Proceedings of the 29th Annual Hawaii International Conference on System Sciences — 1996

A Dynamic Load Balancing Algorithm for a Heterogeneous Computing Environment

Piyush Maheshwari
School of Computing and Information Technology
Griffith University, Brisbane, Queensland, Australia 4111

Abstract

Heterogeneous computing opens up new challenges and opportunities in fields such as parallel processing, design of algorithms for applications, partitioning and mapping of parallel tasks, interconnection network technology and the design of heterogeneous programming environments. Lots of load balancing algorithms have been proposed and experimented with in the past years for homogeneous parallel and distributed systems. We present a priority-based decay usage load balancing algorithm for a heterogeneous computing environment. The algorithm determines the task precedence graph of the parallel jobs dynamically at run-time and assigns appropriate priorities to the processes to resolve the dependencies. The heuristic algorithm has been tested on some heterogeneous program models.

1. Introduction

1.1 Heterogeneous computing (HC)

Hardware and software heterogeneity arises in many computing environments, for example, in an academic department with different experimental research machines and software systems. Heterogeneity may arise by conducting research experiments on the mixture of multiprocessor workstations or specific language machines. Heterogeneity in computation may help us to better map a wider class of algorithms, either onto mixed-mode (e.g. SIMD/MIMD) architectures or onto a suite of heterogeneous architectures. A heterogeneous computing environment (HCE) consists of a connected set of traditional computer systems. Open architectures, workstations and multicomputers are a natural environment for heterogeneity. A simple heterogeneous computing environment could be a departmental network with 3 VAX machines, 16 SUN workstations, and 1 Symbolics Lisp machine. Similarly several autonomous high-performance parallel machines may be connected to each other via a high-speed network to build a heterogeneous supercomputing environment. Heterogeneous computing is a promising cost-effective approach to the design of high-performance computers, which generally incorporates proven technology and existing designs and reduces new design risks from scratch [1].

Several heterogeneous architectures for high-performance computing could be (a) mainframe with integrated vector unit, (b) vector (pipeline and array) processors, (c) attached processors, (d) multiprocessor and multicomputer systems, or (e) special-purpose architectures [2]. While in distributed heterogeneous computing, dissimilar computing machines are spread around a network geographically, an integrated heterogeneous computing system consists of dissimilar machines used in one system, linked by a bus or backplane. Both types give the user greater flexibility in matching the capabilities of the system to the particular needs of a specific application.

Donaldson *et al.* [3] have considered a theoretical basis for calculating speedup in a heterogeneous environment and shown that not only is superlinear speedup possible, but unbounded speedup is possible for linear task graphs even without exploiting parallelism across machines. This suggests that heterogeneous computing may provide significant speedups over homogeneous systems.

1.2 Partitioning and mapping in heterogeneous computing

The code types in an HC application are used to partition the application into program modules. This means that classification of the code based on the type of parallelism is required. Code types that can be identified include vectorizable, SIMD/MIMD parallel, scalar and/or special purpose [4]. Each module contains a single type of parallelism. Modules are clustered together and

Proceedings of the 1996 Hawaii International Conference on System Sciences (HICSS-29)
1060-3425/96 $10.00 © 1996 IEEE

then mapped onto available machines to get the best overall performance. Most of the known work on mapping does not account for heterogeneity of processors or modules. HC poses new constraints on mappings. Like homogeneous parallel computing, the execution time of a program module in HC depends on the data dependencies, the communication costs, and the available processing power [5]. In HC, this execution time also depends on the type of parallelism in the module and the type of machine executing the module. Matching code-type to the machine-type imposes additional constraints on the mapping problem in HC. The mapping problem becomes even more complex if we take issues like load balancing or machine failure into account [6].

Most homogeneous multiprocessors are multiprogrammed, and there is no reason to expect this to change for heterogeneous parallel systems. As in homogeneous parallel systems, scheduling is also one of the important phases of heterogeneous processing; improper load balancing, caused by differences in processor capability, can lead to reduced performance [7]. There could be static or dynamic scheduling policies in any computer system. In HC, apart from usual scheduling methods, a different level of scheduling is needed at the system level. The scheduler has to maintain a balanced system-wide workload by monitoring the progress of all the tasks in the system. Communication bottlenecks and queuing delays are more prominent due to the heterogeneity of different hardware architectures which add different constraints on the scheduling policies. The scheduler also needs to know the different types of tasks and available machine types (e.g., SIMD, MIMD, vector, mixed-mode type, etc.). Hence the issues related to scheduling become more complicated in heterogeneous computing machines, which definitely require further research [1, 4].

