International Journal of Computer Applications (0975 8887)
Volume 106 - No. 12, November 2014

A Dynamic Scheduling Algorithm for Spawn Processes in MPI-2 to Improve and Maintain Load Balancing

Sherif AbdElazim Embaby
Department of Computer Science
Faculty of Computers and Information
Cairo University, Egypt

Ahmed Shawky Moussa
Department of Computer Science
Faculty of Computers and Information
Cairo University, Egypt

Ibrahim Farag
Department of Computer Science
Faculty of Computers and Information
Cairo University, Egypt

ABSTRACT

Some parallel applications that solve big problems in fields like weather forecasting, data analysis, energy fields, and protein folding need to create unpredicted processes at the application run time. The MPI package provides capability to not only write static parallel programs but also to create dynamic processes at run time. However, the MPI standard did not provide any way to schedule these dynamically created processes. Online scheduling can be a solution for this problem. Hence, this work introduces an online scheduling algorithm for dynamically created processes on cluster's nodes according to the machines performance. The objective of the algorithm is to achieve dynamic load balancing along the scheduler run time over heterogeneous cluster's machines. In addition, the proposed scheduler achieves load balancing over heterogeneous hardware according to the current real-time state of the nodes even if the processors are responding to other parallel simultaneous schedulers or running other parallel or sequential programs.

Keywords:

Spawn processes, Dynamic scheduling, Parallel processing, MPI

1. INTRODUCTION

Although MPI 1.2 has succeeded in adding new functionalities to the previous version of the Massage Passing Interface (MPI), it still missed important features which posed limitations on using the MPI packages for many real world parallel computing applications. A major example of these limitations is the inability of many applications to create processes at the applications run time. Therefore, all the applications that cannot anticipate the number of processes needed to complete its work could not be written in MPI 1.2. As a result, these limitations have made a significant weakness in the MPI package. This motivated the MPI developing team to work on recovering from this critical shortcoming by adding a new feature in the next version of MPI [1][2].

The MPI development team succeeded in adding the sought new feature that creates processes during the applications run time in the new version called MPI-2. This new version provides an interface that allows creating processes during the execution of MPI programs namely `MPI_Comm_spawn,` which in turn allows the spawned processes to communicate by message passing between each other, and between each spawned process and its parent. Although MPI-2 was improved by various new other features, this paper focuses on the dynamic creation and management of processes. Thus, the current research concentrates on the `MPI_Comm_spawn` function which will be detailed because it plays a vital role in the introduced work.

The `MPI_Comm_spawn` function implements the dynamic process creation as introduced before. This function provides the creation of new processes after the MPI program is launched. In order to spawn a new process, this function needs some parameters to be passed:

(1) The1st is the [*file executable name*] this parameter should be compiled as MPI program.
(2) The 2nd parameter is the arguments of the executable file at the first parameter.
(3) The 3rd parameter specifies the number of the newly created processes. Each process represents a separate program instance from the 1st parameter.
(4) The 4th parameter is [*mpi info*] which represents the host name on which the process will be created.
(5) The 5th parameter represents the rank of the processes in which previous arguments are examined.
(6) The 6th parameter represents a communicator for which the newly created (spawned) processes belong and consequently can communicate through MPI messages.
(7) The 7th parameter represents a communicator between the spawned process(es) and the parent so that spawned processes may communicate with parent through classical MPI messages.

MPICH-II is considered to be the most widely used package that implements dynamic process creation [3]. Besides, since MPICH is an open source package, the researchers prefer to use it rather than many other commercial packages that implement MPI. By using MPICH, the researchers can learn how MPI is implemented, and those researchers can also improve this implementation.

