P-1165

PARALLEL VIRTUAL MACHINE

PROJECT REPORT

Submitted in partial fulfillment of the
requirement for the award of the degree of
Bachelor of Engineering in Information Technology
of Bharathiar University, Coimbatore.

Submitted by

N.Bhuvaneswari **0027SO708**

J.Nithya Lakshmi **0027SO091**

Under the guidance of

Ms. S. Rajini, B.E.,
Lecturer, CSE department.

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
KUMARAGURU COLLEGE OF TECHNOLOGY,
COIMBATORE – 641006.

MARCH - 2004.

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

KUMARAGURU COLLEGE OF TECHNOLOGY

(Affiliated to Bharathiar University, Coimbatore)

CERTIFICATE

This is to certify that the project entitled

PARALLEL VIRTUAL MACHINE

is done by

N.Bhuvaneswari — **0027SO708**

J.Nithya Lakshmi — **0027SO091**

and submitted in partial fulfillment of the requirement for the award of the degree of **Bachelor of Engineering in Information Technology of Bharathiar University, Coimbatore, during the academic year 2003-2004.**

Professor & Head of the department
(Dr.S.THANGASAMY)

Guide
(Ms. S. RAJINI)

Certified that the candidates were examined by us in the project work and viva -voce examination held on _____.

Internal Examiner

External Examiner

DECLARATION

We,

N.Bhuvaneswari and **J.Nithya Lakshmi**, hereby declare that the project entitled **"Parallel Virtual Machine",** submitted to Kumaraguru College of Technology, Coimbatore (affiliated to Bharathiar University) is a record of original work done by us under the supervision and guidance of **Ms. S.Rajini,** B.E., Lecturer, Department of Computer Science and Engineering.

NAME	REGISTRATION NUMBER	SIGNATURE
N.Bhuvaneswari	0027S0708	
J.Nithya Lakshmi	0027S0091	

Countersigned by Staff- in- charge,

Ms. S.Rajini B.E.

Lecturer,

Department of Computer Science

and Engineering,

Kumaraguru College of Technology.

Place : Coimbatore

Date :

ACKNOWLEDGEMENT

The exhilaration achieved as a result of the successful completion of any task should be definitely shared with the people behind the venture.

At the onset, we take this opportunity to thank the management of our college for having provided us excellent facilities to work with. We express our deep gratitude to our Principal **Dr.K.K.Padmanabhan, B.Sc(Engg), M.Tech., Ph.D.**, for ushering us in the path of triumph.

We are also thankful to our beloved Professor and the Head of the Department, **Dr.S.Thangasamy, B.E.(Hons), Ph.D**, whose consistent support and enthusiastic involvement helped us a great deal.

We are greatly indebted to our beloved guide **Ms.S,Rajini B.E.**, Lecturer. Department of Computer Science and Engineering, for her excellent guidance and timely support during the course of this project.

We also thank our project coordinator, **Mrs.S.Devaki, M.S..**, Assistant Professor, Department of Computer Science and Engineering and our beloved class advisor **Ms.P.Sudha, B.E.**, Lecturer, Department of Computer Science and Engineering, for their invaluable assistance.

Our thanks are due to the lab administrators and all other staff members of Department of Information Technology, Kumaraguru College of Technology, Coimbatore.

We feel proud to pay our respectful thanks to our parents for their enthusiasm and encouragement and we also thank our friends who have associated themselves to bring out this project successfully.

SYNOPSIS

This project is the implementation of mandatory functionalities of a parallel virtual machine, a parallel programming paradigm. The core idea of the project is the process of providing a transparent parallel computing environment enabling the distribution of various tasks of an application to physically different systems connected via a network. The PVM computing model is based on the notion that an application consists of several tasks. Each task is responsible for a part of the application's computational workload. Sometimes an application is parallelized along its functions; that is, each task performs a different function. Sometimes it is parallelized along the data it acts on. PVM provides a set of modules for process management, initiation of remote task execution, maintenance of process registry and inter-process communication across system boundaries. Both data and function parallelism are achieved. Our project implements prime number generation up to ten digits in a time optimized way.

