Parallelization of Prime Number Generation Using Message Passing Interface

IZZATDIN AZIZ, NAZLEENI HARON, LOW TAN JUNG, WAN RAHAYA WAN DAGANG
Department of Computer and Information Sciences
Universiti Teknologi Petronas
31750 Tronoh, Perak
MALAYSIA
{izzatdin, nazleeni, lowtanjung}@petronas.com.my, wan.rahaya@gmail.com

Abstract:-. In this research, we propose a parallel processing algorithm that runs on cluster architecture suitable for prime number generation. The proposed approach was written using Message Passing Interface (MPI) and is meant to decrease computational cost and accelerate the prime number generation process. Several experimental results conducted using High Performance Linpack (HPL) benchmark are presented to demonstrate the viability of our work. The results suggest that the performance of our work is at par with other parallel algorithms.

Key-Words: - Prime number generation, parallel processing, cluster architecture, MPI, primality test.

1 Introduction

Prime numbers which is a sub-branch of mathematics called "number theory" which have intrigued mathematicians for centuries. Despite the efforts of many mathematicians over the centuries, there is no general "formula" for generating prime numbers. However there are some approximations and theorems predicting the number of prime numbers less than a particular upper bound. As of 2003 the largest known prime number is: The 39th Mersenne prime.

Prime numbers has somehow stimulated much of interest in mathematical field or in computer security due to the prevalence of RSA encryption schemes. Cryptography often uses large prime numbers to produce cryptographic keys which can be used to encipher and decipher data. It has been identified that a computationally large prime number is likely to be a cryptographically strong prime. However, as the length of the cryptographic key values increases, this will result in the increased demand of computer processing power to create a new cryptographic key pair. In particular, the performance issue is tightly related to time and processing power required for prime number generation.

Prime number generation comprises of processing steps in searching for and verifying large prime numbers for use in cryptographic keys. This is actually a pertinent problem in public key cryptography scheme, since increasing the length of key to enhance the security level would results in a decrease in performance of a prime number generation system.

Another trade off resulting from using large prime numbers is pertaining to the primality test. Primality test is the intrinsic part of prime number generation and yet the most computational intensive sub process. It has also been proven that testing the primality of large candidates is very computationally intensive.

Apart from that, the advent of parallel computing or processing has invited many interests to apply parallel algorithms in a number of areas. This is because it has been proven that using parallel processing can substantially increase the processing speed.

Choosing the best parallel programming paradigm is also another concern when it comes to parallelization of an application or algorithm. There are a few parallel programming paradigms available such as Message Passing Interface (MPI) ,OpenMP, View-Oriented Parallel Programming (VOPP) or Parallel Virtual Machine (PVM).

We have chosen MPI as the paradigm of choice due to the nature of our problem, the hardware components and the network setup that we have in the laboratory. MPI consists of specifications for message passing libraries that can be used to write parallel programs. It is a widely accepted standard and provides the programmers with a programming model where processes communicate with each other by calling library routines to send and receive messages.

This message passing paradigm not only can be employed within a node but also across several nodes in a cluster. This is the advantage of MPI over OpenMP. Unlike OpenMP, MPI is also viable for wide range of problems. Besides that, MPI offers the user's complete control over data distribution and process synchronization. This feature is vital in order to ensure optimum performance of the parallelization. MPI also provides supports that allow heterogeneity in its specifications. With this capability, it will enable the implementation of our parallelization to run on heterogeneous network of workstations.

PVM may be more suitable for heterogeneous network setup and although MPI does not have the concept of a virtual machine, MPI does provide a higher level of abstraction on top of the computing resources in terms of the message-passing topology. In MPI a group of tasks can be arranged in a specific logical interconnection topology. Communication among tasks then takes place within that topology with the hope that the underlying physical network topology will correspond and expedite the message transfers.

PVM does not support such an abstraction, leaving the programmer to manually arrange tasks into groups with the desired communication organization [13]. Both MPI and VOPP can be adopted for parallelization on distributed memory parallel computers. However, VOPP programs are not as efficient as MPI programs when the number of processors becomes larger [14].

In this paper, we present a parallel processing approach in cluster architecture for prime number generation using MPI that would provide improved performance in generating cryptographic keys.

2 Related Work

Despite the importance of prime number generation for cryptographic schemes, it is still scarcely investigated and real life implementations are of rather poor performance [1]. However, a few approaches do exist in order to efficiently generate prime numbers [1-5].

Maurer proposed an algorithm to generate provable prime numbers that fulfill security constraints without increasing the expecting running time [2]. An improvement has been made to Maurer's algorithm by Brandt et al to further speed up the prime number generation [3]. Apart from that, the proposed work has also included a few ways for further savings in prime number generation [3].

Joye et al has presented an efficient prime number generation scheme that allows fast implementation on cryptographic smart card [1]. Besides that, Cheung et al has originated a scalable architecture to further speed up the prime number validation process at reduced hardware cost [4]. All of these researches however, were focusing on processing the algorithm sequentially.

It has been proven that tasks accomplished through parallel computation results in faster execution as compared to a computational processes that runs sequentially [9]. Tan et al has designed a parallel pseudo-random generator using Message Passing Interface (MPI) [5]. This work is almost similar to ours but with different emphasis. The prime numbers generated are to be used for Monte Carlo simulations and not cryptography. Furthermore, considerable progresses have been made in order to develop high-performance asymmetric key cryptography schemes using approaches such as the use of high-end computing hardware [6, 7, and 8].

3 Methodology

Traditional approach of system development methodology that needs to get the development model mostly correct in the early stage is impossible as this involves more than just one area of studies such as prime number generation algorithm, primality tests, parallel processing and MPI. Various issues need to be considered that may be unforeseen at the beginning stage of development. Thus different conditions and techniques would involve during development phase.

Evolutionary development is an iterative and incremental approach for system development. The system will be delivered incrementally over time. Evolutionary development is new to many existing professional developer and many traditional programmers as well. Fig. 1 illustrates the phases involved in evolutionary development approach.

