MPI Based Cluster Computing for Performance Evaluation of Parallel Applications

Nanjesh B R
Department of Computer Science and Engineering
Adichunchanagiri Institute of Technology
Chikmagalur, Karnataka, INDIA
nanjeshbr@gmail.com

Vinay Kumar K S
Department of Computer Science and Engineering
Adichunchanagiri Institute of Technology
Chikmagalur, Karnataka, INDIA
vinayks109@gmail.com

Madhu C K
Department of Computer Science and Engineering
Adichunchanagiri Institute of Technology
Chikmagalur, Karnataka, INDIA
madhu9587@gmail.com

Hareesh Kumar G
Department of Computer Science and Engineering
Adichunchanagiri Institute of Technology
Chikmagalur, Karnataka, INDIA
hareeshkumarg21@gmail.com

Abstract— **Parallel computing operates on the principle that large problems can often be divided into smaller ones, which are then solved concurrently to save time (wall clock time) by taking advantage of non-local resources and overcoming memory constraints. The main aim is to form a cluster oriented parallel computing architecture for MPI based applications which demonstrates the performance gain and losses achieved through parallel processing using MPI. This can be realized by implementing the parallel applications like parallel merge sorting, using MPI. The architecture for demonstrating MPI based parallel applications works on the Master-Slave computing paradigm. The master will monitor the progress and be able to report the time taken to solve the problem, taking into account the time spent in breaking the problem into sub-tasks and combining the results along with the communication delays. The slaves are capable of accepting sub problems from the master and finding the solution and sending back to the master. We aim to evaluate these statistics of parallel execution and do comparison with the time taken to solve the same problem in serial execution to demonstrate communication overhead involved in parallel computation. The results with runs on different number of nodes are compared to evaluate the efficiency of MPI based parallel applications. We also show the performance dependency of parallel and serial computation, on RAM.**

Keywords—Parallel Execution, Cluster Computing, Symmetric Multi-Processor (SMP), MPI (Message Passing Interface), RAM (Random Access Memory).

I. INTRODUCTION

Merge sort is an efficient divide-and-conquer sorting algorithm. Because merge sort is easier to understand than other useful divide-and-conquer methods. One common example of parallel processing is the implementation of the merge sort within a parallel processing environment. This paper deals how to handle merge sort problem that can be split into sub-problems and each sub-problem can be solved simultaneously. With computers being networked today, it has become possible to share resources like files, printers, scanners, fax machines, email servers, etc. One such resource that can be shared but is generally not, is the CPU. Today's processors are highly advanced and very fast, capable of thousands of operations per second. If this computing power is used collaboratively to solve bigger problems, the time taken to solve the problem can reduce drastically. However the whole operation of parallel processing also depends on the RAM available to the processors for their computation.

A. Existing Frameworks

1) *MPI*: The specification of the Message Passing Interface (MPI) standard 1.0 [9] was Completed in April of 1994. This was the result of a community effort to try and define Both the syntax and semantics of a core message-passing library that would be useful to a Wide range of users and implemented on a wide range of Massively Parallel Processor (MPP) platforms.

2) *MPI2*: All major computer vendors supported the MPI standard and work began on MPI-2, where new functionality, dynamic process management, one-sided communication, cooperative I/O, C++ bindings, Fortran 90 additions, extended collective operations, and miscellaneous other functionality were added to the MPI-1 standard [9]. MPI-1.2 and MPI-2 were released at the same time in July of 1997. The main advantage of establishing a message-passing standard is portability.

3) *Openmp*: It has emerged as the standard for shared-memory parallel programming. The openmp application program interface (API) provides programmers with a simple way to develop parallel application for shared memory parallel computing.

4) *MPICH2*: An all-new implementation of MPI designed to support both MPI-1 and MPI-2. In MPICH2, the collective routines are significantly faster and has very low communication overhead than the “classic” MPI and MPICH versions [10].

B. Framework Used in the Proposed System

This paper deals with the implementation of parallel application such as parallel merge sorting under MPI using MPICH2 for communication between the cores and for the computation. Because it is very much suitable to implement in LINUX systems.

