An Efficient Algorithm for Load Balancing in Multiprocessor Systems

Saleh A. Khawatreh
Dept. of Computer Engineering, Faculty of Engineering, Al-Ahliyya Amman University
Amman-Jordan

***Abstract*—A multiprocessor system is a computer with two or more central processing units (CPUs) with each one sharing the common main memory as well as the peripherals. Multiprocessor system is either homogeneous or heterogeneous system. A homogeneous system is a cluster of processors joined to a high speed network for accomplishing the required task; also it is defined as parallel computing system. Homogeneous is a technique of parallel computing system. A heterogeneous system can be defined as the interconnection of a number of processors, having dissimilar computational speed. Load balance is a method of distributing work between the processors fairly in order to get optimal response time, resource utilization, and throughput. Load balancing is either static or dynamic. In static load balancing, work is distributed among all processors before the execution of the algorithm. In dynamic load balancing, work is distributed among all processors during execution of the algorithm. So problems arise when it cannot statistically divide the tasks among the processors. To use multiprocessor systems efficiently, several load balancing algorithms have been adopted widely. This paper proposes an efficient load balance algorithm which addresses common overheads that may decrease the efficiency of a multiprocessor system. Such overheads are synchronization, data communication, response time, and throughput.**

Keywords—Multiprocessor system; homogeneous system; heterogeneous system; load balance; static load balancing; dynamic load balancing; response time; throughput

I. Introduction

Parallel processing has emerged as a key enabling technology in modern computers, driven by the ever increasing demand for higher performance, lower costs and sustained productivity in real life applications. Concurrent events are taking place in today's high-performance computers due to the common practice of multiprogramming and multiprocessing [1].

Parallel processing is an efficient form of information processing. Parallel events may occur in multiple resources during the same interval. Parallel processing demands concurrent execution of many programs in the computer [2].

Multiprocessor management and scheduling has been a fertile source of interesting problems for researchers in the field of computer engineering. In its most general form, the problem involves the scheduling of a set of processes on a set of processors with arbitrary characteristics in order to optimize some objective function.

Basically, there are two resource allocation decisions that are made in multiprocessing systems. One is where to locate code and data in physical memory, *a placement decision*. The other is on which processor to execute each process, an *assignment decision*. Assignment decision is often called processor management. It describes the managing of the processor as a shared resource among external users and internal processes. As a result, processor management consists of two basic kinds of scheduling: long-term external load scheduling and short-term internal process scheduling [1], [2].

A scheduler performs the selection a process from the set of ready to run processes, and assigns it to run on a processor in the *short-term process scheduling* operation. The *medium* and *long-term load-scheduling operation* is used to select and activate a new process to enter the processing environment [2].

The general objectives of many theoretical scheduling algorithms are to develop processor assignments and scheduling techniques that use minimum numbers of processors to execute parallel programs in the least time. In addition, some algorithms are developed for processor assignment to minimize the execution time of the parallel program when processors are available. There are two types of models of scheduling *deterministic* and *nondeterministic*. In deterministic models, all the information required to express the characteristics of the problem is known before a solution to the problem, a schedule, is attempted. Such characteristics are the execution time of each task and the relationship between the tasks in the system. The objective of the resultant is to optimize one or more of the evaluation criteria. Nondeterministic models, or *stochastic models*, are often formulated to study the dynamic-scheduling techniques that Adaptive Scheduling Algorithm for Load Balance in Multiprocessor System take place in a multiprocessor system [2].

The simplest dynamic algorithm is called *self-scheduling*. Self-scheduling [6] achieves almost perfect load balancing. Unfortunately, this algorithm incurs significant synchronization overhead. This synchronization overhead can quickly become a bottleneck in large-scale systems or even in small-scale systems if the execution time of one process is small. *Guided self-scheduling* [6], [7] is a dynamic algorithm that minimizes the number of synchronization operations needed to achieve perfect load balancing.

Guided self-scheduling algorithm can suffer from excessive contention for the work queue in the system.