INDEX

INTRODUCTION

1.1 Existing System and Limitations

The limits of the processing power of single processor machines have already been reached. Parallel processing provides a very good solution for this. Our test case for PVM is prime number generation up to ten digits. Implementing the prime number generation as a standalone program consumes more time. Implementing the same on PVM, resulted in very good time optimization.

1.2 Proposed System and Advantages

Parallel **V**irtual **M**achine is a public domain software package. It is used to create and execute concurrent or parallel applications. PVM operates on a collection of heterogeneous Unix computers connected by one or more networks, which is called virtual *machine*. Features supplied by PVM include:

- Resource management
- Add/delete hosts from a virtual machine
- Process Control
- Spawn/ kill tasks dynamically
- Message Passing
- Blocking send, blocking and non blocking receive, collective
- Dynamic task groups
- Task can join or leave a group at any time
- Fault Tolerance
- Virtual machine automatically detects faults and adjusts

Parallel Programming: An Overview

People are always trying to make programs run faster. One way to do this is to divide the program into pieces that can be worked on at the same time (on different processors). More formally, creating a parallel program depends on finding independent computations that can be executed simultaneously. Producing a parallel program generally involves three steps:

1) Developing and debugging a sequential program. For existing programs, this step is already done.
2) Transforming the sequential program into a parallel program.
3) Optimizing the parallel program.

Of course, if the second step is done perfectly then the third one will be necessary, but in practice that rarely happens. It's usually much easier – and quicker in the long run- to transform the serial program into a parallel program in the most straight forward way possible,, measure its performance, and then search for ways to improve it. If we focus on step 2, a natural question arises: where does the parallelism come from? The language of parallel programming can be quite ambiguous. On the one hand, there is talk of "Parallelizing programs", a phrase which focuses on the programmers who convert sequential programs into parallel ones. On the other hand, much is also said about "Exploiting parallelism", implying that parallelism in already inherent in the program itself. Which is correct? It is something you find or something you create?

The answer is, of course, it depends. Sometimes a program can be trivially adapted to run in parallel because the work it does naturally divides into discrete chunks. Sometimes a serial program must be restructured significantly in order to transform into a parallel program, reorganizing the computations the program does into units that can be run in parallel. Sometimes an existing program will yield little in the way of independent computations. In this case, it is necessary to rethink the approach the problem that the program addresses, create new algorithms in order to formulate a solution that can be implemented as a parallel program.

To put it another way, a computer program doesn't merely solve a particular problem, but rather embodies a particular approach to solving its specific problem. Making a parallel version can potentially involve changes to the program structure, or the algorithms it implements, or both. The examples in this manual include instances of all three possibilities.

Once the work has been divided into pieces, yielding a parallel program, another factor comes into play: the inherent cost associated with parallelism, specifically the additional effort of constructing and coordinating the separate program parts. This overhead is often dominated by the communication between discrete program processes, and it naturally increases with the number of chunks the program is divided into, eventually reaching a point of diminishing returns where the cost of creating and maintaining the separate execution threads overshadows the performance gains realized from their parallel execution. An efficient parallel program will maximize the ratio of work proceeding in parallel to the overhead associated with its parallel execution.

This ratio of computation to communication is referred to as granularity. Think of dividing a rock into roughly equal-sized parts. There are lots of ways to do it, in fact , there is a continuum of possibilities ranging from fine grains of sand at one ends to two rocks half size of original at the other. A parallel program that divides the work intc many tiny tasks is said to be fine-grained, while one that divides it into a small number of relatively large ones can be called coarse-grained. There is no absolute correct level of granularity. Neither coarse nor fine-grained parallelism is inherently better or worse than the other. However, when overhead overwhelms computation, a program is too fine grained for its environment, whatever its absolute granularity level may be. The optimum level depends on both the algorithms a program implements and the hardware environment it runs in; a level of granularity that runs efficiently in one environment (for example, a parallel computer with very fast inter-process communication channels) may not perform as well in another (such as a network of workstations with much slower communication between processors).