MPICH runs on the Linux operating system by using Multi-Purpose Daemon (MPD), the process manager that is able to launch the execution of a parallel program on multiple machines. Additionally, it provides communication between all launched processes to exchange data [4][5]. The MPD starts on all the nodes participating in the cluster. Using mpdboot, MPD launches itself on all nodes of the cluster. The MPD clusters configuration has a ring topology, where each node is connected to the next node, and the last node is connected to the first one. When a parallel program calls MPI_Comm_spawn, without specifying where to run the newly spawned process through the [*mpi info*] parameter, the process manger MPD starts the new spawned process(es) on the next node in the MPD's ring. For the newly created processes, the default is that these newly created processes follow Round Robin algorithm starting from rank(0) every time [6]. So, the creation of processes by using MPI_Comm_spawn has two scenarios: The first happens when using one MPI_Comm_spawn function to create many processes by using the [*Maxproc*] parameter. It will follow the Round Robin algorithm to distribute processes starting from rank(0), and this will distribute the load equally [2][7]. Conversely, the second scenario suffers from a significant problem when the MPI_Comm_spawn function is called many times. For example, when calling the function iteratively inside a loop, or when making recursive calls for this function. In both cases, each call of this function will submit the new process every time to the first rank. In this case, all the created processes will be submitted to one node, the first machine, as shown in Table 1. As a result, the first machine will have all the load, whereas no tasks will be submitted to the other machines [6].

Table 1. process distribution on machines using mpi primitive spawn function

Spawned processes	Station1	Station2	Station3	Station4	Station5
25	25	0	0	0	0
50	50	0	0	0	0
75	75	0	0	0	0
100	100	0	0	0	0

As stated above, the decision of submitting new tasks to hosts is made in the function level. In every call to this function, the submission of tasks happens according to the Round Robin algorithm starting from the clusters first host independent of any MPI_Comm_spawn previously called. This unbalanced load happens because of the decentralization in decision making that MPI_Comm_spawn uses in submitting the new spawned process to a host since MPI-2 does not introduce any way of scheduling [6]. To overcome this problem, the authors of the current paper propose a centralized decision making and scheduling approach for the newly created dynamic processes. A different idea of the same approach was introduced by Cera et al [6] which will be discussed in the next section.

The rest of the paper is organized as follows: Section (2), the Literature Survey, reviews previous approaches for solving the problem of scheduling spawn processes in MPI. In Section (3), the Proposed Scheduler, the authors present the main contribution of the current paper, which is a new scheduling technique for dynamic processes aiming at achieving load balancing. Section (4) covers the evaluation and results analysis of the experimentation. Finally, Section (5) draws the conclusions and lays out the future plans of continuation of the current research.

2. LITERATURE SURVEY

There are many parallel applications described in the literature as statically load balanced where a fixed amount of resources is allocated a priori for the lifetime of every process. Load balancing is understandably easier to achieve in such cases. Gang scheduling [8] is one of the algorithms that can be used efficiently in the static load balancing. It arranges processes or threads in rows and processors in columns to form a matrix. Accordingly, each row in the matrix represents a time slice since columns represent processors. During the execution time, context switching occurs at the same time through all machine processors to execute processes of the next row, according to their corresponding nodes. The Gange scheduling is designed to let processes in the same row communicate and exchange data during their execution. The drawback of Gang scheduling is its waste of time and resources. Processes in the same raw take different time durations to finish their jobs. Hence, some hosts will be idle for a while until all processes in the same row finish their work.

As for dynamic load balancing, it can be rather easier to achieve when there is a large number of independent tasks in the parallel application. Since this application is implemented with server and computing nodes, in case a node becomes slow, the application can shift work to other nodes. For load balancing jobs dynamically, Gang-scheduling will not be a suitable choice [9]. Processes on the same raw will take equal time slots although, in practice, these processes may differ in execution time. In such case, resources allocated for the processes with shorter execution time will remain idle until the current time slice ends, wasting computing time. For programs with such loosely coupled processes, load balancing can be better achieved by other methods such as Loadleveler [10], Condor [11], or LSF [12][13]. However, in the case of tightly coupled application processes which need to communicate during their running to exchange data, Gang scheduling can be used regardless of the performance issues due to unequal execution times.

For more dynamism, resources utilization optimization, and better performance, Gropp and Lusk introduced a runtime environment architecture for parallel applications to enable portability of parallel applications over different environments while optimizing the usage of environment resources [13]. Gropp and Lusk built their architecture based on client-server model that interacts with newly created processes in any environment so that Gropp and Lusk do not need to make changes in MPI primitive functions.