II. RELATED WORKS

Traditionally, multiple processors were provided within a specially designed "parallel computer"; along these lines, Linux now supports SMP Pentium systems in which multiple processors share a single memory and bus interface within a single computer. It is also possible for a group of computers (for example, a group of PCs each running Linux) to be interconnected by a network to form a parallel-processing cluster [8]. Amit Chhabra, Gurvinder Singh [1] proposed Cluster based parallel computing framework which is based on the Master-Slave computing paradigm and it emulates the parallel computing environment. Alaa Ismail El-Nashar [2] used the dual core Window-based platform to study the effect of parallel processes number and also the number of cores on the performance of three MPI parallel implementations for some sorting algorithms. Husain Ullah Khan, Rajesh Tiwari [3] estimated the performance and speedup of parallel merge sort algorithm. An adaptive framework towards analyzing the parallel merge sort is prposed by Husain Ullah Khan, Rajesh Tiwari [4]. Kalim Qureshi [5] presented the practical performance comparison of parallel sorting algorithms on homogeneous network of workstations. Atanas Radenski [6] implemented the shared memory, message passing, and hybrid merge sorts for standalone and clustered SMPs. Manwade K. B [7] conducted analysis of Parallel Merge Sort Algorithm and evaluated the performance of parallel merge sort algorithm on loosely coupled architecture and compared it with theoretical analysis. It has been found that there is no major difference between theoretical performance analysis and the actual result. We aim to present an architecture using MPI that demonstrates the performance gain and losses achieved through parallel processing. And also demonstrates the performance dependency of parallel applications on RAM.

III. SYSTEM REQUIREMENT

A. *Hardware Requirements*

- Processor: Pentium D (3 G Hz)
- Two RAM: 256MB and 1GB
- Hard Disk Free Space: 5 GB
- Network : TCP/IP LAN

B. *Software Requirements*

- Operating System: Linux
- Version: Fedora Core 14
- Compiler: GCC
- Network protocol: Secure Shell
- Communication protocol: MPI

IV. SYSTEM DESIGN

A. *System Analysis*

The system is to designed such that it demonstrates the performance dependency of parallel and serial execution on RAM and also it demonstrates the following:

- How a client can submit the entire problem to a master and collects the solution back from it without bothering about how it has been solved.
- How the master detects the available slaves on the network, and how it detects the system load on that machine to determine whether it is worth sending a task to that particular client.
- How a problem can be submitted to the slaves.
- How the solutions of the given problem can be retrieved from the slave.
- How the slaves solve the given problem.

The design was made modular i.e. the software is logically partitioned into components that perform specific functions and sub-functions.

1) *Master* is designed such that it has functionality to manage connection and communication with the slave, it scans and identifies all the cores or slaves available on the node here it is only one slave to be identified. It then assigns the processor ranks to identify the cores. The master assigns the problem to slave. It also has to accept the results sent back by the slave after they finish the computation of the sub-tasks assigned to them. Then the received result has to be assembled in the right order to obtain the solution for the main problem.

2) *Slave*: This is designed to have the functionality to read the problem (in case of single slave)/sub-problem sent by the master, evaluate the problem (in case of single slave)/sub-problem and send the result back to the master.

B. *Cluster Based Parallel Computing architecture*

The main problem is taken by the master core and assigns the task into slave cores. Each slave core send back the solutions of the assigned sub problem. The working principle involved in this architecture is shown in Fig.1 and Fig.2 shows the cluster based parallel computing architecture.

