DYNAMIC LOAD BALANCING IN HETEROGENEOUS CLUSTERS: EXPLOITATION OF THE PROCESSING POWER

MOHAMMED ALDASHT *
JULIO ORTEGA **
CARLOS G. PUNTONET **

* Department of Information Technology
Palestine Polytechnic University,
P.O.Box 198 Hebron, Palestine
{mohammed@ppu.edu}

** Department of Computer Architecture & Technology
University of Granada,
E-18071 Granada, Spain
{julio, carlos @atc.ugr.es}

ABSTRACT

The dynamic load balancing techniques, practically, do not assume any information about the tasks to be executed at compilation time. Parameters like execution time or communication time are unknown at compilation time. These techniques are used to distribute the computation tasks of an application between different processors at execution time to achieve some defined performance objectives [1]. In this paper we present a dynamic load balancing algorithm designed especially for heterogeneous network of workstations. The algorithm distributes the parallel tasks dynamically attempting to minimize its execution time. The experiments are done over a network of workstation interconnected via a fast Ethernet. It is a Linux cluster which has some degree of heterogeneity in the processing nodes. Our algorithm is shown to be efficient in increasing the resource utilization and reducing the total execution time of the applications.

Key words: *Heterogeneous Cluster, Dynamic Load Balancing, Centralized Algorithms***.**

1. INTRODUCTION

Load balancing means to distribute the workload of a parallel application among the processors in the platform at hand according to their relative performance, in order to minimize the execution time of the program [6]. Load balancing algorithms can be classified into three main classes: static algorithms, dynamic algorithms, and adaptive algorithms [4]. Static algorithms decide how to distribute the workload according a prior knowledge of the problem and the system characteristics. Dynamic algorithms use state information to make decisions during program execution. Finally, Adaptive algorithms are a special case of dynamic algorithms. They dynamically change its parameters in order to adapt its behavior to the load balancing requirements. Moreover, dynamic load balancing strategies can be divided basically into two main classes: *centralized* dynamic load balancing and *distributed* dynamic load balancing [3]. These strategies define where the load balancing decisions are made. In a centralized scheme, the load balancer is implemented on one master processor and all decisions are made there. In a distributed scheme, the load balancer is replicated on all processors [5]

The dynamic load balancing exploits the communication resources of the parallel platform to exchange state information and tasks between the processors. Therefore, the processors use the local information which they have about the global state of the system, to make decisions that allow obtaining of minimal response time and maximum performance. The efficiency of a load balancing algorithm depends on: the communication cost between processors, the complexity associated with the decision making procedure in each processor, and on the cost of maintaining relative information of the global state of the system in each of the nodes [2,3].

2. OPERATIONS AND PARAMETERS OF A DYNAMIC LOAD DISTRIBUTION ALGORITHM:

Dynamic scheduling is based on the redistribution of processes between processors during the execution time. This redistribution is achieved by transferring tasks from overloaded processors to the under-loaded ones, this operation is called load balancing, and it is done with the objective to improve the performance of the application execution[2,3].

Typically, a load balancing algorithm can be defined by a series of parameters:

- Granularity: it describes the degree of partitioning the application into sub-tasks. Depending on the number of sub-tasks, partitioning of a problem could be fine-grain, medium-grain, coarse-grain or random.
- Initial work distribution: assuming that the load is entering a load queue, initially we can distribute *equal* quantity of load to every processor. A *staggered* distribution can be a good choice if we don't have much information about the node resources. If we have some information about the node resources we can give each node a quantity of load proportional to *its processing power.* Another choice is to achieve a *random* initial distribution.
- Information policy: according to which the processing nodes exchange load information. It can be periodical, on-demand, or on-state. In our algorithm we use the on-demand information policy as processors send to the central node their state information when they ask for new tasks to execute.
- Transfer policy: determines whether a node is in a suitable state to participate in a task transfer. It can be either sender-driven or receiver-driven. With a sender-driven policy the node is letting partner nodes know that it has tasks to be transferred. With a receiver-driven policy the node has extra resources and is ready to accept more tasks from partner nodes. Similar to information policy, this can be done periodically or be threshold-driven. Our algorithm is centralized and the transfer policy used is receiver-driven policy which convenes with the on-demand information policy.
- Location policy: is about finding a suitable transfer partner using information about the node status. A location policy can try to find the most suitable node, which would involve more overhead computation and time delay, or settle with an adequate node, which may not give the best result.
- Selection policy: determines which tasks to be transferred. It can be pre-emptive or non-preemptive. A pre-emptive transfer involves tasks that are partially executed, while a non-preemptive policy only involves tasks that have not begun execution. In our algorithm we have used the non-preemptive selection policy, so the transferred tasks are from the task queue of the master.