Adaptive guided self-scheduling [6], [7] address this problem by using a backoff method to reduce the number of processors competing for tasks during periods of contention. This algorithm also reduces the risk of load imbalance.

Adaptive guided self-scheduling algorithm performs better than guided self-scheduling in many cases.

All these scheduling algorithms attempt to balance the workload among the processors without incurring substantial synchronization overhead.

An *affinity scheduling algorithm* [5], [6] attempts to balance the workload, minimize the number of synchronization operations, and exploit processors affinity. Affinity scheduling Employs a per-processor work queues which minimizes the need for synchronization across processors.

Adaptive affinity scheduling algorithm [5], [6] maintains excellent load balance and reduces synchronization overhead. The main idea behind this algorithm is to minimize local scheduling overhead so that the phase of dynamically balancing the workload can be speeded up, which results in reduction of execution time.

There are many other different algorithms for scheduling the workload on multiprocessor systems. Such algorithms are the *factoring algorithm,* the *tapering algorithm,* and the *trapezoid self-scheduling algorithm.*

These algorithms basically depend on the described algorithms in them structure with some alterations made for improving the algorithm in some characteristic or another [8].

A task for transfer is chosen using *Selection Strategy*. It is required that the improvement in response time for the task and/or the system compensates the overhead involved in the transfer. Some prediction that the task is long-lived or not is necessary in order to prevent any needless migration which can be achieved using past history [8], [10], [11].

The dynamic load balance algorithm applies on Folded Crossed Cube (FCC) network. Basically, FCC is a multiprocessor interconnection network [9].

This paper is divided into the following sections: The proposed method is described in Section 2. Results of the study are analyzed in Section 3. Finally, Section 4 presents the conclusions.

II. Proposed Scheduling Algorithm

A. Assumptions and Considerations

In this section, some considerations are stated concerning the multiprocessor system for which the algorithm is designed. Following are the main assumptions characterizing this multiprocessor system:

1) System Topology

The multiprocessor system is assumed to be configured using *homogeneous* processors. These processors are connected using the *crossbar switches* organization of interconnection networks. Crossbar switches have a good potential for high bandwidth and system efficiency.

2) Operating System Characteristics

During the course of design of the scheduling algorithm, it became apparent that the most suitable operating system to govern the multiprocessor system is the *floating supervisor control.* This operating system provides the flexibility needed to implement the scheduling algorithm since it treats all the processors as well as other resources symmetrically, or as an anonymous pool of resources.

3) Design Characteristics

Similar to the affinity and adaptive affinity scheduling algorithms [5], [6], the proposed scheduling algorithm is also constructed to have three phases as follows:

B. Process Scheduling Phase

Processes arriving at *a **process queue*** are organized using *Nonpreemptive Priority* scheduling algorithm, where the process with the highest priority is always found at the top of the process queue. When two processes have the same priority *First-In-First-Out* scheduling algorithm is applied.

C. Processor Scheduling Phase

In this phase, processes are distributed among ***processor queues.*** Each processor in the system has a local queue scheduled using *Round-Robin* scheduling algorithm with a dynamically adjustable quantum. Processor work states are defined in this phase, and are used to achieve a balanced load distribution in the multiprocessor system.

D. Remote Scheduling Phase

This phase is concerned with ***load redistribution*** in case *a faulty processor* or *a heavily loaded processor is* detected. A feedback approach is utilized to transfer the processes in a faulty or heavily loaded processor back to the process queue for redistribution. This phase ensures that the reliability of the system is maintained and thrashing is avoided. Fig. 1 illustrates the basic idea behind the design.