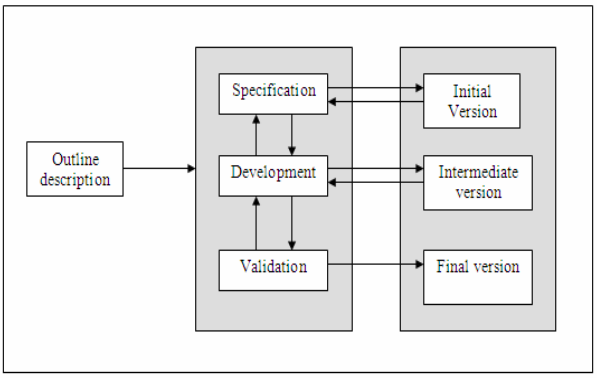

Fig. 1: Phases involved in Evolutionary Development Approach

3.1 Specification Phase

A sequential program of prime number generation in C using MPI libraries is developed. Then in this phase the parts of the sequential program that could be parallelized would be identified. This is the beginning of the specification phase. Although the main objective is to parallelize the prime number generation, but not all part of the program can be parallelized. This is where the partitioning stage of the programming design takes place which is intended to explore the opportunities for parallel execution.

3.2 Development Phase

As mentioned earlier, the parallelization of the algorithm was achieved by using MPI libraries. The parallel program was written incrementally over time which means troubleshooting was done on the program from time to time to avoid error that could not be debugged later on.

3.3 Validation Phase

The program prototype will then go through the validation phase to ensure the project requirements are achieved. If there are still areas that need to be modified and altered, the whole phases will be repeated all over again until the final version of the program is released. Most of the evaluation processes were carried out by the authors.

4 Development Tools

The main reason of choosing C to write the program is because it provides an sequential infrastructure that accommodates mechanism of breaking down the problem into a collection of data structures and operations that is matching the characteristic of parallel processing.

Furthermore, C is also compatible with the concept of partitioning and dynamic memory allocation which, are the concept that is going to be deployed in the parallelization of prime number generation. As mentioned earlier, MPI is used for the parallel processing of the algorithm; a library of subroutine specifications that can be called from C , this is also another reason why the parallel program is written using C. The application that is used to edit the program is Linux gnu.

4.1 Libraries

MPI provides all the subroutines that are needed to break the tasks involved in the massive computational process into subtasks that can be distributed to a number of available nodes for processing. The goal of the MPI is to establish a portable, efficient, and flexible standard for message passing that will be widely used for writing message passing programs. MPI provides an appropriate environment for general purpose message-passing programs, especially programs with regular communication patterns. Fig. 2 shows the general MPI program structure:

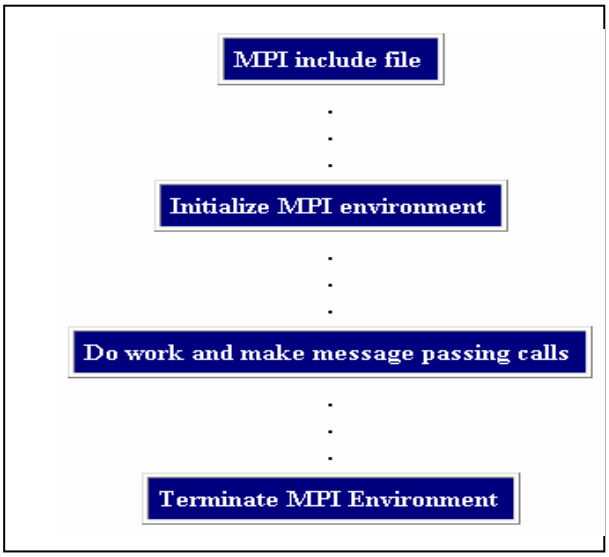

Fig. 2: General MPI Program Structure

MPI contains approximately 125 functions that greatly ease the tasks in implementing common communication structures, such as send-receive, broadcasts and reductions. However, MPI is reasonably easy to learn as a complete message-passing program can be written with just six basic functions.

MPI contains useful communications libraries for applications that need to be ported to various platforms. Different versions of MPI exist for virtually every major platform: message-passing supercomputers, scalable shared-memory machines, symmetric multiprocessors, loosely-coupled workstation clusters, and even individual PCs. With MPI, the programmer can write code *once* and merely recompile it for each new platform.

4.2 Platform

The parallel program of prime number generation will be tested and run on a grid computing

platform. It is worth mentioned here that the idle workstations available in the lab were put together to form a single cluster running MPI programs.

5 System Model

5.1 Experimental Setup

Fig 3 shows the experimental cluster set up in the UTP lab which comprised of 20 SGI machines. Each of the machines consists of off-the-shelf Intel i386 based dual P3-733MHz processors with 512MB memory Silicon Graphics 330 Visual Workstations. These machines are connected to a Fast Ethernet 100Mbps switch. The head node performs as master node with multiple network interfaces [10]. Although these machines may not be as powerful as the latest cluster-machine in terms of the hardware and performance, the important focus would be the parallelization of the algorithm and how jobs can be disseminated among the processors.

Fig.3 UTP Cluster

The software stack on all machines is consisting of Linux Ubuntu 5.10 operating system, MPICH-1.2.7p1 and openMosix for Kernel 2.4.26 stable cluster middlewares, parallel High Performance Linpack (HPL) version 1.0a and Flops.c version 2.0 both for parallel benchmark and individual node flops benchmark, GCC-3.3.6 with Basic Linear Algorithm Subroutine (BLAS) version 3.0 as the program compiler and its supporting math library, and lastly is the MPI communication benchmark using mpptest (part of perftest version 1.3b). The reasoning why we run only HPL C version is by the assumption that the majority of application programs are based on C programming language rather than other programming languages in our implementation [10].

5.2 Number Generation

In order to generate a number, a random seed is picked and input into the program. The choice of seed is crucial to the success of this generation as it has to be as random as possible. Otherwise anyone who uses the same random function would be capable of generating the primes, thus beats the purpose of having strong primes.