3. DESCRIPTION THE ALGORITHM

In this work we propose a dynamic load distribution algorithm by which we try to obtain the maximum processor utilization and exploitation of the processing power in a heterogeneous parallel processing system.

The proposed algorithm is denominated as Exploitation of the Fastest Processor (EFP) which is a centralized algorithm for dynamic load distribution of parallel applications based on the Single Program Multiple Data (SPMD) paradigm. EFP algorithm tends to obtain the maximum utilization and exploitation of the processing power in a heterogeneous parallel system, through distributing the parallel tasks dynamically in the way that permits the execution of the parallel application in the minimum possible time.

Figure 1 shows the block diagram of the multi-computer system architecture. The centralized approach is based on the master worker paradigm as shown.

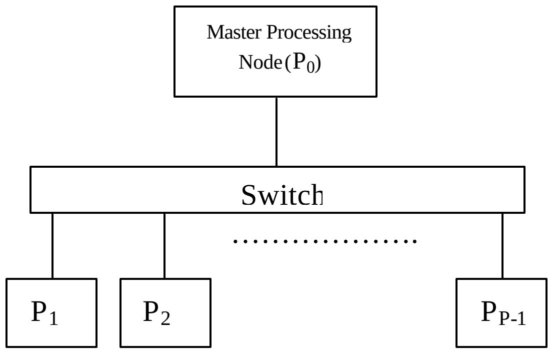


Figure 1: multi-computer system architecture

Figure 2 shows the flowchart of the proposed algorithm. Consider N to be the size of the problem executed by the parallel application. And P processors can be used to execute the application: P_0, P_1, …, P_{P-1}. As we stated above, EFP algorithm is a centralized one, so the fastest processor in selected as the central node to execute the load distribution algorithm. In the case of homogeneous system where all processors have the same processing capacity, our algorithm can select any one to be the central node. The central processor is assigned the identification number 0 (P_0), the rest of the processors are ordered based on their processing capacity where P_1 is the least powerful one and P_{P-1} is the most powerful one. The EFP algorithm is executed in the central node and consists of the following steps:

1. Divide the problem in a number of tasks equals to *G* and put them into a queue. The value of G is determined by the following expression:

$$G(P) = \lceil n \times P \times rand() \rceil \times (P \times (P-1)/2) \quad \ldots \quad (1)$$