The paper is organized as follows. In the next section we briefly discuss some issues related to load balancing in a heterogeneous computing environment. Section 3 gives an overview of the experimental framework used to study the priority-based load balancing algorithm for an HCE presented in Section 4. The algorithm determines the task precedence graph of the parallel program dynamically at run-time. We then discuss the behaviour of our algorithm on two simulated heterogeneous program models under different background load conditions in Section 5. Finally, conclusions and future work are presented in Section 6.

2. Load balancing in HC

The processing capacity of a heterogeneous computing system cannot be effectively exploited unless the resources are properly scheduled. In homogeneous systems, a scheduler typically consists of two components – the load distributor and the local scheduler. Load distributor improves performance by correcting anomalies that arise in distribution of the load among nodes by process transfer or migration. Local scheduler allocates local resources among the resident processes. Two distinct load distributing strategies for improving performance have been identified in homogeneous network systems [8]. Load sharing algorithms attempt to improve the performance of a few machines and their processes by assuring that no node lies idle while processes wait for service. Load balancing algorithms go a step further to equalize the workload among nodes. Casavant and Kuhl [9] presents a taxonomy of distributed scheduling in an attempt to provide a common terminology and classification of the problem for distributed computing systems.

Let us consider an HCE that consists of pools of machines, with each pool being a homogeneous distributed system. Scheduling in such systems consists of three parts – *task assignment*, *load balancing* and *local scheduling*. Task assignment is the process of matching tasks to machines that are best suited for their execution. Local scheduling acts within each pools of machines, following some distributed scheduling policy. Load balancing acts at a higher level and tries to reduce the idle time of machines. It should be noted that, in HCE, a machine can idle if there are no tasks (or processes) that matches with its architectures. Due to this reason, load sharing algorithms are not applicable in HCE.

Load balancing algorithms can be broadly classified into two categories – *static* and *dynamic* [10]. Static policies make load balancing (or task placement) decisions before run time resulting in low run time overhead. These policies need fairly accurate prediction of the resource utilization of each task at compile time, which is generally very difficult in most of the applications [11, 12]. Dynamic (or adaptive) policies, on the other hand, rely on recent system state information and determine the task assignments to processors at run time [13, 14, 15]. Hence, they are more attractive from a performance point of view [8].

The simplest form of load balancing, in which tasks

Proceedings of the 1996 Hawaii International Conference on System Sciences (HICSS-29)
1060-3425/96 $10.00 © 1996 IEEE

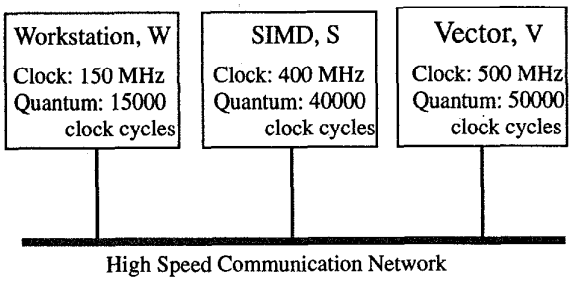

Figure 1. HCE configuration.

requiring differing times for completion are to be equally distributed as possible between two processors, is clearly equivalent to the partition problem, and is NP-complete [16]. In the context of heterogeneous computing, the distributed components of the load balancing algorithm should cooperate with each other in decision making and work towards a common system-wide goal.

Consider a simple HCE consisting of two machines, M_1 and M_2. Let P be a parallel program that has two tasks T_1 and T_2 assigned to M_1 and M_2 respectively. Assume that the tasks need to synchronize quite often. If the load on M_1 is high and if T_2 is the only task on M_2, then, assuming T_2 runs ahead of T_1, machine M_2 will be idling most of the time with T_2 waiting for T_1 to finish its computation cycle and send a synchronization message. This leads us to finding the *task precedence graph* of the given program to represent dependencies between the tasks. In most cases, these dependencies are not known ahead of run-time. This adds an extra burden on the programmer or the compiler. Our load balancing algorithm, presented in Section 4, tries to improve the response time of a heterogeneous program by determining the task precedence graph dynamically. This helps the algorithm to prioritize tasks to resolve the dependencies.