5.3 Primality Test

We have selected trial division algorithm as the core for primality test. Basically trial division divides an n-bit random number by primes up to $\sqrt{(n)}$ is a kind of deterministic primality test. This algorithm is based on a given composite integer n, and trial division consists of trial-dividing n by every prime number less than or equal to $\sqrt{(n)}$. If a number is found which divides evenly into n, that number is a factor of n.

In other words trial division primality test is a method of sequentially trying test divisor into a number n so as to partially or completely factor n. The process starts with the first prime divisor, i.e. 2, and keeps dividing n by 2 until no more division can be done, then the next prime, i.e. 3, is used as divisor on the remaining unfactored portion. The process is repeated until a trial divisor that is greater than the square root of the unfactored portion since this unfactored portion is a prime.

In factorization by trial division [11] up to a specified maximum, m can be given in a form of,

$$n = P_1^{e1} \text{ x } P_2^{e2} \text{ x } \ldots \text{ x } P_r^{er} \text{ x } f$$

Where f is the unfactored portion and f is larger than the square of the largest trial divisor or $f = 1$. The algorithm to implement this shall be,

get n , m;
i = 0; /*counts the number of distinct prime factors*/
f = n; /*records the still unfactored portion*/
for d = 2 to 3 do {
    if (f mod d) = 0 then Divide(f,d,i)
    d = 5; inc = 2;}
while d <= m and d^2 <= f do { /*trial loop*/
    if (f mod d) = 0 then Divide(f,d,i);
        d = d + inc;
    inc = 6 – inc; }
if $d^2 > f$ then do {/*a big prime*/
    i = i + 1;P_I = f;ei = 1;
    f = 1; } /* if $d^2 > f$ then f is a prime*/
Divide(f,d,i) {
    i = i + 1;
    P_i = d;
    ei = 1;
    f = f/d;

While (f mod d) = 0 do {
$ei = ei + 1$;
$f = f/d$; }
return }

In speeding up the trial division process we can exploit the fact that after the first prime of 2, the rest of the primes are odd. So 2 and odd numbers may be used as trial divisor. A short description of this algorithm is given by [12]. The algorithm produces the multiset $\mathcal{T}$ of the primes that divide n, given that $n > 1$.

1. [divide by 2]
$\mathcal{T} = \{\}$; /*empty set*/
$N = n$;
while (2|N) { $N = N/2$;
$\mathcal{T} = \mathcal{T} \cup \{2\}$;}
2. [main division loop]
$d = 3$;
while($d^2 <= N$) {
while($d|N$) { $N = N/d$;
$\mathcal{T} = \mathcal{T} \cup \{d\}$; }
$d = d + 2$;}
if ($N == 1$) return $\mathcal{T}$;
return $\mathcal{T} \cup \{N\}$;

5.4 Parallel Approach

Once a random number have been generated, master node will create a table of dynamic 2D array, which later will be populated with odd numbers. As shown in Fig.4, a pointer-to-pointer variable ***table* in master, will points to an array of pointers that subsequently points to a number of rows. This will result in a table of dynamic 2D array. After the table of dynamic 2D array is created, master will then initialize the first row of the table only.

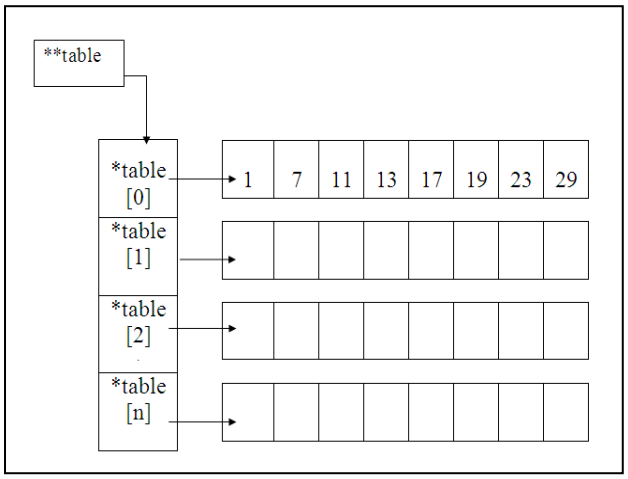

Fig.4 Master creates a dynamic 2D array to be populated with odd numbers

The parallel segment begins when master node broadcasts the row[0] to all nodes by using MPI_Bcast. This row[0] will be used by each node to continue populating the rest of the rows of the table with odd numbers. Master node will then equally divide *n-1* number of rows left that is yet to be populated by number of nodes available in the grid cluster. Each node will be given an equal number of rows to be populated with odd numbers.

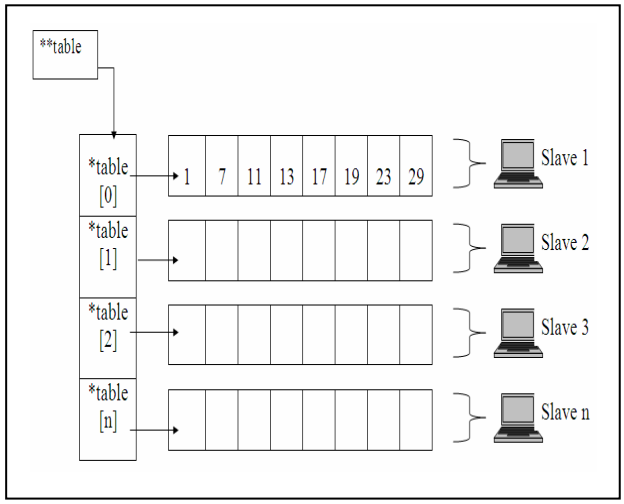

Fig.5. Master sends an equal size of row to each slave

This could be achieved by using MPI_Send. A visual representation of this idea is depicted in Fig.5.Each node will receive n numbers of rows to be populated with odd numbers. This is where the parallel process takes place. Each node will process each row given concurrently. Each node will first populate the rows with odd numbers. Then they will filter out for prime numbers using the primality test chosen. The odd prime numbers will remain in the rows but those that are not will be assigned to NULL. Each populated row are then returned to master node, whom then randomly pick for three distinct primes for the value of *p,q,* and public key *e* of the cryptographic scheme.