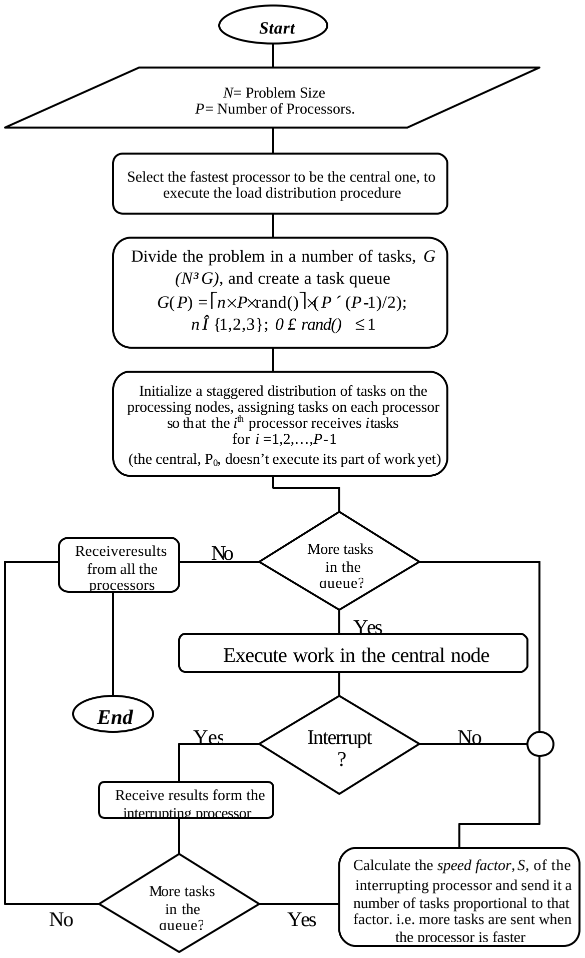
Where, $\lceil x \rceil$ is the least integer larger than or equals to x; n is a positive integer between 1 and 3; and rand() is a function that returns a real number between 0 and 1. Value of G increases proportional to the number of processors and it must be less than the problem size N. Therefore, increasing the number of processors result in fine-grain division for the problem. Using the integer n, we can prefix different values of the granularity for a fixed number of processors, as needed. In the experiments we have done, 1, 2, or 3 are used as values of n, as indicated.

2. The central processor P_0, performs a staggered distribution of tasks on the processing nodes, assigning tasks on each processor so that the ith processor receives *i* tasks for *i* =1,2,…,P-1.
3. When tasks are assigned on other processors, and while those are executing, the central processor executes tasks form its queue and attends to the incoming interrupts from other processors. In the case where there is an interrupt:
   - Identify the interrupting processor.
   - Receive results of the executed task along with some information relative to the processor speed.
   - If the task queue on the central processor is empty go to step 5.
   - Calculate the *Speed factor, S,* of the interrupting processor and send it a number of tasks proportional to that factor. i.e. more tasks are sent when the processor is faster.
4. If the task queue on the central processor is not empty go to step 3.
5. The central processor waits for results from all the processors that are terminating their assigned tasks, and sends a processing termination signal to the processors according to their finishing order.

In the design of the EFP algorithm we consider that the computing platform can be heterogeneous, which means that processing nodes can have different processing capacity and different memory capacity. To be able to implement the algorithm it is needed to have information that can help in deciding about the processing power of the nodes. E.g. in our implementation, three parameters are used to decide about the processing power: the clock frequency of the processor, internal cache capacity, and the RAM capacity of the node. The processor with the most capacity is assigned to be the central processor, where our procedure of load distribution is executed. The rest of the processors are assigned identifiers in the order of their computing capacity. P_1 is the least capable processor, and P_{p-1} is the most capable one.

4. FLOWCHART AND ANALYSIS OF THE EFP ALGORITHM

The following chart shows the flow of the control and data operations achieved by the algorithm. All the operations are expressed above:



Start
N= Problem Size
P= Number of Processors.
Select the fastest processor to be the central one, to execute the load distribution procedure
Divide the problem in a number of tasks, G (N³G), and create a task queue
G(P) =⌈n×P×rand()⌉×(P ´ (P-1)/2);
n Î {1,2,3}; 0 £ rand() ≤1
Initialize a staggered distribution of tasks on the processing nodes, assigning tasks on each processor so that the ith processor receives i tasks for i =1,2,…,P-1 (the central, P0, doesn't execute its part of work yet)
Receiveresults from all the processors
No
More tasks in the queue?
Yes
Execute work in the central node
End
Yes
Interrupt ?
No
Receive results form the interrupting processor
Calculate the speed factor, S, of the interrupting processor and send it a number of tasks proportional to that factor. i.e. more tasks are sent when the processor is faster
More tasks in the queue?
No
Yes

Figure 2: flowchart of the EFP algorithm

In the first round $(P(P-1)/2)$ tasks are distributed, so that the ith processor receives i tasks for $i=1,\ldots,P-1$. Thus, slower processors never have more tasks than the faster processors. On the other hand, using staggered distribution it is less probable that two or more processors finish their assigned work and interrupt the central processor simultaneously, to send back results and ask for new tasks. This is important, especially when having a shared bus network where the contention on the bus increases the communication time, consequently the performance of the system is affected negatively.