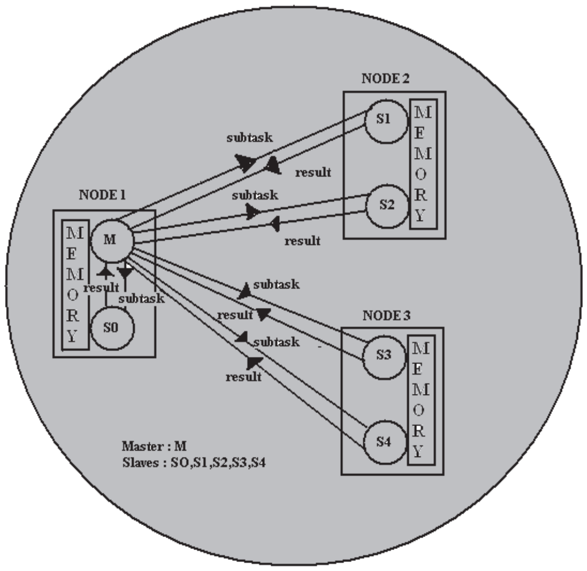


Fig 1. Operations involved in cluster based parallel computing architecture

C. *MPI Configuration*

Download the mpich-2 package and type the following commands in the terminal to install.
Unpack the tar file and go to the top level directory:
tarxzf mpich2-1.3.2.tar.gz
cd mpich2-1.3.2
Configure MPICH2 specifying the installation directory: ./configure --prefix=/home/<USERNAME>/mpich2-install |& tee c.txt
Build MPICH2:make 2>&1 | tee m.txt
Install the MPICH2 commands:
Make install 2>&1 | tee mi.txt
Add the bin subdirectory of the installation directory to your path in your startup script (.bashrc for bash, .cshrc for csh):
PATH=/home/<USERNAME>/mpich2-install/bin:$PATH; export PATH.

D. *SSH Configuration for remote login*

SSH is a program that runs on your personal computer (e.g. PC, Macintosh, or UNIX workstation) and is used to login to a remote computer system. The steps to configure

SSH are as follows. The first thing we'll do is simply connect to a remote machine. This is accomplished by running 'ssh hostname' on your local machine. Here the hostname that you supply as an argument is the hostname of the remote machine that you want to connect to. By default ssh will assume that you want to authenticate as the same user you use on your local machine.The first time around it will ask you if you wish to add the remote host to a list of known hosts, go ahead and say yes.

1) *Generating a Key*: Once you're back to your local computer's command prompt enters the command 'ssh-keygen -t dsa'. It will prompt you for the location of the key file. Unless you have already created a key file in the default location, you can accept the default by pre ssing 'enter'. Next it will ask you for a passphrase and ask you to confirm it.

2) *Installing Public Key*: If you do not have the ssh-copy-id program available, then you must use this manual method for installing your ssh key on the remote host. Copy your public key which is in ~/.ssh/id_dsa.pub to the remote machine using the scp command.

Syntax: scp id_dsa.pub hostname:~/.ssh/authorized_keys.

It will ask you for your system password on the remote machine and after authenticating it will transfer the file. You may have to create the .ssh directory in your home directory on the remote machine first. scp is a file transfer program that uses ssh.

3) *Using the ssh-agent Program*: The true usefulness of using key based authentication comes in the use of the ssh-agent program. you can use the ssh-add program to add your passphrase one time to the agent and the agent will in turn pass this authentication information automatically every time you need to use your passphrase. So the next time you run: ssh username@hostname. You will be logged in automatically without having to enter a passphrase or password. Once you've verified that ssh-agent is running, you can add your ssh key to it by running the ssh-add command: ssh-addIf the program finds the DSA key that you created above, it will prompt you for the passphrase. Once you have done so it should tell you that it has added your identity to the ssh-agent: /home/username/.ssh/id_dsa (/home/username/.ssh/id_dsa).

V. IMPLEMENTATION

Implementation is the most crucial stage in achieving a successful parallel system. The problem to be solved has to be parallelized so that computation time is reduced. The architecture consists of a client, a master core, capable of handling requests from the client, and slave, capable of accepting problems from the master and sending the solution back. The architecture consists of a client, a master core, capable of handling requests from the client, and slave, capable of accepting problems from the master and sending the solution back. The master and the slave communicate with each other using MPICH2 under MPI. The problem has to be divided such that the communication between the master and the slaves is minimum. The total computational time to solve the problem completely is effected by the communication time between the nodes.

A. Parallel Merge Sorting Design

The algorithm which we have implemented is for merge sorting on several nodes, it may be for only one or more slaves. One of the cores is designated as a master and remaining acts as slaves.

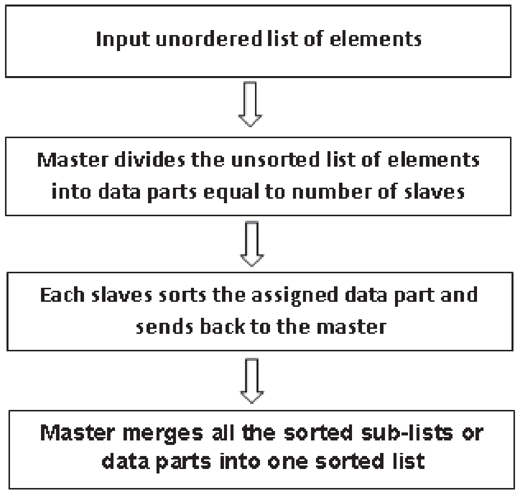

Input unordered list of elements
Master divides the unsorted list of elements into data parts equal to number of slaves
Each slaves sorts the assigned data part and sends back to the master
Master merges all the sorted sub-lists or data parts into one sorted list