There are two ways to address this issue (and we'll look at examples of both of them in the course of this manual). First, many problems offer a choice of granularity level. Which approach is correct depends on the structure of the overall problem, and each is better than the other in some circumstances. The other solution is to build adjustable granularity into programs so that they can easily be modified for different environments. Changing the granularity level then becomes as simple as changing a few parameter definitions. This technique complements the preceding one, and both can be used in the same program. These are the major issues facing a parallel programmer.

Approaches to Parallel Programming

There are two main challenges facing any parallel programmer:

1. How to do the work among the available processors.
2. Where to store the data and how to get it to processor that need it.

Two radically different approaches to these problems have emerged as the dominant parallel processing paradigms: message passing and distributed data structures. Distributed data structure programs use a shared data space, with he individual processes reading data from it and placing results into it. The data structure is distributed in the sense that it can reside in different processors, but it looks in global memory space to the component processes. This approach is highly suitable in areas where single function has to perform on a large amount of data (Multiple Data Single Instruction).

Our project concentrates on the earlier approach where functional parallelism is of main interest. It provides tools for multi-task application that could be run parallel.

SYSTEM REQUIREMENTS AND ANALYSIS

System study is an activity that encompasses most of the tasks that we have collectively called computer system engineering. System study is conducted with the following objectives.

- Identify the needs.
- Evaluate the system concept for feasibility.
- Perform economic and technical analysis.
- Allocate function to hardware, software, people and other system elements.
- Create a system definition that forms a foundation for all subsequent engineering work.

2.1 Product definition

Briefly, the principles upon which PVM is based include the following:

❖ **User- configured host pool:** The application's computational tasks execute on a set of machines that are selected by the user for a given run of the PVM program. Both single CPU machines and hardware multiprocessors (including shared-memory and distributed-memory computers) may be part of the host pool. The host pool may be altered by deleting machines during operation (an important feature for fault tolerance).

❖ **Translucent access to hardware**: Application programs either may view the hardware environment as an attribute less collection of virtual processing elements or may choose to exploit the capabilities of specific machines in the host pool by positioning certain computational tasks on the most appropriate computers.

❖ **Process- based computation**: The unit of parallelism in PVM is a task (often but not always a UNIX process), an independent sequential thread of control that alternates between communication and computation. No process-to-processor mapping is implied or enforced by PVM; in particular, multiple tasks may execute on a single processor.

❖ **Explicit message** – passing model: Collections of computational tasks, each performing a part of an application's workload using data-, functional-, or hybrid decomposition , cooperate by explicitly sending and receiving messages to one another. Message size is limited only by the amount of available memory.

❖ **Heterogeneity support**: The PVM system supports heterogeneity in terms of machines, networks and applications. With regard to message passing, PVM permits messages containing more than one data type to be exchanged between machines having different data representations.

❖ **Multiprocessor support**: PVM uses the native message-passing facilities on multiprocessors to take advantage of the underlying hardware. Vendors often supply their own optimized PVM for their systems, which can still communicate with the public PVM version.

2.2 Project Plan

In the analysis phase, the connectivity decisions regarding the creation of virtual environment was taken.

In the design phase, the PVM virtual machine comprising of heterogeneous computers was formed.

During the implementation phase, coding for prime number generation is done as a test case.

Then testing is done to prove that parallel programming using PVM provides good time optimization.

SOFTWARE REQUIREMENTS SPECIFICATION

3.1 INTRODUCTION

3.1.1 SCOPE

Our project illustrates parallel processing on heterogeneous systems. We have taken prime number generation up to ten digits as test case.