When the central processor completes the distribution in the first round, it continues executing tasks from its queue, at the same time it enables interrupts from other processors when they need to send back results and receive new tasks to execute. Really, the procedure executed in the central processor simulates the interrupt through a periodic check for interrupting messages sent by other processors. If an interrupt request from another processor is detected, the central processor executes an interrupt service routine by which it identifies the interrupting processor, and then receives the results and further information about the processor speed. Specifically: computation time, communication time, wait time, and the quantity of tasks executed. This information is sent to the central processor to help in deciding how much tasks it will send to the interrupting processor. The processor which finishes its assigned work more rapidly will receive more work in the next assignment, and otherwise it will receive less work in the next assignment. When there are no tasks in the queue, the central processor finishes the current task and waits for results from all the processors that are terminating their assigned tasks.

It is evident that the EFP algorithm is simple and has linear time complexity. The time of the algorithm is $O(G)$. This simplicity makes small overhead of load distribution on the central processor.

5. EXPERIMENTAL RESULTS:

In this section we describe the experimental result obtained from generating and studying the proponed algorithm. It will be shown that the fastest processor always executing more tasks. And the finish time of the processors is similar due to the suitable distribution done by the algorithm. This improves the utilization of the processors.

As a benchmark, in this paper we have used the matrix multiplication algorithm because of its simplicity and scalability. The product of two matrices A and B is defined by $c_{ij} = \sum_{k=1}^{n} a_{ik} \times b_{kj}$, where a_{ij}, b_{ij}, and c_{ij} is the element in the i'th row and jth column of the matrix A, B, and C respectively, and C is the result matrix. for simplicity we use a square matrices of order n. So we can increase or decrease the workload by simply changing the order of matrices n. Thus, the matrices A, B, and C are n×n matrices. The sequential algorithm of this matrix multiplication requires n^3 multiplications and additions, therefore its time complexity is $O(n^3)$.

We have selected this algorithm because of its simplicity and because it is one of the most important linear algebra algorithms which may simulate many real applications like image processing, video compression, …etc. also the workload of the MM algorithm is scalable and very easy to modify.

A heterogeneous cluster of computers connected by fast Ethernet is used as the parallel processing platform in our experiments. Table 1 shows the characteristics of the cluster used. Massage passing library MPI is used as the parallel programming environment implemented in C.

TABLE 1: CHARACTERISTICS OF THE CLUSTER USED

Number of nodes	CPU Speed (MHZ)	Cache size (KB)	RAM size (MB)
1	PIII 1000	512	118
8	PII 333	512	120

The following figure shows the effect of the workload on the execution time (seconds) with a fixed number of processors.

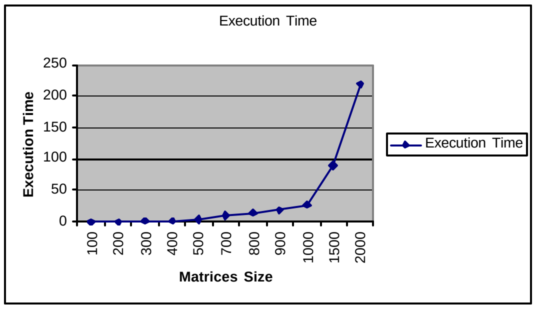


Figure 3: effect of matrix size (workload) on the execution time.

Table 2 and figure 4 show the detailed execution results with the matrix size 100×100. It is obvious in the table that the central processor multiplies the greatest number of rows, and the other processors almost multiply similar number of rows. Also shown the three components of the execution time, wait time (Twait), communication time

(Tcomm), and computation time (Tcomp). Because of the small size of the matrix (small workload) used it is obvious that the major component of the execution time is the communication time.