As an example, if there are 4 processors available to execute the above tasks, and there are 1200 rows need to be populated with prime numbers, each slave will be given 300 rows to be processed. The overall procedure is depicted in Fig.6.

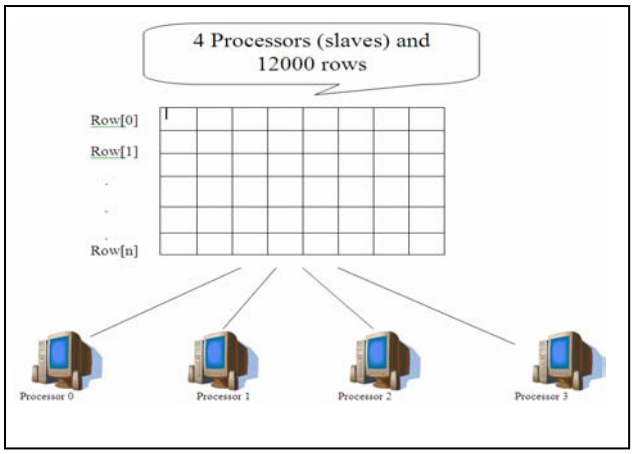

Fig.6. Example of assigning 1200 rows to 4 processors (slaves)

Processor 0 will process row(1) up to row(299), processor 1 will be processing row(300) up to row(599), processor 2 will be processing row(600) up to row(899) and lastly processor 3 will be processing row(900) up to the last row, row(1199).

After each node returns the populated rows to master node, it will then pick randomly prime numbers to be assigned as the value of *p, q,* and *e*. These values can later be used for encryption and decryption part of a cryptosystem algorithm. It is to be reminded that the parallel process that takes place in the whole program is only on the prime number generation.

