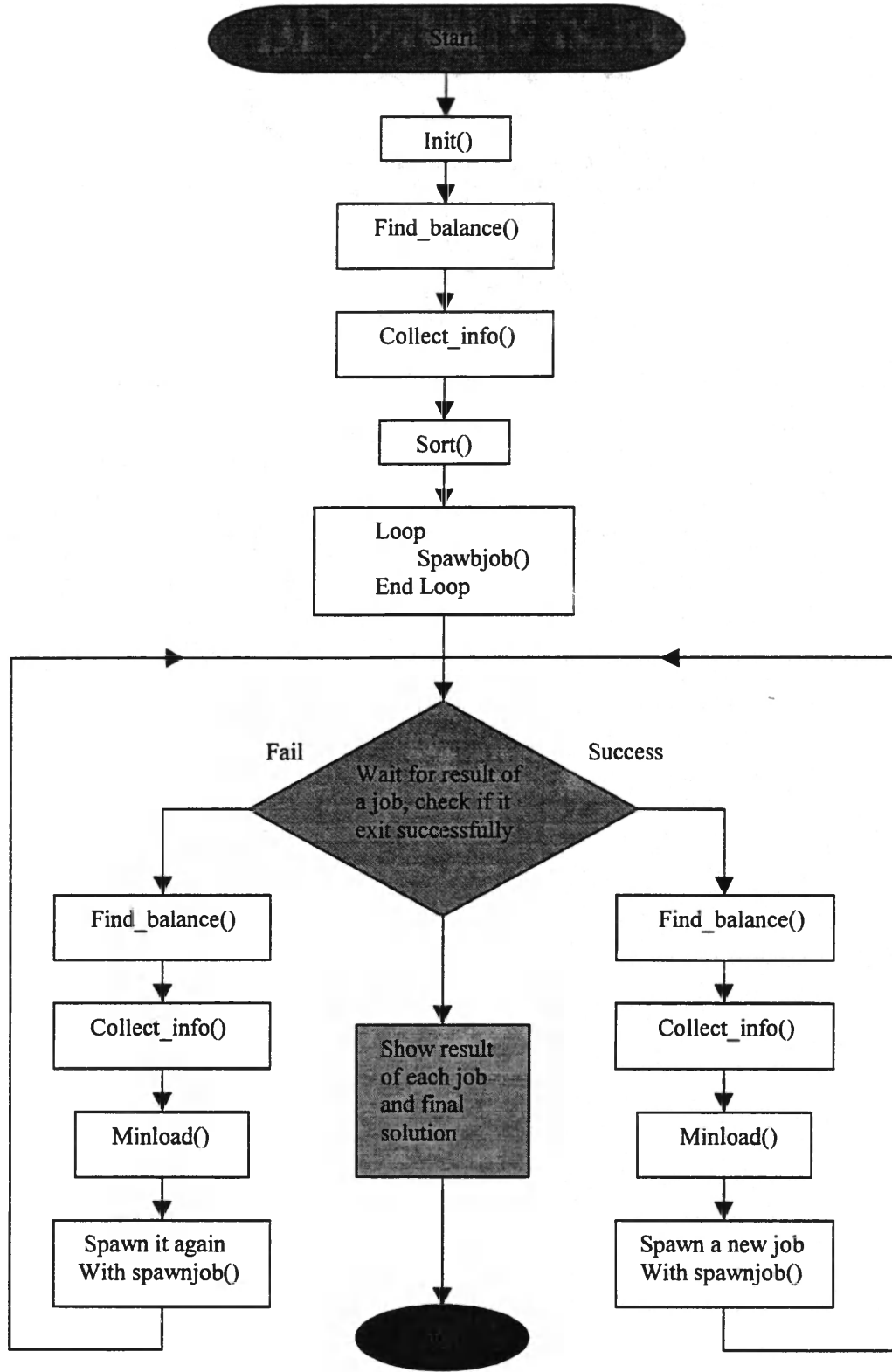


Figure 6.2 Flowchart of load balancing and program

During the period when I implemented load balancing, I encountered a strange problem on Solaris, the process that subprocess performs uptime command, reads load information, and sends back to master executes too fast to return result to master. Master always finds tasks exiting before they return results. In order to solve the problem, `pvm_recv()` is used in the subprocess to wait until master receives information, once master receives load information, it will send acknowledgement. In this way, subprocess exits when it knows master has received its load information.

6.3 Discuss

In fact, to do load balancing is to increase load for each host, particularly for dynamic load balancing, you have to ask master to send subprocesses to slaves (nodes) for obtaining load information. Therefore, it is not surprising to find that the performance with dynamic load balancing is not higher than that without dynamic load balancing on a lightly loaded network. The program without load balancing makes master send next task immediately to the host which just completes its task. This procedure also performs an acceptable load balancing on a lightly loaded homogenous network. Thus, dynamic load balancing scheme is desirable for a heterogeneous computing environment, because all nodes in the network do not have identical computation capacities.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

In this project, in order to show the basic idea of a parallel-distributed computation of master-slave model using PVM for the integrated biophysical data analysis, we designed and implemented a parallel-distributed computation to find primes for a natural sequence using PVM on a set of specified hosts, and confirmed that the PVM routine was both correct and efficient. The results of tests indicated that the PVM routine is 3.36 times faster when running on four processors than the non-PVM routine.