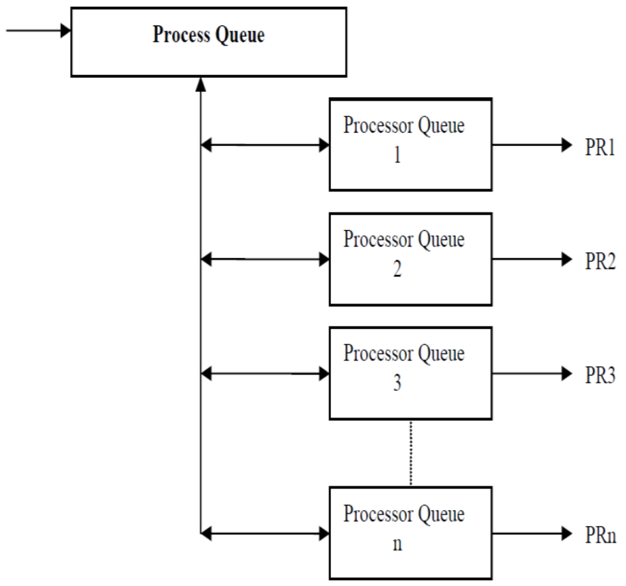

Fig. 1. Process and processor queues.

E. *Scheduling Algorithm Design*

This section contains the detailed design of the proposed scheduling algorithm. The algorithm consists of three phases, as described above, and it proceeds as follows:

1) *Process Scheduling Phase*

a) Construct a process queue Q.

b) Each process *Pi* arrives to the process queue carrying a priority variable *PPi.*

The process with the highest priority is assigned an *HP* variable; the **highest priority variable.**

c) To place a process *Pi* in the right priority order, its *PPi* is compared with *HP*. Then after iteratively with the rest of the *PP's* until the correct position is found.

2) *Processor Scheduling Phase*

a) Construct processor queues *PQi* for each processor *PRi*

b) The work states of a processor are partitioned as follows:

- Faulty Processor FP: NF = NT.
- Heavily-Loaded Processor *HL:* NF >NT/2.
- Normally-Loaded Processor
- *NL:* NF=NT/2.
- Lightly-Loaded Processor *LL:* NF <NT/2.

Where;

NT is the total number of processes in *a PQ.*

NF is the number of processes left in *a PQ.*

3) Define a ***processor-state checking variable*** *SCi for each PRi. This variable indicates the state of a PRi as follows:*

a) For *a FP.*• SC = 3.

b) For *a HL* processor: *SC = 2.*

c) For *a NL* processor: *SC = 1.*

d) For *a LL* processor: SC = 0.

4) Distribute processes to PQ's from Q by checking the SC variable for each ***PQ.***

A process is assigned to the *LL* processors first, then to the *NL* processors. In case of high load, processes are also assigned to the *HL* processors when needed.

5) *Remote Scheduling Phase*

a) A procedure for checking the workload in each PR is as follows:

- When SC = 3, a FP is detected:
  NR=NF.
- When *SC = 2,* a HL is detected:
  NR = NT - NF.

Where;

NR is the number of processes to be remotely scheduled.

b) NR processes are returned to *Q* for redistribution to *LL* processors and *NL* processors.

c) This procedure is repeated until the *SC* variable for all PR's indicates *a NL* or *LL* work states.

TABLE I. WITH LOAD BALANCE

Pr..	Resp. Time	Exec. Time	Start Time	End Time	Av. Res. Time	Orig. Job	Mig. Job	Ack. Job	Run Jobs
1	40	37	1	64	3.64	14	5	2	11
2	21	21	4	66	1.91	6	0	5	11
3	22	22	1	64	2.00	13	2	0	11
4	23	21	2	47	2.30	9	1	2	10
5	13	13	14	66	1.86	7	0	0	7
6	10	10	3	68	1.43	7	0	0	7
7	11	11	3	68	1.38	8	0	0	8
8	33	26	3	39	2.36	14	0	0	14
9	10	10	3	47	1.25	8	0	0	8
10	31	30	4	48	2.38	14	1	0	13
	214	201	1	68	2.14	100	9	9	100