Fig.2. Flow diagram for merge sorting on several nodes

The unsorted list of elements is obtained with randomized method. The master divides the unsorted list of elements into the data parts equal to the number of slaves. Then slaves use the sequential version of merge sort to sort their own data. The sorted sub-lists are sent to the master. Finally, the master merges all the sorted sub-lists into one sorted list. Fig.2 shows the flow of operations involved in parallel merge sorting. The outline for the implementation of merge sort is shown in Fig.3. Hence we need to implement parallel systems consisting of set of independent desktop PCs interconnected by fast LAN cooperatively working together as a single integrated computing resource so as to provide higher availability, reliability and scalability. The cluster based parallel computing architecture is as shown in the Fig.4. But to show the performance dependency on RAM we are considering only single node with two cores, one act as master and other as slave. So there will be no division of problem, instead entire unsorted list is submitted to the single available slave.

```
/* Merge Sort */
1. merge_sort(sub-list[], start, last)
2. {   Allocate spaces for "sublist1" and "sub-list2" of size
       (last-start)/2 each;
3.     mid = (first+last)/2;
4.     lcount = mid - first + 1;
5.     ucount = last - mid;
6.     if (last == first) { return};
7.     else {
8.             sub-list1=merge_sort(sub-list[], first, mid);
9.             sublist2=merge_sort(sub-list[], mid+1, last);
10.            merge(sublist1, lcount, sublist2,ucount);
11.        }
12. }
```

Fig.3. Merge sort implementation outline

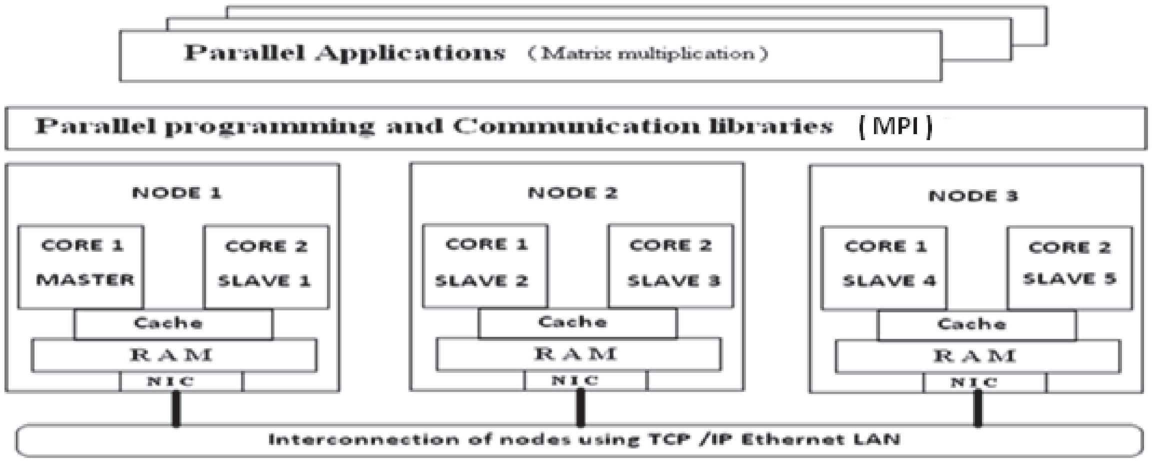

Parallel Applications (Matrix multiplication)
Parallel programming and Communication libraries (MPI)
NODE 1
CORE 1 MASTER
CORE 2 SLAVE 1
Cache
RAM
NIC
NODE 2
CORE 1 SLAVE 2
CORE 2 SLAVE 3
NODE 3
CORE 1 SLAVE 4
CORE 2 SLAVE 5
Interconnection of nodes using TCP /IP Ethernet LAN