TABLE 2: EXECUTION RESULTS WITH 100×100 MATRIX

Processor	Number of Rows	Texec	Tcomp	Tcomm	Twait
0	9	0.072	0.013	0.055	0.004
1	10	0.071	0.014	0.049	0.007
2	8	0.072	0.008	0.06	0.004
3	8	0.071	0.009	0.057	0.005
4	6	0.07	0.009	0.052	0.009
5	8	0.072	0.011	0.05	0.01
6	9	0.072	0.013	0.048	0.011
7	10	0.072	0.014	0.046	0.011
8	32	0.078	0.078	0	0

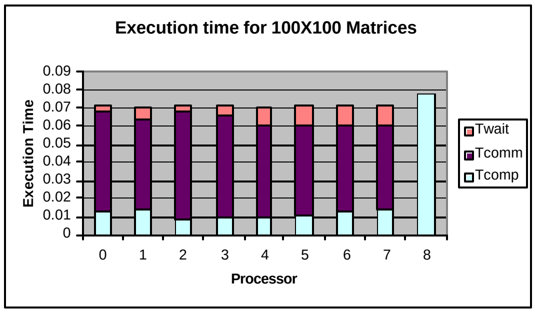


Figure 4: Execution times for a matrices sizes 100×100.

Tables 3 through 6 and figures 5 through 8 show the detailed execution results with the matrix size 400×400, 800×800, 1000×1000, and 2000×2000, respectively, it is obvious in the table that the central processor multiplies the greatest number of rows, and the other processors almost multiply similar number of rows. Also shown the three components of the execution time, wait time (Twait), communication time (Tcomm), and computation time (Tcomp). In contrast to the previous execution and because of the increase in work load (matrix size), It is obvious that the major component of the execution time is the computation time, which increases efficiency.

TABLE 3: EXECUTION RESULTS WITH THE MATRIX SIZE 400×400

Processor	Number of Rows	Texec	Tcomp	Tcomm	Twait
0	38	1.853	1.457	0.251	0.146
1	37	1.837	1.426	0.209	0.202
2	46	1.806	1.221	0.548	0.036
3	40	1.792	1.249	0.452	0.091
4	32	1.827	1.37	0.2	0.257
5	36	1.863	1.38	0.17	0.313
6	34	1.806	1.297	0.142	0.367
7	34	1.84	1.294	0.122	0.424
8	103	1.865	1.865	0	0

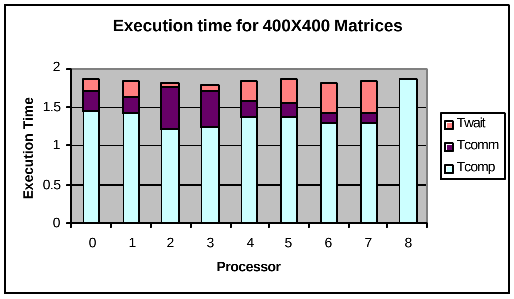


Figure 5: Execution times for a matrices sizes 400×400.

TABLE 4: EXECUTION RESULTS WITH THE MATRIX SIZE 800×800

Processor	Number of Rows	Texec	Tcomp	Tcomm	Twait
0	75	13.667	12.169	0.928	0.57
1	76	13.999	12.306	0.903	0.79
2	99	13.35	11.042	2.178	0.13
3	84	13.269	11.185	1.735	0.349
4	65	13.517	11.741	0.765	1.011
5	72	13.725	11.658	0.835	1.233
6	69	13.503	11.173	0.874	1.455
7	68	13.441	11.011	0.749	1.681
8	192	14.039	14.039	0	0

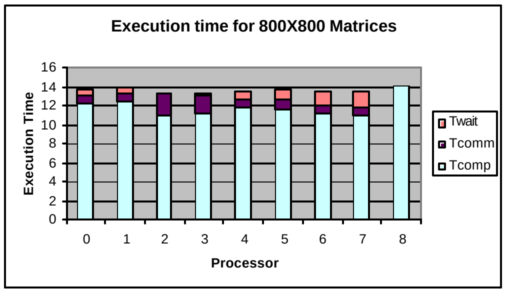


Figure 6: Execution times for a matrices sizes 800×800.