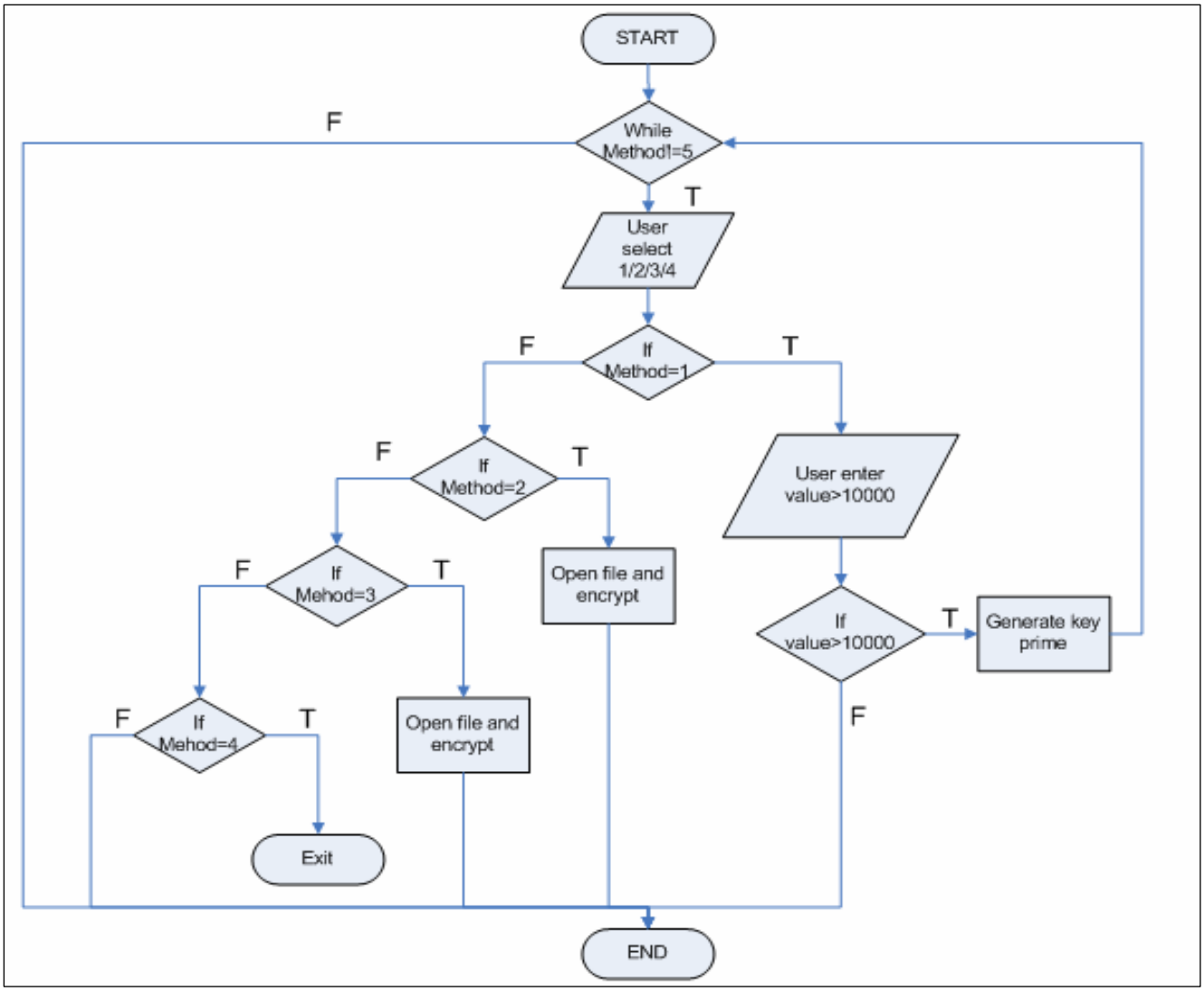

Fig.7. Sequential algorithm in a flow chart form

6 Sequential Algorithm

6.1 Sequential Algorithm

The meticulous process of the sequential algorithm to generate prime number is presented in Fig.7 above. This process is useful as later on it can be used to compare time execution results with the proposed parallel approach. Using the same primality test as mentioned in section 5.3, a loop is utilized to iterate the process. A series of selection takes place to finally produce the prime number. The sequential program as shown below uses $x^2 \leq n$ which, is equivalent to $x \leq \sqrt{n}$

```
unsigned int trial_division_squaring (unsigned int n) {  unsigned int x, x_squared;

   for(x=2, x_squared=4;
      x_squared > 2*x – 1 && x_squared <= n;
      x++, x_squared += 2*x – 1)
   {
      if ((n % x) == 0) {
         return x;
      }
   }
   return IS_PRIME;}
}

int main(int argc, char* argv[]) {
   int i;
   unsigned int n = atoi(argv[2]);

   if (strcmp(argv[1], “trial_division_sqrt”) == 0) {
      for(i=0; i<10000 – 1; i++)
trial_division_sqrt(n);
      printf(“%u\n”, trial_division_sqrt(n));
   }
 else if (strcmp(argv[1], “trial_division_squaring”) == 0) {
      for(i=0; i<10000 – 1; i++)
trial_division_squaring(n);
      printf(“%u\n”, trial_division_squaring(n));
   }
 else if (strcmp(argv[1], “trial_division_odd”) == 0) {
      for(i=0; i<10000 – 1; i++)
trial_division_odd(n);
      printf(“%u\n”, trial_division_odd(n));
   }
else if (strcmp(argv[1], “trial_division_primes”) == 0) {
      generate_prime_list(65536);
      for(i=0; i<10000 – 1; i++)
trial_division_primes(n);
      printf(“%u\n”, trial_division_primes(n));
   } else {
      printf(“Invalid algorithm selection.\n”);
   }
   return 0;}
```

6.2 Proposed Parallel Algorithm

The algorithm of the parallel program is as follows:

Start
Master creates a table of odd numbers and initialized row [0] only
Master broadcasts row [0] to all slaves
Master sends a number of rows to each slave
Each slave will receive an initialized row from master
Each slave will populate row prime numbers
Each slave will return populated row to Master
Master waits for results from slaves
Master receives populated rows from each slave.
Master checks unpopulated rows
If maxRow > 0
Master sends unpopulated row to slave
Master picks prime numbers randomly
Prompt to select program option
Switch (method)
Case 1: prompt to enter a value greater than 10000
If value > 10000, generate key primes
Else, Exit program
Case 3: open file and decrypt
Case 4: exit program
End
End

A better understanding of the algorithm is perhaps best represented in the flowchart form as depicted in Fig.8. As clearly shown, the parallelization takes place mainly at

generating prime number. Slaves' processors will play an active participation during this part; where as the master processor would mainly disseminate and gather the finalized result. This piece of algorithm would basically exist in each and every processor upon the execution of run command in console.

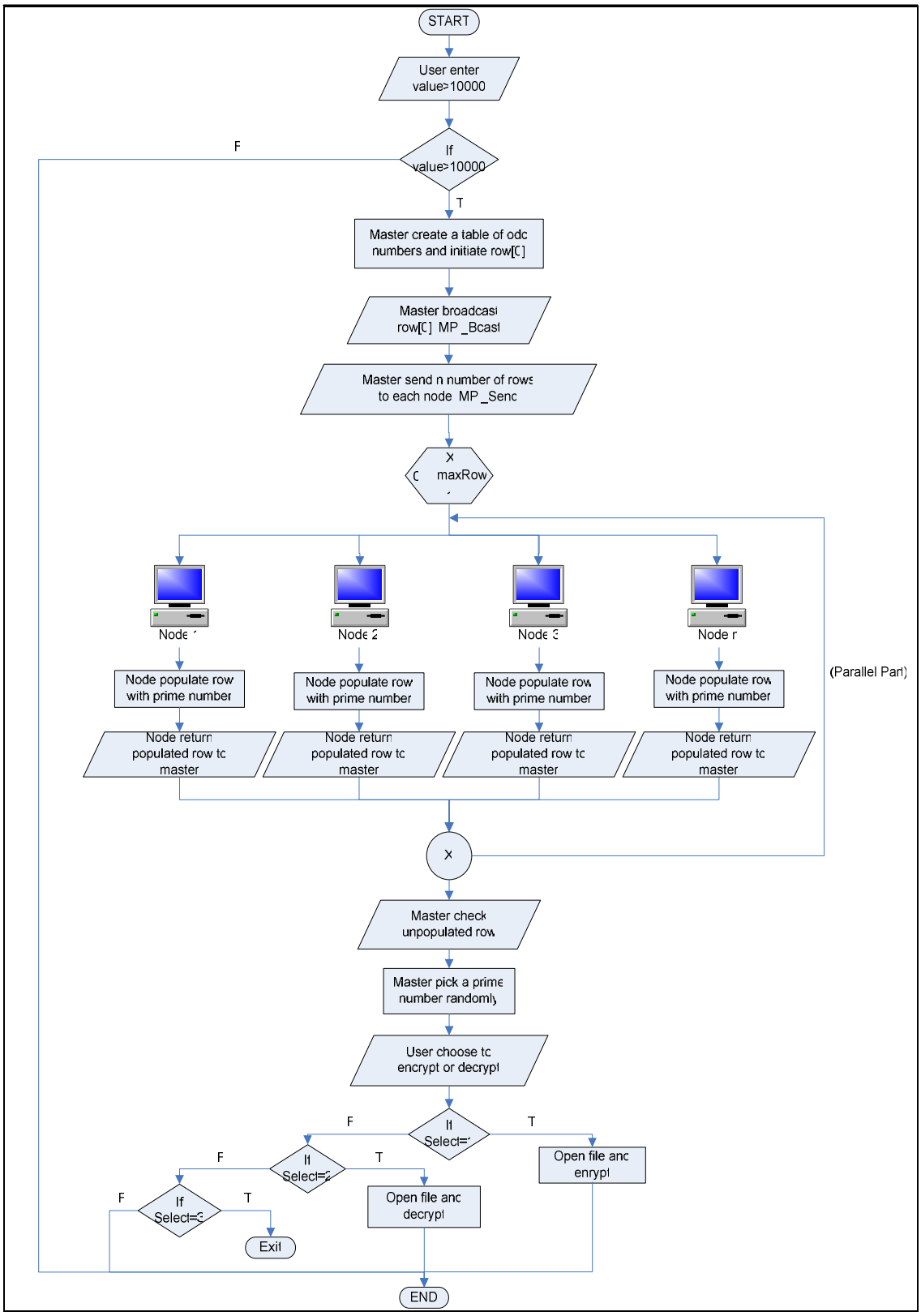

Fig.8.The proposed parallel algorithm represented in a flowchart form

7 Evaluation

It is significant to report that the time measurement obtained from the execution using a sequential prime number generation was at average of 7.6 ms for three trials. Subsequent paragraph discussed the time measurement with the parallel approach.