To make the PVM routine robust, we implemented fault tolerance; the program has the ability to monitor all tasks, if it finds a task fails without sending its result to master, the task will be spawned by master again. In order to extract the maximum performance, we implemented static load balancing and dynamic load balancing, making sure all processors are busy doing useful work most the time.

Scheduling is an important part of load balancing. If the number of nodes on a heterogeneous network is a very large number, we need a better sorting and selecting algorithm, such as quick sort and merge sort, otherwise, if the number of nodes is relatively small number, the speed of a common sorting and selecting algorithm is fast enough to do scheduling.

As a result of the project, we believe that PVM can be selected to implement similar parallel-distributed computation. Dr. Demeler's integrated analysis routine would be more efficient if a similar parallel-distributed computing were done.

Two other things that may be considered to do in the future work are that: 1) If the master pvmd dies then the entire virtual machine shuts down. Is there a way that is able to find such an event and recover from the event? However, PVM fault detection and recovery is built on the assumption that master must never crash. 2) If a slave host is temporarily not connected to the network for some reason, and after a while the host is connected to the network again, then how the task on the host is handled gracefully.

For 2), however, it is a problem that depends on how long the slave host is not connected to the network. PVM fault detection originates in the pvmd-pvmd protocol. When the pvmd times out while communicating with another, it calls `hostfailentry()`, which scans waitlist and terminates any operations waiting on the down host. PVM daemons communicate with each other via UDP and the PVM daemon on a machine communicates with tasks on the same machine via TCP or via UNIX domain sockets. The PVM daemons estimate the round trip time to the other daemons. It initially resends packets after 3 times of the estimated round trip time has elapsed without an acknowledgement being received. It doubles the retry wait for each additional retry, up to 18 seconds. The round trip time estimate itself is limited to 9 seconds. It will retry at least 9 times before giving up and if it doesn't receive an acknowledgement after 3 minutes it considers the other daemon to be unreachable and calls `hostfailentry()` [4]. Therefore,

after 3 minutes, we have to face to the problem how to recover the daemon on a cluster if we still need the task on the daemon. In fact, it is not a big issue for my program in the project, because if the master knows a host dies, it will check whether the task on the host is finished or not. If not, it will send the task to other host again, and before sending the task it reconfigures and selects a lowest load host.

References

1. Ralston and Jennrich 1978, Dud, A Derivative-Free Algorithm for Nonlinear Least Squares, Technometrics, Vol.20, No.1
2. Michael T. Heath, 1997, Scientific Computing - An Introductory Survey, McGraw-Hill
3. Dr. Demeler' proposal and his code 1999
4. Al Geist et., 1994, PVM: Parallel Virtual Machine, A Users' Guide and Tutorial for Networked Parallel Computing
5. S. Petri and H. langendorfer, 1995, Laod Balancing and Fault Tolerance in Workstation Clusters Migrating Groups of communicating Processes, Operating Systems Review, Vol. 29, No.4
6. Larry L. Peterson and Brice S. Davie, 2000, Computer Networks, A Systems Approach, Morgan Kaufmann
7. H.M.Deitel and P.J.Deitel, 1994, How C++ to Programming, Prentice Hall

8. G.A.Geist, J. A. Kohl, P.M. Papadopoulos, 1996, PVM and MPI: A Comparison of Features, *Calculateurs Paralleles*, 1996, Vol. 8, No. 2
9. Wilson, Ronald. Spring 1998, Presented in C++ course (CS205)
10. Demeler, Bories. Summer 2000, Personal correspondence
11. Hajratwala, Nayan et.al., 1999, <http://www.utm.edu/research/primes/largest.html>

Appendix

The result of running PVM prime routine using four hosts with `pvm_rcv()`.

```
scinet-inside:~$ ptest4
mytid: 262146
myptid: -23
myhost: scinet-inside.scinet.prairie.edu
Please enter host1's name:
p5
p5 is added ..
Please enter host2's name:
p6
p6 is added ..
Please enter host3's name:
p1.scinet.prairie.edu
p1.scinet.prairie.edu is added ..
```

My master: Received 78498 primes between 1 and 1000000 from task 524289, it took 57 second. Its host is p6

My master: Received 70435 primes between 1000001 and 2000000 from task 262147, it took 95 second. Its host is scinet-inside.scinet.prairie.edu

My master: Received 67883 primes between 2000001 and 3000000 from task 786433, it took 118 second. Its host is p5

My master: Received 66330 primes between 3000001 and 4000000 from task 1048577, it took 164 second. Its host is p1.scinet.prairie.edu

My master: Received 65367 primes between 4000001 and 5000000 from task 524290, it took 152 second. Its host is p6

My master: Received 64336 primes between 5000001 and 6000000 from task 262148, it took 164 second. Its host is scinet-inside.scinet.prairie.edu

My master: Received 63799 primes between 6000001 and 7000000 from task 786434, it took 177 second. Its host is p5

My master: Received 63129 primes between 7000001 and 8000000 from task 1048578, it took 227 second. Its host is p1.scinet.prairie.edu

My master: Received 62712 primes between 8000001 and 9000000 from task 524291, it took 197 second. Its host is p6

My master: Received 62090 primes between 9000001 and 10000000 from task 262149, it took 206 second. Its host is scinet-inside.scinet.prairie.edu

Time elapsed = 465 seconds

Total primes = 664579

P5, p6, and p1.scinet.prairie.edu was deleted ..

mytid: 262146 exiting PVM ...