TABLE 5: EXECUTION RESULTS WITH THE MATRIX SIZE 1000×1000

Processor	Number of Rows	Texec	Tcomp	Tcomm	Twait
0	84	24.996	23.413	0.698	0.886
1	89	26.786	24.804	0.747	1.235
2	122	26.001	23.455	2.346	0.2
3	106	25.667	24.211	0.914	0.542
4	77	26.243	23.947	0.714	1.583
5	83	25.699	23.122	0.642	1.935
6	84	26.331	23.409	0.635	2.287
7	78	24.916	21.731	0.542	2.643
8	277	26.829	26.829	0	0

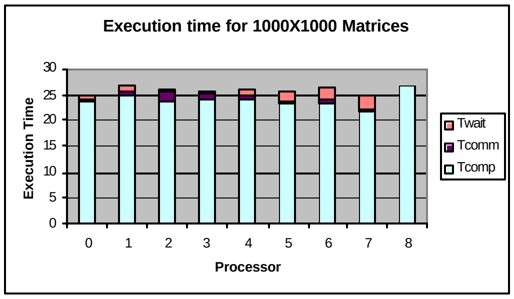


Figure 7: Execution times for a matrices sizes 1000×1000.

TABLE 6: EXECUTION RESULTS WITH THE MATRIX SIZE 2000×2000

Processor	Number of Rows	Texec	Tcomp	Tcomm	Twait
0	179	211.706	205.747	2.361	3.597
1	180	213.177	206.159	2.013	5.005
2	265	213.088	209.345	2.937	0.807
3	200	195.12	190.405	2.518	2.197
4	181	212.706	206.747	2.361	3.597
5	181	217.244	207.515	1.859	7.87
6	176	212.779	201.625	1.828	9.327
7	176	214.054	201.779	1.484	10.791
8	462	218.367	218.367	0	0

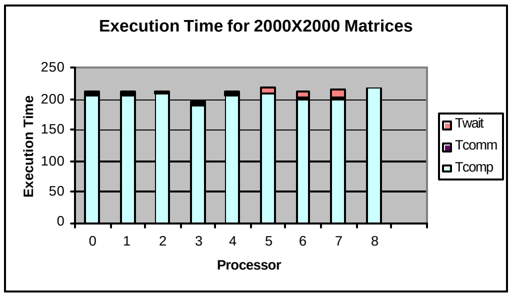


Figure 8: Execution times for a matrices sizes 2000×2000.

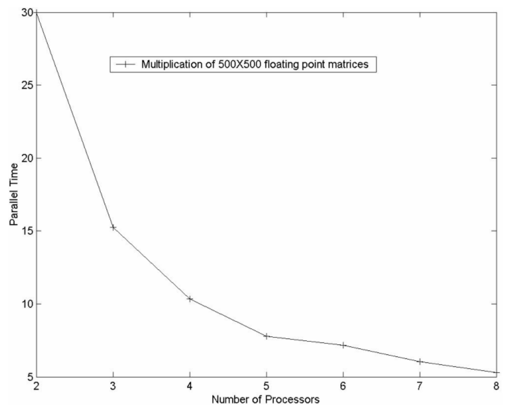


Figure 9: effect of number of processors on the Execution time (seconds) for fixed matrix size.

As shown in figure 9, as the number of processors increase the total parallel execution time decreases. Figure 10 shows a comparison of the static load balance efficiency and the dynamic load balance efficiency. Figure 11 shows a comparison of the static load balance speedup and the dynamic load balance speedup, it is clear in the figure that our algorithm using the dynamic load balance achieves better efficiency and speedup.