3. Our experimental HCE

Since there are no standard models and assumptions available in the literature, we consider a simple HCE consisting of three machines – a workstation, a vector machine and a SIMD machine. Each machine may represent a pool of machines of the same architecture, and load balancing issues within that pool are to some extent modelled by process scheduling on the single machine. Each machine in the HCE executes single process jobs as well as processes of parallel jobs. We model network

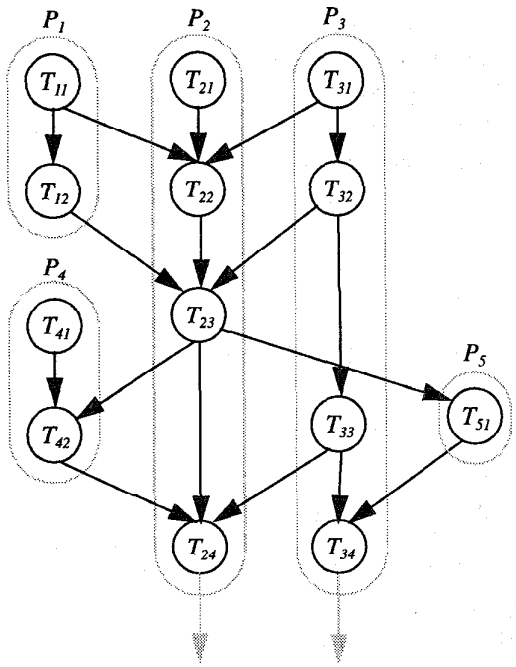

Figure 2. An example of task graph representing workload.

delay in message transmission between machines. We assume a maximum lag of one tenth of the least quantum interval among all the machines in the system, for the smallest size message possible in the HCE. For the simplicity of the simulator, the quantum interval has been kept same for all machines. Figure 1 gives details of the clock frequency and the quantum size of each machine of the HCE used in this study.

The base case local scheduler in each machine uses the round-robin process scheduling policy [17]. Round-robin scheduling is typically done in uniprocessor systems using two different queues, namely, the `io queue` and the `cpu queue`. In our simulation, we have extended the scheduler on each machine to take into account the parallel nature of jobs, by maintaining a `communication queue` for ready processes that have just received messages from other processes of the same job. These processes are provided high priority to those in the `io queue` or the `cpu queue` to help resolve dependencies as quickly as possible.

3.1 Workload representation

We represent parallel heterogeneous program as a task precedence graph (see Figure 2) [6, 18]. Each node

Proceedings of the 1996 Hawaii International Conference on System Sciences (HICSS-29)
1060-3425/96 $10.00 © 1996 IEEE

represents a task and each arc represents a dependency between the two nodes (tasks). Each task is defined to be a part of the parallel program that consists predominantly of one particular code structure. Task assignment is done by matching the code structures of each of these tasks with the available machines in the system.

All tasks having the same code structure that are assigned to a single machine can be combined into a single *process*, unless they can be executed concurrently. Each such process is given a unique name within the parallel program and dependencies between the tasks are modelled by message-passing between these processes. In our model, the processes do not wait after sending a message to a remote process, but start the execution of the next task, if there is any.

The `run` and `block` instructions have a single argument that specifies the number of clock cycles that the process has to execute or block. The `send` and `receive` instructions have the name of the destination or source process and the machine where it runs as arguments. The `send` instruction also contains the message. For every `send` instruction there is a matching `receive` instruction at the destination process.

3.2 The simulator

We have simulated the execution of above workload sets in the HCE in parallel over a physically distributed network of machines. We model each machine in HCE by a UNIX process, called the `machine_manager`. A front-end process, the `master`, initiates these `machine_managers` on different machines in the computing network used for simulation. A heterogeneous parallel program is submitted to the `master` and, based on the task assignment information, the `master` partitions it into different processes and assigns them to the different `machine_managers` for execution. A set of individual jobs can also be modelled as a parallel program that does not have any dependencies among the tasks.

`machine_manager` gets the description of the tasks submitted by the `master`, and manages the queues for ready processes, blocked processes and waiting processes. Different `machine_managers` are synchronized by imposing barriers at which they receive the actual messages sent by other machines since the previous barrier. This leads to an approximation in that the real arrival time of a message will differ from its actual dispatch time, with an upper bound on the difference being the barrier interval time. Using this approximation, we model the communication overhead in the HCE by the random delay in the network used for the simulation. The simulator is developed in the C programming language.

4. The priority-based dynamic load balancing algorithm

Our priority-based load balancing algorithm is a distributed and cooperative algorithm that tries to reduce the execution (response) time of the given heterogeneous parallel program by determining the task precedence graph dynamically. Each machine in the system consists of a load balancing agent as part of the local operating system which implements the priority algorithm.