Fig 4. Cluster based parallel computing architecture

TABLE I

PERFORMANCE DEPENDENCY ON RAM

Type of Execution	RAM size ↓	10000	20000	300000	4000000	5000000
Serial	256 MB	0.00062 seconds	0.00091 seconds	0.00264 seconds	3.22311 seconds	5.56543 seconds
	1000 MB	0.00061 seconds	0.00093 seconds	0.00261 seconds	3.21631 seconds	5.58412 seconds
Parallel using MPI	256 MB	0.25621 seconds	5.42741 seconds	14.0342 seconds	40.74265 3seconds	126.27217 seconds
	1000 MB	0.11321 seconds	0.44631 seconds	1.78462 seconds	6.64854 seconds	10.06431 seconds

TABLE II

PERFORMANCE FOR SMALLER UNSORTED LIST OF ELEMENTS

Type Of Execution	Number of nodes ↓	10000	20000	30000	40000	50000
Parallel using MPI at 256 MB RAM	Two	0.13521 seconds	1.57364 seconds	2.39516 seconds	3.23421 seconds	4.91722 seconds
	Three	0.14136 seconds	1.91452 seconds	2.98153 seconds	3.71349 seconds	5.54262 seconds

TABLE III

PERFORMANCE FOR LARGER UNSORTED LIST OF ELEMENTS

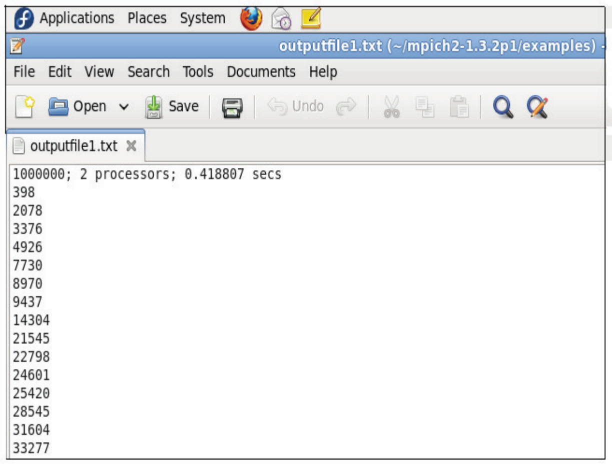
Type of execution	Number of nodes ↓	1000000	2000000	3000000	4000000	5000000
Parallel with MPI at 256 MB RAM	Two	1.05426 seconds	1.84810 seconds	2.31324 seconds	2.81328 seconds	4.71486 seconds
	Three	0.78216 seconds	1.64632 seconds	2..04327 seconds	2.17449 seconds	3.81042 seconds

VI. RESULTS AND ANALYSIS

We have analyzed the performance of parallel method against traditional serial method. The results are tabulated and compared. We calculated the time for sorting the unordered list of elements using both serial merge sorting and a case of parallel merge algorithm using MPI. Analysis can be made under four different cases.

Case 1: Single node analysis to show performance dependency on RAM. Table I shows that performance of serial execution almost remains same even after the increase in RAM size. There are negligible computation time variations for increase in RAM size. This is because the Serial execution is performed by the cores itself with negligible RAM usage and also due to the no communication involved between cores. Hence it is independent of RAM. We can also conclude that performance of parallel execution increases when there is increase in RAM size. It shows drastic decrease in computation time with the increase in RAM. Because parallel execution often uses RAM for the communication between cores and also it involves lot of send and receive operations and temporarily storing the result of problem assigned to cores. We can analyze that higher the size of unordered list the time difference is very high in the table, because higher the input size, more will be sends and receives resulting in the need of higher utilization of RAM. So for smaller RAM the computation time will be more and larger the RAM size computation time will be less in parallel execution finally resulting in better performance. However by seeing the time results for larger inputs such as 4000000 and 5000000, there is a large reduction in computation time when there is increase in RAM size.

Case 2: Single node analysis to show communication overhead involved in parallel computations. Generally we can say parallel execution is faster than the serial execution but the results of serial execution with 1000MB RAM and parallel execution using MPI with 1000MB RAM shown in the Table I depicts that serial execution is faster than parallel execution in a single node having two cores, for different sizes of matrices.