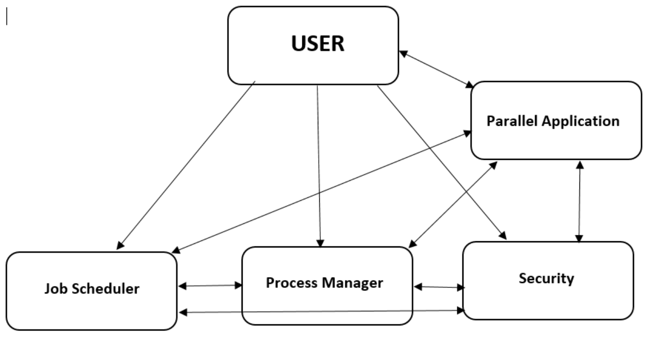

Fig. 1. Parallel runtime environment architecture

Gropp and Lusk realized that parallel environments have distributed resources besides the need to distribute processes equally over resources. Hence, Gropp and Lusk decided to centralize the information about both resources and processes' locations. Accordingly, Gropp and Lusk designed general architecture not related to specific parallel library. This architecture is composed of three main parts: job scheduler, process manager, and security as illustrated in Figure 1.

`Job scheduler:` It is responsible for the time the process will be launched and the processor that will run it. So job scheduler is considered to be as a queuing system [4].

`Process manager:` It takes management from the point of handling *stdin*, *stdout* and *stderr* by guaranteeing the delivery of their signals and handling them in a reasonable way.

`Security:` It handles the security from the point of guaranteeing that the scheduler allocates resources of users programs correctly.

Gropp and Lusk designed this architecture regardless of its implementation or the parallel library used. Besides, it did not introduce a specific algorithm to deal with spawn processes. The MPI package can be taken as a case study or testbed to implement the architecture of Gropp and Lusk. MPI-2 has a primitive `MPI_Comm_spawn` function that creates processes at the application run time. This primitive has a noticeable problem in dynamic process creation since it cannot achieve balancing among machines. Consequently, some kind of load distribution should be achieved. Cera, Pezzi, Mathias, Maillard, and Navaux [2] suggested to centralize the allocation decision making of distributing the load between all available machines. Thus, Cera, Pezzi, Mathias, Maillard, and Navaux followed Gropp and Lusk steps and designed architecture composed of centralized job scheduler and resource manager which deal in parallel with `MPI_Comm_spwan` functions that are included in a parallel application.

The Round Robin scheduler of Cera et. al. must be running in parallel as a daemon besides any parallel application that spawns many processes at its run time. In addition, this scheduler will be more efficient when it allocates resources in a good way beside a short overall run time of the parallel application [2]. Cera et. al. built their solution based on two main parts: a scheduler and a job manager. The job manager keeps track of all jobs in the system maintaining the information needed for the scheduler. The scheduler is mainly a daemon running beside the parallel spawning application that uses `MPI_Comm_spawn` function with some modifications, that enable communication between a daemon and the running application. The daemon assigns hosts for the running parallel application calling `MPI_Comm_spawn` according to the Round Robin algorithm, in order to achieve distribution balance between all hosts. The scheduler depends on the job manager which records all distributed tasks to determine the next host for the new spawned process according to Round Robin algorithm to achieve equal distribution for the newly created processes between all hosts [2].

For verification and testing, the research team of the current paper reimplemented this Round Robin scheduler of Cera et. al. and applied some test cases using a matrices computation program. The program spawns many times by a specific given number that is passed as a parameter at run time. The matrices are also created dynamically by passing the number of rows and columns also as parameters. The authors of the current paper verified and confirmed that this Round Robin scheduler of Cera et. al. distributes the load according to the given list of machines on the test cluster. The scheduler distributes the total number of processes equally between the given machines. When the number of processes is not divisible by the number of machines in the cluster it distributed the remaining processes on the machines from the beginning of the scheduler's list of machines according to their order in this list. This is simply because `MPI_Comm_spawn` assigns process on a specific machine (host) not a specific core by using [*mpi info*] parameter to assign the machine name to this parameter [7]. The authors carried out the testing experimentation on the HiPer-FC cluster at the Faculty of Computers and Information - Cairo University, which is a heterogeneous cluster running Scientific Linux operating system. Station1 has a quad core processor with 4GB RAM. Station2 to station5 are dual core processor machines with 2GB RAM. According to this Round Robin scheduler of Cera et. al. station1 being quad machine takes double the load of any of the other four machines. The rest of processes will be distributed on the top of the machine list. In this specific case, the remainder of these processes will be loaded on station1. Table 2 summarizes the experimentation results of applying the Round Robin scheduler of Cera et. al. [2] to distribute the spawned processes.