Table 1: Comparison of Execution Time for Different Number of Nodes.

Number of nodes	Execution Time (ms)
1	7.850
2	0.039
3	0.039
4	0.042
5	0.043
6	0.043
7	0.049
8	0.051
9	0.051
10	0.053
15	0.060
20	0.072
25	0.089
30	0.093

Table 1 shows timing measurements of the parallel implementation of the prime number generation using various numbers of nodes. The times are obtained by calculating the total time taken by master node to compute its tasks as well as the total time for slave node to complete the assigned tasks. As expected, there is a significant improvement of performance using three nodes over one node. However, the performance is degraded slowly with the increasing number of nodes. As shown in the Table 1, using 30 nodes did not improve much performance over using 5 nodes, although it is 6 times more nodes, yet the different of the execution times is just 0.05 milliseconds. We noted that, this effect is due to the fact that the communication overhead now outweighs any reduction in computation time. As the problem increases, however, the scalability would improve, resulting in higher efficiency for larger number of nodes.

It would be trivial to compare the sequential execution of the prime number generation algorithm with a single node execution of the parallel approach. Noted, that the sequential approach is faster by 0.25 ms as compared to the parallel approach. This is plausibly due to the masters demanding tasks to disseminate the jobs to other processors even though it is a single node execution. However master would still need to perform this tasks as commanded in the parallel algorithm.

Fig.9 shows the performance measurement using MPI_GATHER tested on 15 nodes. This figure was captured using the MPICH Jumpshot4 tool to evaluate the algorithm usage of MPI libraries. The numbers plotted shows the amount of time taken for each node to send back the prime numbers discovered back to the master node.

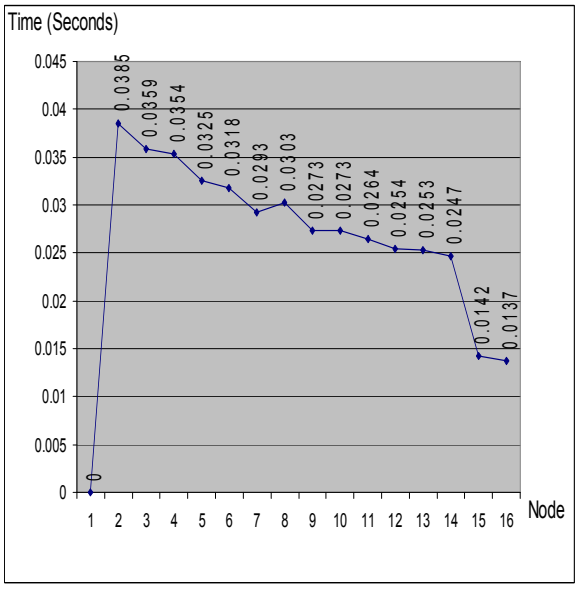

Fig.9. Time taken for MPI_BCAST and MPI_GATHER running on 15 nodes.

From the figure, it is observed that the algorithm gather was massive for the first node and deteriorated as it approached the last node. This is due to the frequent prime numbers discovered at the beginning of the number series and becomes scarce as the numbers becomes larger towards the end. This will prove that the relative frequency of occurrence of prime numbers decreases with size of the number which result in lesser prime numbers were sent back to master node by later nodes.

7.1 Benchmarking

High Performance Linpack (HPL) is chosen over other parallel benchmarking application since this utility provides the necessary computational performance figure that we set to achieve. A quick description of HPL is an application to identify the computational performance using dense matrix factorization [15] [16].

The basic platform as skimmed earlier is parallel HPL P3-CBLAS, again under the assumption that the majority of applications are in C using Linux (or OpenSource) BLAS library

from Automatically Tuned Linear Algebra Software (ATLAS) project. Other math library has been reported to have better performance at the cost of less stability (for example, GOTO is faster but has instability issue). Throughout the performance measurement, only the GCC version 3.3.6 is used due to the unfortunate fact that the recommended openMosix stable patch only available for Linux Symmetrical Multi-Processor (SMP) kernel 2.4.26 machine (Linux kernel 2.4 is only compilable using GCC 3.xx and lower).

After countless trial run and reference study from numerous resources resemblance to our cluster testbed, few essential parameters are established and the complete parameters list is attached in the appendix section of this paper [17] [18].

HPL algorithm is based on block cyclic distribution to load-balance the matrix-matrix multiplication and LU decomposition, however it may not be the perfect solution for heterogeneous environment [19]. In brief description, HPL algorithm is to solve a linear system of order n, $Ax = b$, by computing the LU factorization using partial row pivot of n-by-n+1 coefficient matrix $[A \quad b] = [[L,U] \quad y]$ in which data is distributed onto matrix PxQ grid of processes to achieve load balance and scalable algorithm.

The best problem size (Ns) is determined by the maximum available physical memory just before start using the virtual memory (swapping to slower storage, i.e. harddrive) which shall be avoided. Hence advisable figure is around 80% of the total available memory from all nodes involved in benchmarking and leaving 20% for the operating system and any other necessary applications.

The optimum matrix block size (NBs) is recommended by the HPL's guideline somewhere between 32-256 and our trial run found 144, as shown on the result chart to be the best block size to achieve good data distribution for computational granularity, i.e. smaller NB the more balanced load but too small will not utilize the data reuse in memory for computing performance [20].