3.1.2 GLOSSARY

daemon

A special-purpose process that runs on behalf of the system, for example, the pvmd process or group server task.

host

A computer, especially a self-complete one on a network with others. Also, the front-end support machine , for example, a multiprocessor.

message tag

An integer code (chosen by the programmer) bound to a message as it is sent. Messages can be accepted by tag value and/or source address at the destination.

pvmd

PVM daemon, a process that serves as a message router and virtual machine coordinator. One PVM daemon runs on each host of a virtual machine.

spawn

To create a new process or PVM task, possibly different from the parent.

task

The smallest component of a program addressable in PVM. A task is generally a native "process" to the machine on which it runs.

virtual machine

A multi-computer composed of separate (possibly self-complete) machines and a *software backplane* to coordinate operation.

3.1.3 REFERENCES

Web Sites:

www.csm.ornl.gov/pvm

www.netlib.org/pvm3

www.comp.parallel.pvm

www.mersenne.org/prime.html

3.2 GENERAL DESCRIPTION

3.2.1 PRODUCT PERSPECTIVE

It is an easy means of parallel processing using the PVM software. The software if installed on one machine is available for other machines in the network to work with . It can also be used as an educational tool for parallel processing in institutions.

3.2.2 PRODUCT FUNCTIONS

Parallel processing is accomplished through a series of routines. Routines to establish a virtual machine, packing and unpacking messages, message communication, spawning processes and much more are available.

3.2.3 GENERAL CONSTRAINTS

In case of finding prime numbers the speed is significant and greater in case prime numbers up to ten digits and beyond that speed is reduced.

3.3 SPECIFIC REQUIREMENTS

3.3.1 FUNCTIONAL REQUIREMENTS

3.3.1.1 INTRODUCTION

The main aim is to generate prime numbers within a given range. We take into consideration prime numbers ranging up to ten digits.

3.3.1.2 LIST OF INPUTS

The upper and lower limits of the range of numbers to be tested.
The machines that constitute the virtual machine

3.3.1.3 INFORMATION PROCESSING REQUIREMENTS

For machines to cooperate upon a task, the master packs messages and send it across machines. The individual machines unpack messages, perform the required operation and the results are integrated to generate output.

3.3.1.4 MODULES DESCRIPTION

Master Program

The number of hosts connected in the virtual machine is identified and the number of tasks to be spawned is decided based on that. The slave program is then spawned in all the systems in the virtual machine. The given range of numbers is split into smaller ranges and the lower and upper bounds are sent to the respective spawned tasks. After the execution on slave side the results, that is, the list of prime numbers are collected at the master side. The results are then displayed.

Slave Program

After getting inputs from master side, i.e., the upper and lower bounds of each range computation is performed to check for the available prime numbers and they are collected in the array. The resulting array is sent to the master where it is displayed.

Timing Analysis

In the master program time is in Micro seconds required for the entire process of generating prime numbers is calculated to be able to determine the time optimization PVM provides very good time optimization over stand alone prime number generation programs

. rhosts File

This file will contain the name of hosts to b added, password, where the executables are stored, path of the daemon etc.

3.3.4 PERFORMANCE REQUIREMENTS

SECURITY

Security is strong in PVM. When trying to add machines, using either rsh or rexec the machine intimates for a password and only after checking, it adds the machine.

AVAILABILITY

PVM is freely downloadable software on the net. It is sufficient to install PVM on one machine in the network.

CAPACITY

This can be used to speed up larger processes using the effective power of a network of computers. This is a cost effective solution and can near the range of the speed of super computers.

RESPONSE TIME

Response time is very significant in case of parallel processing. Faster response times can be achieved by using a network of machines. With the increase in the number of systems in virtual machine configuration, response time increases.

3.4 DESIGN CONSTRAINTS

3.4.1 USER INTERFACES

Since this project is developed in character user interface mode, users are not provided with graphical facilities.

3.4.2 SOFTWARE INTERFACES WITH OTHER SYSTEMS

While many heterogeneous systems are connected together, the response time will be reduced due to message passing among many processors and because of other synchronization activities done by PVM.

3.5 DEVELOPMENTS AND OPERATING ENVIRONMENT

The development environment gives the minimum hardware and software requirements.