We have modelled dependencies as a set of non-blocking `sends` and blocking `receives`. Whenever a process P_1 blocks for some other process P_2, the best way to reduce the waiting time of P_1 is to schedule P_2 without preemption until the dependency is removed, that is, until the message is sent to P_1. An analogy can be drawn from the scheduling of UNIX processes. An io-bound UNIX process can be modelled as a parallel program consisting of two tasks – a purely I/O one and a computation one. Whenever the io-task finishes execution on the I/O processor, it waits for the computation-task to get serviced on the CPU. This can be characterized as the io-task blocking for a message from the computation-task. In this simple example, UNIX enhances the priority of the computation-task so that it can finish and thereby resolve the dependency as quickly as possible.

The situation in HCE is quite different than the above example; here we can have several machines with lots of parallelism in the program. Conflicts may arise in situations when two processes in a machine are to be scheduled simultaneously to resolve two different dependencies as quickly as possible. To handle such situations, our algorithm implements and manages a `high priority queue` of ready processes on top of the `ready queue` managed by the local scheduler. It should be noted that higher numeric value implies lower priority. A process with *priority* = 0 has the highest priority. The load balancing component of the distributed scheduling consists of a load balancing agent in each machine. Each of these load balancing agents perform

Proceedings of the 1996 Hawaii International Conference on System Sciences (HICSS-29)
1060-3425/96 $10.00 © 1996 IEEE

the following:

- Whenever a process P_i blocks for a data (i.e., a message) from some other process P_j, a message is sent to the remote machine where P_j is running, requesting it to speedup the execution of P_j.
- On receipt of the message, the load balancing agent enhances the priority of process P_j. If P_j is already present in the `ready queue`, it is moved up into the `high priority queue` immediately; otherwise it is added to the `high priority queue` once it becomes ready.
- Processes in the `high priority queue` get scheduled before any process in the `ready queue` that is managed by the local scheduler. As a process gets service, its usage factor U increases depending on the duration for which it occupies the CPU.
- The priorities of these processes are degraded to the base level priority B by the *decay usage scheduling* algorithm [19]. Any process having a poorer value of priority than B leaves the `high priority queue` and joins the `ready queue`.
- The decay usage scheduling algorithm has three system-dependent parameters: D, the decay (CPU) usage factor; T, the scheduling (or decay) interval at the end of which U is decayed using D; and R, the rate at which priority increases for each quantum consumed (it basically converts units of CPU consumption into units of priority). R is different for each machine. We have made it proportional to the clock frequency so that the change in U is same for utilizing any CPU for the same number of seconds.
- Whenever a process whose priority is within the high priority range is context-switched, its priority is updated as follows:

  New priority = Old priority + (decayed) CPU usage

  and is inserted into the `high priority queue`.
- After every T quanta (i.e., a decay cycle), the load balancing agent decays the measure of the CPU consumed for all processes whose priorities are above the base priority B. It should be noted that the decay operation is done once every T quanta and not every T context-switches.

The above algorithm assumes no *a priori* knowledge of the jobs and relies on information collected dynamically at run-time. In the next section we analyse the behaviour of the algorithm for two different program models under diverse background load conditions.

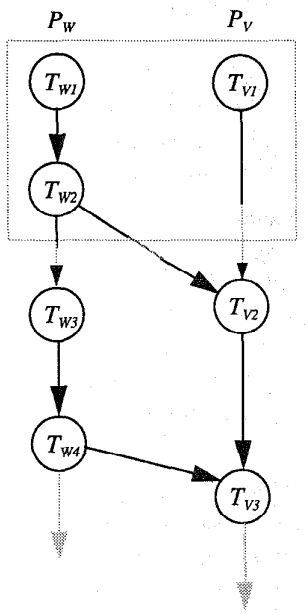

Figure 3. Load cycle of Heterogeneous Program Model I.

5. Experimental results

5.1 Heterogeneous Program Model I

Our first program consists of two parallel processes – one for the vector machine and the other for the workstation. It captures a typical realistic situation where a general-purpose machine like a workstation is heavily loaded, and hence is the bottleneck in reducing the execution time of the parallel program. The parallel program executes many iterations of the load cycle shown in the dashed box of Figure 3.

Each load cycle consists of some computation done concurrently by T_{W1} and T_{V1}, followed by a sequential computation (task T_{W2}) performed by the workstation W. This is modelled by the vector machine V doing a blocking receive until it gets a synchronizing message from W, which is sent after the sequential computation is finished. Note that there is no message sent by V to W.