Table 2. process distribution on machines using mpi modified spawn function with Round Robin scheduler of Cera et. al.

Spawned processes	Station1	Station2	Station3	Station4	Station5
25	9	4	4	4	4
50	18	8	8	8	8
75	27	12	12	12	12
100	36	16	16	16	16

Figure 2 illustrates the comparison in time performance between the native MPI scheduling and the Round Robin scheduler of Cera et. al. It is clear from the figure and the underlying experimentation that Cera, Pezzi, Mathias, Maillard, and Navaux achieved two main benefits which become clearer with the growth of the number of spawned processes: (1) significant performance enhancement demonstrated by the growing difference in response time, and (2) the new scheduler enabled spawning larger number of processes as shown in Figure 2.

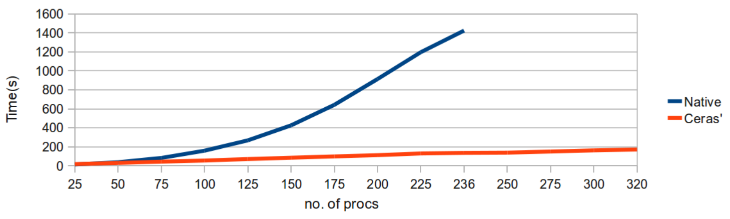

Fig. 2. Comparison between MPI native scheduling and Round Robin scheduler of Cera et. al.

3. THE PROPOSED SCHEDULER

It was concluded from the previous sections that the default dynamic process manager in MPI-2 is the MPD process manger, which starts the new spawned process(es) on the first next node in the MPD ring [14]. This results in a problem in balancing the load when `MPI_Comm_spawn` primitive function is called many times such as the cases when using it inside a loop or making recursive calls for a spawn function. According to the MPD native scheduling policy, every call to the spawn function will submit the new task every time starting from the first rank [14][6]. This imbalance is illustrated by Figure 3. As a consequence, and as shown in Figure

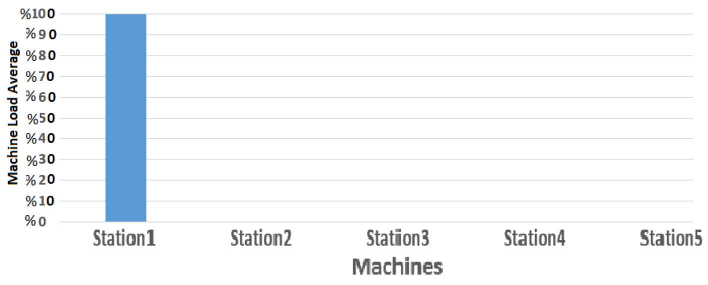

Fig. 3. Iterative MPI Comm spawn function call with MaxProc=1.

3, all the created processes will be admitted to one machine. This will defy the load balancing concept as the system will end up with one overloaded machine even though it may have many other idle machines. This problem is caused mainly by the decentralization of the decision to determine the host when using the MPI spawn primitive. Therefore, each new MPI spawn call decides the machine that will spawn on regardless the other previous MPI spawn call decisions and not knowing anything about the status of the other machines.

The embedded Round Robin scheduler of Cera et. al. [2][6] appeared to centralize the decision making by running the scheduler as a daemon beside the running application to make the scheduler connect to the embedded scheduler every time the application needs to spawn a new process. The scheduler then decides the next host machine according to the Round Robin algorithm. This was the approach Cera et. al. used to achieve the load balancing concept over all the system's machines equally, overcoming the overload problem on one machine when calling the primitive MPI spawn iteratively or recursively.