Hardware Specification

- Processor — Pentium III
- RAM — 64 MB
- Cache — 128KB
- Hard Disk — 10 GB
- Floppy Drive — 1.44 FDD
- Monitor — 14" Monitor

Software Specification

- Operating System — Red Hat Linux 8.0
- Language — C
- Software Package — PVM3.4

SYSTEM DESIGN

The system design is the high level strategy for solving the problem and building a solution. System design includes decisions about the organization of the system into subsystems, allocation of subsystems to hardware and software components, and major conceptual and policy decisions that form the framework for the detailed design.

Parallel **V**irtual **M**achine is a public domain software package. It is used to create and execute concurrent or parallel applications. PVM operates on a collection of heterogeneous Unix computers connected by one or more networks, which is called *virtual machine*. It is comprised of two main components:

- The PVM daemon **(pvmd3)**
- Library interface routines

Pvmd

PVM daemon, a process that serves as a message router and virtual machine coordinator. One PVD daemon runs on each host of a virtual machine.

Pvm console:

The PVM console, called pvm, is a stand-alone PVM task that allows the user to interactively start, query, and modify the virtual machine. The console may be started and stopped multiple times on any of the hosts in the virtual machine without affecting PVM or any applications that may be running.

pvm>

Some commands under pvm are

add

followed by one or more host names, adds these hosts to the virtual machine.

conf

lists the configuration of the virtual machine including hostname, pvmd task ID, architecture type, and a relative speed rating.

delete

followed by one or more host names, deletes these hosts from the virtual machine. PVM processes still running on these hosts are lost.

halt

kills all PVM processes including console, and then shuts down PVM. All daemons exit.

help

can be used to get information about any of the interactive commands. Help may be followed by a command name that lists options and flags available for this command.

spawn

starts a PVM application. Options include the following:

->

redirect task output to console.

->file

redirect task output to file.

version

prints version of PVM being used.

PVM supports the use of multiple consoles . It is possible to run a console on any host in an existing virtual machine and even multiple consoles on the same machine. It is also possible to start up a console in the middle of a PVM application and check on its progress.

Host File Options

The *hostfile* defines the initial configuration of hosts that PVM combines into a virtual machine. It also contains information about hosts that you may wish to add to the configuration later.

Some of the options are,

lo= specifies the login name
so= password
dx= location of pvmd
ep= paths to user executables
wd= working directory

Simple hostfile listing virtual machine configuration

Linux73 dx=/pvm3/pvm3/lib/LINUX/pvmd lo=root ep=/pvm3/pvm3/bin/LINUX wd=/pvm3/pvm3

90.0.1.69 dx=c:\progra~1\pvm3.4\lib\win32\pvmd3.exe lo=administrator

Programming Models:

There are two main programming models that codes written under PVM follow:

Master/worker
hostless.

Under the master/worker paradigm, one task is used as the master. The sole purpose of the master task is to create all other tasks that are designated to work on the problem, coordinate the input of initial data to each task and collect the output of results from each task

Basic PVM Calls

Here is a brief list of basic PVM calls.

tid = pvm_mytid
enrolls the process and either returns or generates a unique tid

numt = pvm_spawn

starts new PVM processes

info = pvm_pk*

pack the active message buffer with arrays of prescribed data type

info = pvm_upk*

unpacks the active message buffer into arrays of prescribed data type

info = pvm_send

send the data in the active message buffer

bufid = pvm_recv

receives a message

SYSTEM TESTING

Testing is an activity to verify that a correct system is being built and is performed with the intent of finding faults in the system. Testing is an activity. however not restricted to being performed after the development phase is complete. But this is to be carried out in parallel with all stages of system development, starting with requirement specification. Testing results once gathered and evaluated, provide a qualitative indication of software quality and reliability and serve as a basis for design modification if required.

System Testing is a process of checking whether the development system is working according to the original objectives and requirements. The system should be tested experimentally with test data so as to ensure that the system works according to the required specification. When the system is found working, test it with actual data and check performance.