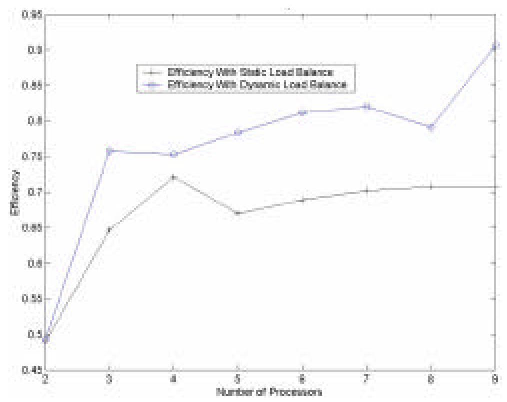


Figure 10: efficiency evolution comparison as a function of the number of processors, P, using a matrix size 1000×1000.

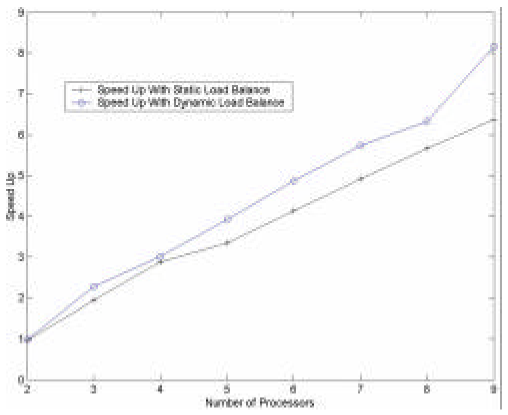


Figure 11: Speedup evolution comparison as a function of the number of processors, P, using a matrix size 1000×1000.

6. CONCLUSION

In this paper we have proposed a centralized dynamic load distribution algorithm capable to achieve efficient load distributions in a heterogeneous parallel environment. We started with presenting in detail the operations and characteristics of a dynamic load distribution algorithm in section II.

To compare the results we have implemented a static load distribution procedure which performs the load distribution before the execution begins. Our proposed algorithm denominated EFP is capable to achieve good distributions permitting that all processors terminate their work almost simultaneously. Furthermore, this algorithm is shown to be efficient in heterogeneous parallel platforms. As shown in figures 4-8 above.

We have used matrix multiplication as a benchmark executing in a heterogeneous platform. The obtained results show that EFP algorithm is efficient and adaptive for different workload (matrix sizes) and different number of processors. However, it is important to take into account that centralized approach present a bottleneck in the central node. This bottleneck can limit the scalability of the of the parallel algorithm, to deal with this problem, the EFP algorithm always selects the fastest and the most powerful processor to be the central node. Our experiments show that the algorithm maintains a good increase in the efficiency and speedup as the number of processors increase. As shown in figures 10 and 11.

BIBLIOGRAPHY

[1] Theys, M.D., et al.:"What are the Top Ten most Influential Parallel and Distributed Processing concepts of the past millenium?". Journal of Parallel and Distributed Computing, Vol.61, pp.1827-1841, 2001.

[2] Antonis, K.; Garofalakis, J.; Mourtos, I. ; Spirakis, P. : "A hierarchical adaptive distributed algorithm for load balancing". J. Parallel Distrib. Comput., 64, pp.151 -162, 2004.

[3] Barker, K.; Chernikov, A.; Christoides, N.; Pingali, K.:"A Load Balancing Framework for Adaptive and Asynchronous Applications". IEEE Trans. on Parallel and Distributed Systems, Vol.15, No. 2, pp.183-192. Febrero, 2004.

[4] Luis Paulo Peixoto Dos Santos, Load Distribution: A Survey, Technical Report, Universidade do Minho, Departamento do Informática Oct. 1996.

[5] Shahzad Malik, Dynamic Load Balancing in a Network of Workstations, 95.515F Research Report, Nov. 2000.

[6] Mohammed Javeed Zaki, Wei Li, and Srinivasan Parthasarathy, Customized Dynamic Load Balancing for a Network of Workstations, Journal of Parallel and Distributed Computing 43, pp. 156–162 (1997).