Despite the improvement gained in both performance and load balancing by using The embedded Round Robin scheduler of Cera et. al., the current research team noted another problem when the cluster's machines are not dedicated to one application. Some machines of the cluster get higher load than others because of running other programs on some machines of the cluster while using the embedded Round Robin scheduler. It distributes the processes of this application equally over all cluster machines regardless if some machines are preloaded by other running applications. This impairs the targeted load balancing on all machines intended by the scheduler of Cera et. al. [2][6]. Furthermore, sometimes the machine's overload may hit the limitations of the machine's resources. This may lead the overloaded machines to crash, even though other machines in the system may have some idle processes. Figure 4 illustrates a situation when various nodes on the system have different levels of resources availability to start with when the target parallel program is launched.

When the load is distributed equally on the nodes according to the embedded Round Robin mechanism of Cera et. al., the end result will be load imbalance and in some cases cuasing overloads hitting the resources limitations which may lead to machines crashing as shown in Figure 5.

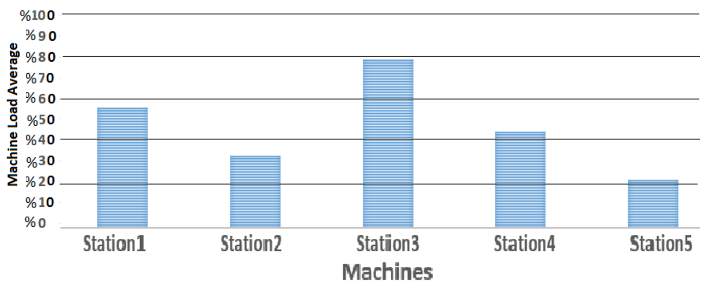

Fig. 4. Independed individual processes running on cluster's machines.

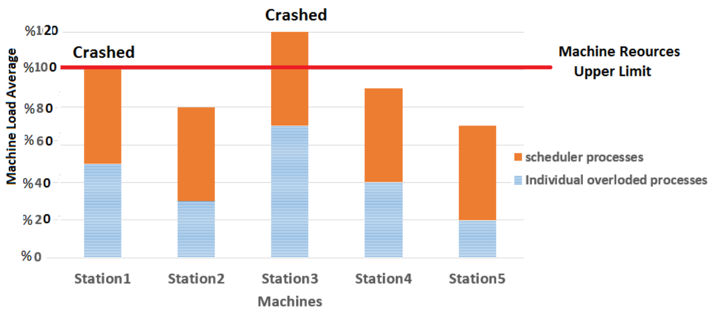

Fig. 5. Round Robin scheduler of Cera et. al. processes running besides independed individual processes.

In Figure 5 above, some machines hit or exceed the resources limits, such as Stations 1 and 3 in the Figure, while others, Stations 2, 4, and 5 have different degrees of availability to run more tasks. Designing a mechanism to enable the redistribution of tasks not necessarily equally but considering the actual status of the node would lead to true load balancing and crash avoidance. The crash to be avoided can be either a machine crash due to hitting limitations or a whole system crash due to a crashed station not returning results to parents.

This imbalance imposed by the heterogeneity of the system or status of the nodes running other unfinished jobs motivated the current research team to consider the design and development of a solution that takes the load status of the cluster's machines into consideration even if the load is attributed to other running programs or external factors. This should lead to better true load balance and crash avoidance. It will also leads to more reliable programs and higher performance because the probability that some machines may crash due to hitting or exceeding resource limitations resulting in the prevention of returning results from the processes on those machines to their parent processes, or to the scheduler, will be much lower.

The underlying task of estimating the load on each machine in the system in order to device a load balancing scheduler necessitates the study of how the `MPI_Comm_spawn` assigns its new tasks. `MPI_Comm_spawn` creates a new process and by using the [*mpi info*] parameter in this function the programmer can assign it to a specific machine. Accordingly, this function assigns the newly created process to a specific machine, not on a specific core [14]. At the machine level, the newly submitted processes will be scheduled according to the operating system's scheduler. Hence, to achieve both the load balancing and the optimization in using resources, the authors of the current paper proposed to assign a number of spawned processes on each machine equal to the number of cores in each machine and the tasks of which the machine was already preloaded from other running programs. Since the cluster's machines are heterogeneous, the load of each machine can be estimated by recording the machine's completion time of each assigning task. According to each completion time, a number is given to the machine that represents the priority of using this machine when scheduling new tasks. Therefore, according to the proposed solution, the time is measured for each spawned process starting from the assigning time on a machine until the responding time has arrived to the spawner again. When a machine is needed to be choosen for a new spawned process, the generated list of machines is sorted according to each machine priority then the best priority machine is selected.