For the dimension of process rows and columns for LU matrix factorization (PxQ = total #s of nodes), it is suggested for simple ethernet network interconnection to have fairly flat process grid (close figures of lower Ps and higher Qs) are recommended due to performance and scalability limitation of HPL benchmark. So 4-by-9 matrix to equaling 36 nodes total provides the optimum result.

HPL LU matrix panel decomposition broadcast pattern has a significant role in distributing the process via the MPI message. Hence, both the parameters with panel broadcast 2, the increasing-2-ring and panel broadcast 3, the increasing-2-ring modified, are the optimum for our performance measurement.

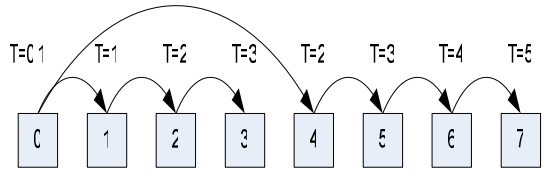

Figure 10. Panel Broadcast 2, The Increasing-2-Ring

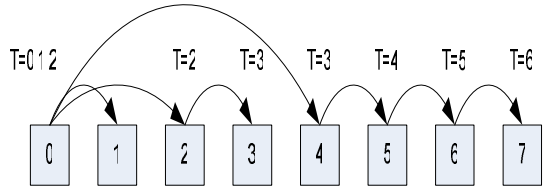

Figure 11. Panel Broadcast 3, the Increasing-2-ring modified

However the hybrid cluster does not take the advantage of this communication feature since all process initiated by MPICH will only be run locally and be distributed by openMosix's mosrun later on.

7.2 Parallel Benchmark Result

The efficiency is calculated from the theoretical performance figure, which is 12.138 Gflops using the previous single node benchmark and the theoretical formula.

Table 2: 36 Nodes Performance and Efficiency for Beowulf

NB	Gflops	% efficiency
112	6.982	0.57
128	7.160	0.59
144	7.394	0.61
160	7.026	0.58

Table 3: 36 Nodes Performance and Efficiency for Hybrid

NB	Gflops	% efficiency
112	1.570	0.129
128	1.601	0.132
144	1.600	0.132
160	NA	NA

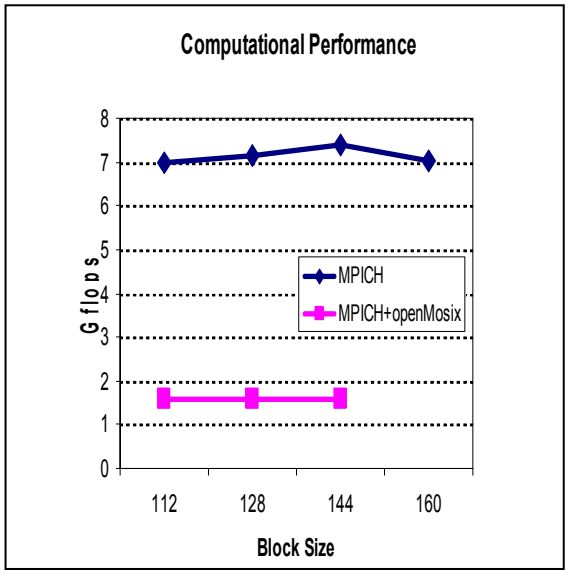

Fig 12. HPL Benchmark result

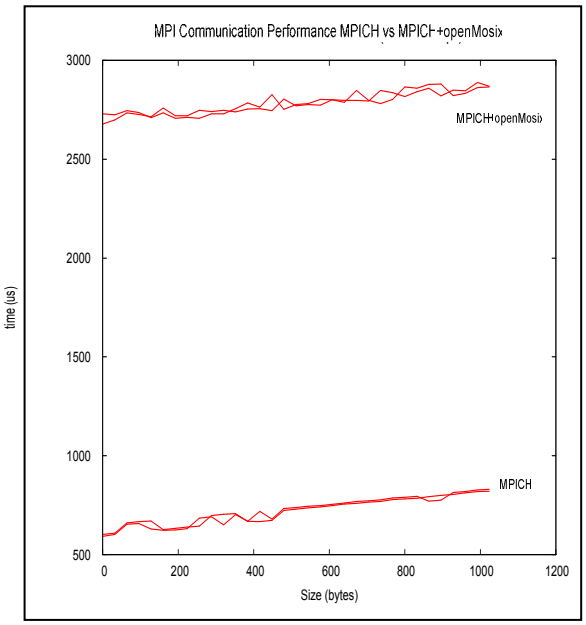

Fig 13. MPI Communication Performance

The result explain the poor performance of the multiprocessing in single node approach in which the MPI communication library becomes the main barrier in achieving high performance. This is a very strong evidence that it follows in which as the processor number (n) increases the performance decreases due to the overhead of multiprocessing on the very same node. Load imbalance may also contribute since openMosix does it's own load balancing algorithm while HPL via MPICH also tries to balance in distributing the process, hence only partial nodes are utilized at any given time (HPL using MPICH alone will distribute and load balance evenly on all computing nodes).

This MPI communication benchmark is somewhat consistent and able to explain the HPL benchmark results for relative comparison purposes. A performance figure by a factor of roughly 1/4 to 1/5 for hybrid cluster is substantially slower compare to pure MPI cluster.

8 Future Work

Our work raises a number of issues for further researches. One of the important issues is to find the suitable load balancing algorithm for allocating the tasks among the nodes efficiently. Currently, all tasks are assigned at random to all nodes, which means some nodes will receive simple tasks (due to small prime numbers) and some will need to perform very exhaustive tasks due to big prime numbers to resolve. As a result, the workload of the nodes will vary during the run time and subsequently give impact to the overall performance of the prime number generation. Therefore, it will be interesting to study on what metrics are needed to determine the node's workload so that the tasks allocation can be made efficiently. By having a suitable load balancing algorithm, substantial speed up of execution of the application is expected.

Another issue that worth investigating is pertaining to dynamic memory allocation. We are using fixed size array and we would like to investigate the performance of the application if linked list is used instead. It has been mentioned in the literature that linked list is more flexible in terms of inserting and deleting elements. However, this is yet to be tested for this application.