Software testing is a critical element of software quality assurance and represents the ultimate review of specification, design and coding. The increasing visibility of software as a system element and the attendant "cost" associated with a software failure are the motivation forces for a well planned, thorough testing.

5.1 Testing Objectives

The testing objectives are summarized in the following three steps. Testing is the process of executing a program with the intent of finding an error. A good test case is one that has high probability of finding an error. A successful test is one that uncovers as–yet-undiscovered errors.

5.2 Testing Principles

All tests should be traceable to customer requirements. Tests should be planned long before testing begins, that is, the test planning can begin as soon as the requirements model is complete. Testing should begin "in the small" and progress towards resting "in large". The focus of testing will shift progressively from programs to individual modules and finally to the entire project. Exhaustive testing is not possible. To be more effective, testing should be one, which has highest probability of finding errors.

The following are the attributes of good tests:

- A good test has a high probability of finding an error.
- A good test is not redundant.
- A good test should be "best of breed"
- A good test should be neither too simple nor too complex.

5.3 Levels of Testing:

The details of the software functionality tests are given below. The testing procedure that has been used is as follow:

- Unit Testing
- Integration Testing
- Validation Testing
- Output Testing

Unit Testing

Unit testing is carried out to verify and uncover errors within the boundary of the smallest unit or a module. In this testing step, each module was found to be working satisfactory as per the expected output of the module. In the package development, each module is tested separately after it has been completed and checked with valid data. Unit testing exercise specific paths in the modules control structure to ensure complete coverage and maximum error detection.

The project is divided into four modules. These three modules are developed separately and verified whether they function properly.

Integration Testing

Integration Testing address the issues associated with the dual problems of verification and program construction. After the software has been integrated a set of higher-order tests are conducted. The main objective in this testing process is to take unit-tested modules and build a program structure that has been dictated by design.

The following are the types of integrated testing:

Top Down Integration:

This method is an incremental approach to the construction of the program structure. Modules are integrated by moving downward the control hierarchy, beginning with the main program module. The module sub-ordinates to the main program module are incorporated to the structure in either a depth-first or breadth-first manner.

Bottom Up Integration:

This method begins the construction and testing with the modules at the lowest level in the program structure. Since the modules are integrated from the bottom up, processing required for modules subordinate to given level is always available and the need for stubs is eliminated. The bottom up integration strategy may be implemented with the following steps:

The low level modules are combined in to clusters that perform a specific software sub-function. A driver i.e. the control program for testing is return to coordinate test case input and output. The cluster is tested and drives are removed and clusters are combined moving up ward in the program structure.

The four modules which are developed during unit testing are combined together and they are executed and verified whether the linking of the files and the corresponding modules without any error.

Validation Testing

At the end of integration testing, software is completely assembled as a package, interfacing errors have been uncovered and correction testing begins.

Validation Test Criteria

Software testing and validation is achieved through series of black box tests that demonstrate conformity with the requirements are achieved, documentation is correct and other requirement are met. Here the four modules which are checked in the integration testing are assembled and programmed in the code is testing for any correction.

Output Testing

Output testing is series of different test whose primary purpose is to fully exercise the computer based system. Although each test has a different purpose, all the work should be verified so that all system element have properly integrated and perform allocated functions.

Output testing is the stage of implantation, which is aimed at ensuring that the system works accurately and efficiently before live operation commences. The input screens, output documents were checked and required modification made to suite the program specification. Then using rest data prepared, the whole system was tested and found to be a successful one.

Here giving checks the whole system sample inputs i.e. the predefined no of threads in the scheduler and clients. It is checked for the whole function so that the system functions well and the requirements are met.

FUTURE ENHANCEMENTS

Due to the short duration that we had with us for the completion of this project, there have been some features that could not be implemented. These features are essentially enhancements to the PVM model and will affect the current implementation purposes in no significant way. The following are the features that have been postponed to later release :

1. The extension of the functionalities provided, introducing a PVM environment where all the work is carried out, introduction of systems into the machine even while the machine is running instead of configuring the machine for the systems where new task is to be run and much more enhanced functionalities.