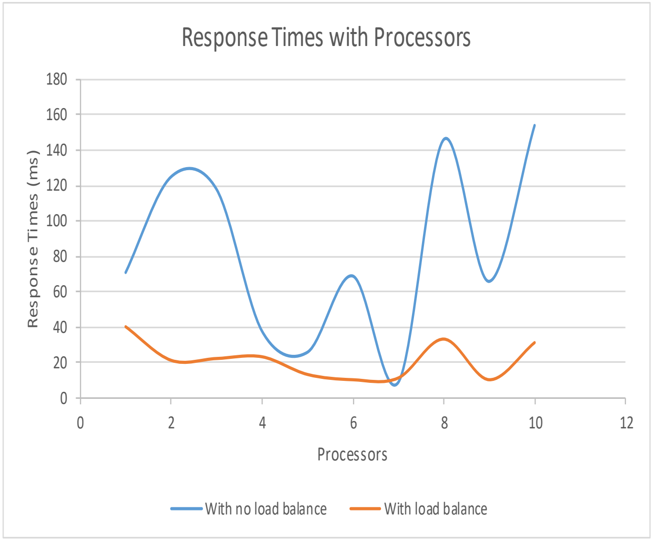

Fig. 2. Response times variations.

III. RESULTS

This algorithm is applied on a simulated system consists of 10 processors and 100 jobs.

The response time is taken as a performance measure. Table I contains the results where the load balance algorithm is applied.

Table II contains the results where load balance algorithm is not applied.

By comparing the results of Table II with the results of Table I, the difference in the response time is clear where the total response times is 214 ms compared with 822 ms.

Fig. 2 shows the difference in the response time between the systems with load balance and without load balance.

The terms which are used in the tables are defined as follows:

Response time = finish time – arrival time

Average response time = response time/ number of jobs executed

Orig. jobs = jobs which originated at this processor.

Mig. Jobs = jobs which were migrated to other processors.

Acq. Jobs = jobs which were acquired from other processors.

TABLE II. WITH NO LOAD BALANCE

Pr.	Resp. Time	Exec. Time	Start Time	End Time	Average Res. Time	Orig. Job	Mig. Job	Ack. Job
1	71	48	1	64	5.46	13	0	0
2	125	37	2	39	12.5	10	0	0
3	118	46	1	52	11.8	10	0	0
4	38	35	5	44	5.43	7	0	0
5	26	24	2	64	3.71	7	0	0
6	69	39	3	58	6.9	10	0	0
7	9	9	3	22	1.8	5	0	0
8	146	43	3	64	12.17	12	0	0
9	66	52	3	65	5.5	12	0	0
10	154	51	4	55	11.0	14	0	0
	822	384	1	65	8.22	100	0	0

IV. CONCLUSION

The main objective of this paper is to design an algorithm that achieves a balanced load state on a multiprocessor system. The proposed scheduling algorithm is a deterministic dynamic algorithm, which outlines an excellent load balancing strategy. It also addresses some major overheads that may prove problematic in the multiprocessing operation. Multiprocessor systems have many advantages, which make them economical compared to multiple single systems. One advantage is *increased throughput. By* increasing the number of processors, more work would get done in a shorter period of time.

Another reason for multiprocessor systems is that they increase *reliability.* When functions are distributed properly among several processors, failure of one processor will not halt the system, but would only slow it down. This ability to continue providing service proportional to the level of non-failed hardware is called *graceful degradation. Resource sharing, computation speedup,* and *communication* are other advantages for building multiprocessor systems [2]-[4].

In the algorithm presented in this paper, we tried to maximize the advantages of multiprocessor systems. By achieving a balanced load distribution state on the multiprocessor system, it was possible to observe the properties of this system. Throughput is increased. Graceful degradation is another apparent characteristic. In addition to these advantages, this algorithm overcomes many overheads that may occur in a multiprocessor system when it applies other algorithms. One overhead is synchronization. In the presence of the process queue and processor queues, synchronization does not have to be addressed as a complication. Synchronization is automatically achieved in the design of the adaptive scheduling algorithm. Data communication overheads are minimum, since the queue length at all times is kept relatively short for all the system queues. The only state which may suffer this overhead is when the system is in a high load state.