We are also currently working on resolving the communication overhead problem. From the results, it can be clearly seen that as number of nodes grows, the communication overhead will as well increase and outweigh any reduction in computation time.

$$nP/cpu$$

9 Conclusion

We have proposed a parallel approach to accelerate the process of prime number generation. The parallel algorithm was implemented on a clustered architecture and being tested with large prime numbers. The outcome of this approach demonstrated a high degree of reduction in prime number generation time. This reduction is possible due to the parallel generation of prime number on each of the grid nodes. However, the authors shall suggest the followings that may provide further improvements:

(1) Use other primality test that is more significant or feasible for large prime number generation such as Rabin-Miller algorithm.

(2) Use other random number generation that can produce random numbers with less computation yet provides higher security level. One good candidate shall be the prime number sieve i.e. the sieve of Eratosthenes [21]. In this implementation the upper bound of the sieve must be specified, say *n*. The generalized algorithm may be expressed as below:

Eartisthenes(n)
{initialization}
A[1] $\leftarrow$ *0*
For i = 2 to n do a[i] $\leftarrow$ *1*
p = 2
while p^2 *<= n do*
{sieve out multiples of p}
For j = p to [n/p] do a[jp] = 0
{find the next prime}
Repeat p = p+1 until a[p] = 1
Return(a)

Note that we can avoid storing the integers themselves (which might need a large storage space) by storing a 1 in location *i* if *i* is a prime, and 0 otherwise. This algorithm returns an array of such that for all I with 1 <= *i* <= *n*, we have a[*i*] = 1 if *i* is prime, otherwise a[*i*] = 0.

However do take note that in practice this algorithm is more of storage space concerned rather than time which may be interpreted in the advantage of fast prime number generation i.e. less time concerned.

(3) To compare the parallel algorithm proposed with other parallel algorithms using the same system model and hardware setting.

References:

[1] M. Joye, P. Paillier and S**.** Vaudenay, Efficient Generation of Prime Numbers, *Cryptographic Hardware and Embedded Systems*, vol. 1965 of *Lecture Notes in Computer Science,* pp. 340-354, Springer-Verlag, 2000.

[2] Maurer, Fast Generation of Prime Numbers and Secure Public-Key Cryptographic Parameters, *Journal of Cryptology*, vol.8 no.3 (1995), 123-156.

[3] J.Brandt, I. Damgard, and P. Landrock. Speeding up prime number generation. *In Advances in Cryptology -- ASIACRYPT '91*, vol. 739 of *Lecture Notes in Computer Science*, pp. 440--449, Springer-Verlag, 1991.

[4] Cheung, R.C.C., Brown, A., Luk, W., Cheung, P.Y.K., A Scalable Hardware Architecture for Prime Number Validation, *IEEE International Conference on Field-Programmable Technology, 2004.* pp. 177-184, 6-8 Dec. 2004.

[5] Tan, C. J. and Blais, J. A. PLFG: A Highly Scalable Parallel Pseudo-random Number Generator for Monte Carlo Simulations. *8th international Conference on High-Performance Computing and Networking* (May 08 - 10, 2000). *Lecture Notes In Computer Science*, vol. 1823. Springer-Verlag, London, 127-135.

[6] Agus Setiawan, David Adiutama, Julius Liman, Akshay Luther and Rajkumar Buyya, *GridCrypt :* High Performance Symmetric Key Cryptography using Enteprise Grids. *5th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT 200)* , Singapore. Springer Verlag Publications (LNCS Series), Berlin, Germany. December 8-10, 2004.

[7] Praveen Dongara, T. N. Vijaykumar, Accelerating Private-key cryptography via Multithreading on Symmetric Multiprocessors. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, March 2003.

[8] Jerome Burke, John McDonald, Todd Austin, Architectural Support for Fast Symmetric-Key Cryptography. *Proc. ACM Ninth Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX),* Nov. 2000.

[9] Selim G Aki, Stefan D Bruda, Improving A Solution's Quality Through Parallel Processing. *The Journal of Supercomputing* archive.Volume 19 , Issue 2 (June 2001).

[10] Dani Adhipta, Izzatdin Bin Abdul Aziz, Low Tan Jung, Nazleeni Binti Haron .Performance Evaluation on Hybrid Cluster: The Integration of Beowulf and Single System Image*, The 2nd Information and Communication Technology Seminar (ICTS),Jakarta.* August 2006.

[11] David M Bressoud, "*Factorization and Primality Testing*", Springer Book 1989.

[12] Richard Crandall, Carl Pomerance, *"Prime Numbers A Computational Perspective"*, 2nd edition, Springer Book.

[13] Gropp W., and Lusk E., Why are PVM and MPI so Different? *Technical Report PREPRINT ANL/MCS-P667-0697,* Mathematics and Computer Science Division, Argonne National Laboratory, June 1997.

[14] Huang,Z., Purvis, M., and Werstein P., Performance Comparison between VOPP and MPI. *In: Proc. of the Sixth International Conference on Parallel and Distributed Computing, Applications and Technologies*, pp.343-347, IEEEComputer Society (2005).

[15] J. J. Dongarra, "Performance of Various Computers Using Standard Linear Equations Software," 2006.

[16] P. M. Papadopoulos, C. A. Papadoulos, M. J. Katz, W. J. Link, and G. Bruno, "Configuring Large High-Performance Clusters at Lightspeed: A Case Study," 2005

[17] O. I. Vdovikin, "Running High-Performance Linpack on IBM pSeries JS20 cluster with Myrinet interconnect."

[18] F. Crawford, "Building Australia's Fastest Computer," vol. AUUG 2004, 2004.

[19] Y. Kishimoto and S. Ichikawa, "An Execution-Time Estimation Model for Heterogeneous Clusters," Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04), 2004.

[20] Z. Wenli, F. Jianping, and C. Mingyu, "Efficient Determination of Block Size NB for Parallel Linpack Test," 2004

[21] Eric Bach, Jeffrey Shallit, "Algorithmic Number Theory – Efficient Algorithms", Foundations of Computing Series, Vol.1, The MIT Press.