2. The implementation of a multithreaded Dispatcher (task manager). This is essentially an optimization issue. Multithreaded Dispatcher offer better service in case a lot of systems are attached to the machine for service.

We intend to carry out these enhancements whenever we find time to concentrate once again on PVM.

CONCLUSION

To conclude, it would be appropriate to say that the initial design objectives have been satisfactorily met. All the core and important features of PVM have been implemented. The system has been reasonably optimal, considering the sub-optimal resources that we had to use throughout this project. The prime number generation application has been executed and results have been verified.

The project has been emotionally and mentally satisfying one. This project has been a big learning trip.

BIBLIOGRAPHY

Books

- Richard Stevens, "UNIX Network Programming ", Addison Wesley Longman, 1996, Second Edition.
- Richard Stevens, "UNIX Advanced Programming",Addison Wesley, 1995, First Edition.
- Michael K. Johnson and Eric W.Troan, "Linux Application Development",Addison Wesley,2000,First Edition.

Websites

- www.ieee.org
- www.csm.ornl.gov/pvm
- www.netlib.org/pvm3
- www.comp.parallel.pvm
- www.mersenne.org/prime.html

APPENDIX

9.1 SAMPLE CODE

MASTER PROGRAM

```
#include <stdio.h>
#include<string.h>
#include <stdlib.h>
#include <math.h>
#define MAXSIZE 10
#define FALSE 0
#define TRUE 1
#include "pvm3.h"

typedef unsigned long ulong_t;
typedef   ulong_t * Data;
typedef unsigned char Boolean;
typedef unsigned char Byte;

typedef struct stack {
 Data item;
 struct stack *next;
} *Stack;


Boolean push (Stack *pstk, Data item)
{
 Stack temp;

 temp = malloc (sizeof *temp);
 if (temp == NULL)
     return FALSE; /* There is no room for another element */
        temp->item = item;
 temp->next = *pstk;
 *pstk = temp;
 return TRUE; }


Boolean pop (Stack *pstk, Data *pitem)
{
 Stack temp = *pstk;
 if (temp == NULL) {
     memset (pitem, '\0', sizeof (Data));
     return FALSE; /* Stack is empty */
 }
 *pitem = temp->item; /* return item */
```

```
  *pstk = temp->next;
  free (temp);
  return TRUE; }

Boolean isPrime (ulong_t num)
{
 ulong_t n = (ulong_t) sqrt (num);
 ulong_t i;

 if (num == 2)
     return TRUE;
 if ( num == 1 || num % 2 == 0)
     return FALSE;
 for ( i = 3; i <= n; i++ )
     if (num % i == 0)
          return FALSE;
 return TRUE;
}


int gen_prime (ulong_t r1, ulong_t r2, Stack *stk)
{
 Data prime_num = malloc (MAXSIZE * sizeof (ulong_t));
 ulong_t i;
 int p_index = 0;

 for (i = r1; i <= r2; i++) {
     if (isPrime (i)) {
          if (p_index > MAXSIZE-1) {
               push (stk, prime_num);
               prime_num = malloc (MAXSIZE * sizeof
(ulong_t));
                    prime_num[0] = i;
               p_index = 1;
          }
          else
               prime_num[p_index++] = i;
     }

 }
 push (stk, prime_num);
 return p_index;
}
```

```
int main()
{
    int mytid;        /* my task id */
    int n1,i1, nproc, master, msgtype,j,bufid,id;
    unsigned long count=0;
    unsigned long low,up,data[500];
    int result,flag=0;
    char *str="prime";
    char path[30]="/pvm3/pvm3/examples/";
    char tmp[9];
    FILE *ptr;
    ptr=fopen("/pvm3/pvm3/examples/nithya.txt","a+");

    Stack stk = NULL;
    int n, i;
    Data p;
    ulong_t u;

    /* enroll in pvm */
    mytid = pvm_mytid();
    master=pvm_parent();

    /* Receive data from master */
    msgtype = 1;
    pvm_recv( master, msgtype );
    pvm_upkulong(&low, 1, 1);
    pvm_upkulong(&up, 1, 1);
   // printf("\n LOW: %lu \t",low);
   // printf("UP:  %lu",up);

    n = gen_prime (low,up,&stk);

    printf ("\n");
    if (pop (&stk, &p))
    {
     for (i = 0; i < n; i++)
     {
     //printf("%lu Linux72",p[i]);
     fprintf(ptr,"%lu\t",p[i]);

     data[count++]=p[i];

     }
     free (p);
    }
```

```
    while (pop (&stk, &p))
    {
     //printf ("\t");
     for (i = 0; i < MAXSIZE; i++)
     {
          //printf ("%lu Linux72", p[i]);
          data[count++]=p[i];
          fprintf(ptr,"%lu\t",p[i]);
     }
          free (p);
    }
    fprintf(ptr,"%c",13);


    /* Send result to master */
    pvm_initsend(PvmDataDefault);
    pvm_pkulong(&count,1,1);
    pvm_pkulong(data,count,1);
    master = pvm_parent();
    pvm_send( master,5);

    /* Program finished. Exit PVM before stopping */
    fclose(ptr);
    //pvm_kill(mytid);
    pvm_exit();

    return 0;
}
```