Algorithm 1: The Priority scheduler algorithm

```
1.  Scheduler read hostsname.txt and specifications from host file.
2.  Scheduler initialize the table with machines name and
    specifications.
3.  if machine specifications equals another
4.  then
5.  first machine priority = second machine priority
6.  end if
7.  Scheduler launch the main parallel application.
8.  Parallel application send request to scheduler to spawn a
    process.
9.  if highest machine priority = another machine priority
10. then
11. get the first according its order in scheduler table
12. end if
13. Send machine name to the parallel application.
14. Parallel application spawn new process on the received
    hostname.
15. Scheduler set machine timer to zero and start counting once
    the parallel application call spawn function
16. Spawned process send to the scheduler when it is ended
17. Scheduler stops the timer.
18. Scheduler set machine priority according to its completion
    time.
19. Scheduler update itself by new machine priority of the
    spawned host name.
20. Scheduler terminated when all spawned processes ended up.
```

The experimentation with the proposed priority based scheduler revealed clearly that it did not assign the same number of tasks to each machine every time. This is because assigning processes to a machine depends on the load status of the machine every time a process is spawned. As a result, unlike the embedded Round Robin scheduler which assigns the same number of processes to each machine, it differs every time according to the machine's load status. After that the performance of the proposed priority based scheduler can be evaluated by measuring its completion time when scheduling different numbers of spawned processes. This is to assess the effect on performance as a result of achieving the load balancing between all cluster's machines.

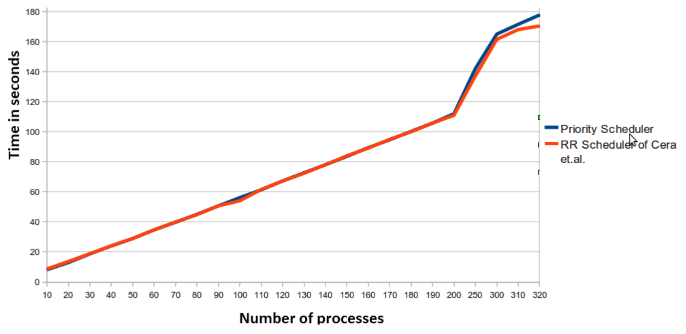

Fig. 6. RR and Priority schedulers without any additional load in hosts.

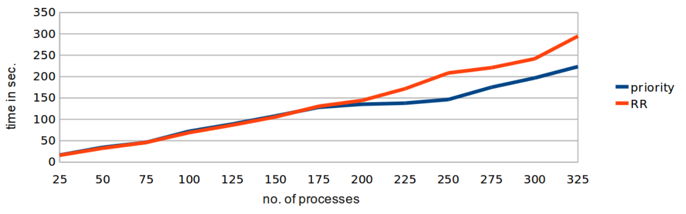

Fig. 7. RR and Priority schedulers with two programs loads.

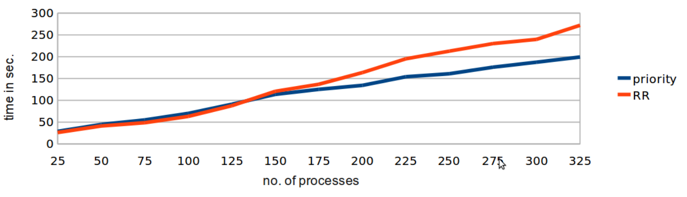

Fig. 8. RR and Priority schedulers with 6 programs loads.