Applications Places System
outputfile1.txt (~/mpich2-1.3.2p1/examples)
File Edit View Search Tools Documents Help
Open Save Undo
outputfile1.txt

```
1000000; 2 processors; 0.418807 secs
398
2078
3376
4926
7730
8970
9437
14304
21545
22798
24601
25420
28545
31604
33277
```

Fig.5. Result of merge sorting the unsorted list containing 1000000 elements

This is due to the communication overhead involved in the parallel execution but this can be overcome by increasing the number of nodes. Overheads that are considered are the connection time required to connect to slave, time taken to send the problem along with inputs to slave time taken to retrieve the solutions from the client, time taken to assimilate the results obtained. Fig.5 shows the example of the result obtained for parallel merge sorting.

Case 3: Multiple nodes analysis with smaller unsorted list of elements (computation time < communication time). Table II shows the time taken to solve the problem wholly is more when the number of nodes is more for smaller unordered list of elements. Because the problem has to be communicated among all the slave cores hence the communication time is larger than the computation time. So the 2 nodes can compute it and assemble it faster than a 3 node or a 4 node system.

Case 4: Multiple nodes analysis with larger unsorted list of elements (computation time > communication time). Table III shows that for larger unordered list of elements, the performance of the system increases phenomenally with increase in number of nodes. As the size of the input increases the computation time also increases. The computation time is so large that the communication time is negligible compared to it.

VII. CONCLUSIONS

We presented a model that demonstrates the performance gain and losses achieved through parallel processing. We also presented a model that demonstrated the evaluation of the performance dependency of parallel MPI based applications and its serial version on RAM showing Serial computation involves negligible RAM and parallel computation utilizes more RAM especially for larger inputs. Serial execution is faster for smaller input size because of the communication and connection overheads in parallel execution. The performance of parallel execution is far greater compared to serial execution when the size of the input is very large. The total time taken to compute the result decreases drastically when the number of nodes increases.

VIII. FUTURE WORKS

Even though the method that has been used here can be deployed to solve larger order problems, it is cumbersome to give the data input for the larger unordered list of elements. Hence this work can be extended to give input from files for larger list of unsorted elements. It can also be extended to solve other similar problems such as matrix parallel multiplication, finding the determinant and other backtracking problems. The analysis is also useful for making a proper recommendation to select the best algorithm related to a particular parallel application. If the nodes are extended, node failure can be a problem that has to be tackled.

REFERENCES

[1] Amit Chhabra, Gurvinder Singh "A Cluster Based Parallel Computing Framework (CBPCF) for Performance Evaluation of ParallelApplications", International Journals of Computer Theory and Engineering, Vol. 2, No. 2 April, 2010.

[2] Alaa Ismail El-Nashar, “Parallel Performance of MPI Sorting Algorithms on Dual Core Processor Windows-Based Systems”, International Journal of Distributed and Parallel Systems (IJDPS) Vol.2, No.3, May 2011.

[3] Husain Ullah Khan, Rajesh Tiwari, ”An Adaptive Environment To Evaluate The Performance Of Parallel Merge Sort”, International Journal of Engineering Research & Technology (IJERT), Vol. 1 Issue 8, October – 2012.

[4] Husain Ullah Khan, Rajesh Tiwari, “An Adaptive Framework towards Analyzing the Parallel Merge Sort”, International Journal of Science and Research (IJSR), Volume 1 Issue 2, November 2012.

[5] Kalim Qureshi, “A Practical Performance Comparison of Parallel Sorting Algorithms on Homogeneous Network of Workstations”, TELE-

INFO'06 Proceedings of the 5th WSEAS international conference on Telecommunications and informatics, Pages 276-280, 2006.

[6] Atanas Radenski, “Shared Memory, Message Passing, and Hybrid Merge Sorts for Standalone and Clustered SMPs”, The 2011 Interanational Conference on Parallel and Distributed Processing Techniques and Applications,pp. 367-373, 2011.

[7] Manwade K. B, “Analysis of Parallel Merge Sort Algorithm”, 2010 International Journal of Computer Applications (0975 - 8887) Volume 1 – No. 19 66, 2010.

[8] A. Nazir, H. Liu, and S.-A. Sørensen, “On-demand resource allocation policies for computational steering support in grids, ” in International Conference on High Performance Computing, Network and Communication Systems, Orlando, USA, 2007.

[9] Message Passing Interface, MPI Standard: http://www.mpi-form.org.

[10] MPICH2: A New Start for MPI Implementations, Recent Advances in Parallel Virtual Machine and Message Passing Interface, Lecture Notes in Computer Science, Volume 2474, 2002, p 7.