SAMPLE OUTPUT

Case 1: Running Prime Number Program As Stand-Alone Program

The Prime Numbers Are :

1000000409	1000000411	1000000427	1000000433	1000000439
1000000447	1000000453	1000000459	1000000483	1000000207
1000000223	1000000241	1000000271	1000000289	1000000297
1000000321	1000000349	1000000363	1000000403	1000000007
1000000009	1000000021	1000000033	1000000087	1000000093
1000000097	1000000103	1000000123	1000000181	

Time Analysis: 34734 Micro seconds

Case 2: Running Prime Number Program using PVM on two machines

[2::t40006]
[2:t80005]
[2:t80005] EOF
[2:t40006] EOF
[2:t40007]
[2:t80006]
[2:t40007] EOF
[2:t40008]
[2:t40008] EOF
[2:t80007]
[2:t80006] EOF
[2:t40005]
[2:t40005] **Number Of Processses Spawned = 6 .The Host Ids Are:**

[2:t40005] host id=524288 host id=524288 host id=524288
host id=262144 host id=262144 host id=262144

The Prime Numbers Are:

1000000007..1000000009..1000000021..1000000033..1000000087..1000000093
..1000000097..1000000103..1000000123..1000000181..1000000207.1000000223
.1000000241..1000000271..1000000289..1000000297..1000000321..1000000349
..1000000363..1000000403..1000000409..1000000411..1000000427..1000000433
..1000000439..1000000447..1000000453..1000000459..1000000483..

Time Analysis:

[2:t40005] **Time Taken To Complete The Process is : 22203 Micro Seconds**
[2:t40005] EOF

Case 3 : Running Prime Number Program Using PVM On A Single Machine

[2:t40007]
[2:t40007] EOF
[2:t40008]
[2:t40008] EOF
[2:t40009]
[2:t40009] EOF
[2:t40006]
[2:t40006] **Number Of Processses Spawned = 3.The Host Ids Are :**
[2:t40006] host id=262144 host id=262144 host id=262144

Prime Numbers Are :

1000000007..1000000009..1000000021..1000000033..1000000087..1000000093..1000000097..1000000103..1000000123..1000000181..1000000207..1000000223..1000000241..1000000271..1000000289..1000000297..1000000321..1000000459..1000000483..1000000349..1000000363..1000000403..1000000409..1000000411..1000000427..1000000433..1000000439..1000000447..1000000453..from linux73

Time Analysis :

[2:t40006] **Time Taken To Complete The Process Is : 43986 Micro Seconds**
[2:t40006] EOF