4. EVALUATION AND RESULT ANALYSIS

The results of the performance comparison between the proposed priority based scheduler and the embedded Round Robin scheduler are summarized and illustrated by figures 6, 7, and 8. It is obvious from Figure 6 that the Priority Scheduler takes more time than the Embedded Round Robin Scheduler. This is mainly due to the fact that the Round Robin decision of determining in machine order is based on formula (1) [6].

$$new\_resource = (last\_resource + 1)\%total\_resources[5] \quad (1)$$

Equation (1) gives the formula for sample variance.
On the other side, the Priority Scheduler decision making depends on searching for the best priority. This is the main reason the results show that Embedded Round Robin scheduler has results better than Priority Scheduler due to the searching overhead in the absence of overload. However, when comparing between the results of Embedded Round Robin Scheduler and Priority Scheduler in case an overload is made on some cluster's machines by running external applications, the Embedded Round Robin scheduler started equal or slightly better because all machines, even the overloaded ones, still have idle processes. Hence, no need for searching for machines, which appeared in the first part of Figure 7 until 175 submitted processes. The advantage of Priority scheduler appeared when the overloaded machines reached or almost reached

its peak number of processes, while the other machines still have some idle processes. In which case, the Embedded Round Robin scheduler will submit tasks on those overloaded machines that do not have any idle processes according to the Round Robin mechanism[6]. This leads to these spawned processes either taking long times to be completed which leads to the overall application completion time delay, or the machine will be crashed because of submitting a huge number of processes that exceed its resources capacity. In contradiction, the Priority Scheduler will avoid these overloaded machines when the scheduler notices, these overloaded machines take more time to respond after processing the spawned processes. This means that either the machine is running too many processes or there is a problem with the network connection that leads to this delay. In both cases the proposed Priority Scheduler avoids this machine by assigning it a low priority in order to make it the last choice. Accordingly, the results returned from such machine will not delay the overall completion time of the application. This makes the priority scheduler proposed in this paper better and more efficient. The time differences of small spawned processes is usually small and can be ignored. This makes Priority Scheduler better in average and worst cases that cannot be predicted in a heterogeneous system. Experimentation showed that it also worked comparatively well in best cases too as shown in Figure 6, which proves overall significant enhancement over the embedded round robin scheduler.

5. REFERENCES

[1] "mpi-2: Extensions to the message passing interface" message passing interface forum, 1997. http://www.mpi-forum.org.

[2] Márcia C Cera, Guilherme P Pezzi, Elton N Mathias, Nicolas Maillard, and Philippe O a Navaux. Improving the dynamic creation of processes in MPI-2. *Lecture Notes in Computer Science*, 4192/2006:247–255, 2006.

[3] MPICH Organization. http://www.mpich.org/.

[4] Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, Jayesh Krishna, Ewing Lusk, and Rajeev Thakur. PMI : A Scalable Parallel Process-Management Interface for Extreme-Scale Systems.

[5] William Gropp, David Ashton, Darius Buntinas, and Brian Toonen. MPICH2 Installers Guide. 2009.

[6] Márcia C Cera, Guilherme P Pezzi, Nicolas B Maillard, and Philippe O A Navaux. Scheduling Dynamically Spawned Processes in MPI-2.

[7] Edgar Gabriel, Graham E Fagg, and Jack J Dongarra. Evaluating the Performance of MPI-2 Dynamic Communicators and One-Sided Communication. pages 88–97, 2003.

[8] Peter Brucker and P Brucker. *Scheduling algorithms*, volume 3. Springer, 2007.

[9] William Saphir, Leigh Ann Tanner, and Bernard Traversat. Job Management Requirements for NAS Parallel Systems and Clusters. *NAS Technical Report NAS-95-006*, pages 1–19, 1995.

[10] Subramanian Kannan, Mark Roberts, Peter Mayes, Dave Brelsford, and Joseph Skovira. Workload Management with LoadLeveler. *IBM Corporation, International Technical Support Organization*, 2001.

[11] Condor Team. Condor® version 6.8. 2 manual. 2006.

[12] Songnian Zhou, Xiaohu Zheng, Jingwen Wang, and Pierre Delisle. Utopia: a load sharing facility for large, heterogeneous distributed computer systems. *Software: Practice and Experience*, 23(12):1305–1336, 1993.

[13] W Gropp and E Lusk. Dynamic process management in an MPI setting. *ProceedingsSeventh IEEE Symposium on Parallel and Distributed Processing*, pages 530–533, 1995.

[14] Message Passing and Interface Forum. MPI : A Message-Passing Interface Standard. *Forum American Bar Association*, 8(UT-CS-94-230):647, 2009.