The proposed scheduling algorithm adopts an organization with a process queue, where each arriving process to the system is entered, and processor queues for each existing processor in the system. The processes are distributed from the process queue to processor queues, where they await execution. The process queue is scheduled with *a nonpreemptive priority algorithm.* This has many advantages but may present some limitations. One advantage is prevention of deadlocks [4]. The main problem of priority scheduling is starvation [4]. This can be solved by aging low priority processes [4]. Processor queues are scheduled using *Round-Robin scheduling algorithm.* The quantum is utilized here as a control factor. Response time and the throughput depend on the quantum, which may be dynamically adjusted to give the desired characteristics [4]. A major limitation of RR scheduling is the switching between processes present in the processor queues. This presents an overhead to this algorithm, which may be overcome by practical testing to achieve an optimal quantum value. On comparing it with other scheduling algorithms, the proposed scheduling algorithm proved superior to them in many aspects. In its unique design of having both a process queue supported by processor queues, the proposed scheduling algorithm utilized the advantages of the various other designs while overcoming many of them limitations. The presence of a central work queue unsupported in a multiprocessor tends to be a performance bottleneck, resulting in a longer synchronization delay. Heavy traffic is generated because only one processor can access the central work queue during allocation. The third limitation is that a central work queue does not facilitate the exploitation of processor affinity. On the other hand, including a process queue in the presented design, provides the possibility of evenly balancing the workload. To eliminate the central bottleneck, the proposed scheduling algorithm supported the process queue with local processor queues. This approach reduces contention and so, prevents thrashing. Thrashing occurs when the degree of multiprocessing increases beyond the maximum level and this causes the system utilization to drop sharply [4]. A central work queue may cause thrashing during heavy traffic.

For future work, the process queue is to be scheduled with *a preemptive priority algorithm* and the results will be compared with non-preemptive queue scheduling.

REFERENCES

[1] K. Hwang, 'Advanced Computer Architecture: Parallelism, Scalability, Programmability, 'McGraw-Hill, Inc., 1993.

[2] K.Hwang and F.A.Briggs` Computer Architecture and Parallel Processing,' McGraw-Hill, Inc., 2002.

[3] A. S. Tanenbaum, `Computer Networks,' Prentice Hall, FourthEdition,2003.

[4] Silberschatz and Galvin, ` Operating System Concepts,' Addison-Wesley Publishing Company, sixth Edition,2003.

[5] Y.Yan, C.Jin, and X.Zhang,`A datively Scheduling Parallel Loops in Distributed Shared-Memory System,' IEEE Trans. Parallel and Distributed. Systems, Vol.8, No. 1, pp. 70-81, January 1997.

[6] E.P.Markatos and T.J.LeBlanc,` Using Processor Affinity in Loop Scheduling On Shared-Memory Multiprocessors ,' IEEE Trans. Parallel and Distributed Systems, Vol.5, No. 4, pp. 370-400, April 1994.

[7] C.D. Polychronopoulous andD.J.Kuck, `Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Super Computers,' IEEE trans. Computer, Vol. C-36, No. 12, pp.1425-1439, December 19987.

[8] Samad, A., Siddiqui, J., & Ahmad, Z,' Task Allocation on Linearly Extensible Multiprocessor System. *International Journal of Applied Information Systems*, *10*(5), 1–5, 2016.

[9] Samad, A., Siddiqui, J., & Khan, Z. A.,' Properties and Performance of Cube-Based Multiprocessor Architectures,' *International Journal of Applied Evolutionary Computation*, *7*(1), 63–78, 2016.

[10] Singh, J., & Singh, G., 'Improved Task Scheduling on Parallel System using Genetic Algorithm,' *International Journal of Computers and Applications*, *39*(17), 2012.

[11] K., Alam, M., & Sharma, S. K. (2015). A Survey of Static Scheduling Algorithm for Distributed Computing System. Interconnection Network. International Journal of Computers and Applications, 129(2),25–30, 2015.