5.2 Heterogeneous Program Model II

This program model consists of a parallel program with three processes – one running on the SIMD

Proceedings of the 1996 Hawaii International Conference on System Sciences (HICSS-29)
1060-3425/96 $10.00 © 1996 IEEE

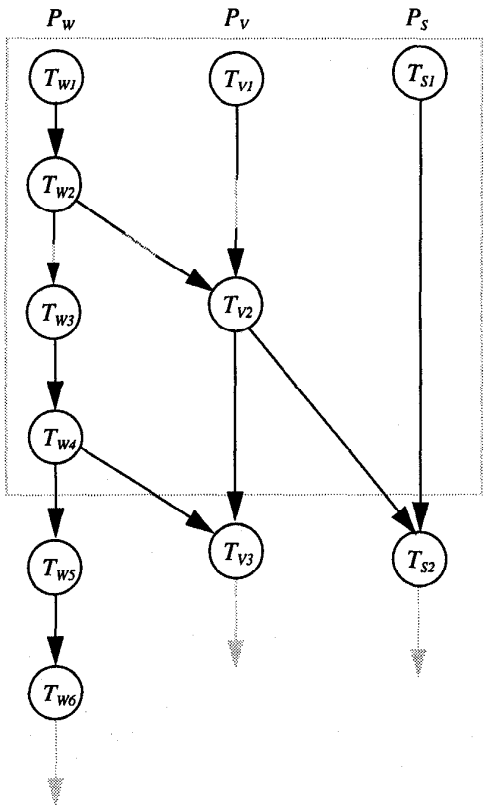

Figure 4. Load cycle of Heterogeneous Program Model II.

machine *S*, one on the vector machine *V* and the third on the workstation *W*. This program extends the first model to capture two levels of bottlenecks. The task on the SIMD machine runs ahead of the one on the vector machine, which in turn is slowed down by the one on the workstation. The parallel program executes many iterations of the load cycle shown in the dashed box of Figure 4.

Table 1 gives details of the two program models assumed in the simulation study.

5.3 Background load conditions

We evaluate the performance of the above two program models under three different background load conditions and compare them with the base cases (see Table 2).

Background load *A* consists of individual edit jobs. All these jobs are executed on the workstation. Background load *B* consists of a mix of individual jobs belonging to several classes. It consists of an equal number of jobs assigned to all three machines. Background load *C* consists of a parallel program obtained by superimposing a random task precedence graph over background load *B*.

It is assumed that all the above load conditions are present in the system before issuing the parallel program and that no jobs arrive after the issue of the parallel program. This assumption is reasonable since most high-performance machines are operated in batch processing mode with very low arrival rate.

5.4 Discussion

We define our speedup relative to the base case as follows:

$$\text{Speedup} = \frac{T_b - T_{lb}}{T_b}$$

T_b : Execution time of the heterogeneous program for the base case.

T_{lb} : Execution time of the heterogeneous program with

Table 1. Details of Heterogeneous Program Models I and II.

Parallel Program Model	Number of processes			Number of messages	Distribution of mean ratios			Service demand in seconds		
	S	*V*	*W*		*S*	*V*	*W*	*S*	*V*	*W*
I	0	1	1	350	–	0.8	0.7	–	0.06	0.2
II	1	1	1	359	0.9	0.8	0.7	0.075	0.06	0.2

Proceedings of the 1996 Hawaii International Conference on System Sciences (HICSS-29)
1060-3425/96 $10.00 © 1996 IEEE

Table 2. Execution times for base case scheduling.

Parallel Program Model	Background Load Condition		
	A	*B*	*C*
I	0.630	0.900	0.530
II	0.709	0.860	0.538

priority-based load balancing for a given parameter set.

Table 2 gives the execution times for the two heterogeneous program models under different background load conditions for the base case scheduling.

Figure 5, speedup vs. decay usage factor, shows that speedup increases gradually with D from D=1 to a value around D=10, when it starts saturating. It is clear that as D increases time taken by a process inducted into the `high priority queue` to leave the same is extended, and hence the process gets better service for a longer time, thereby improving its execution time. Figure 5 also shows that speedup decreases rapidly as T increases from T=1 to T=3. Since T is an integral multiple of the quantum interval for each machine, results were not obtained for a finer variation of T between T=1 and T=2. This rapid decrease in speedup can be attributed to the fact that the rate at which a high priority process comes down to the base priority B is governed by how often the CPU usage field U of the process is decayed.

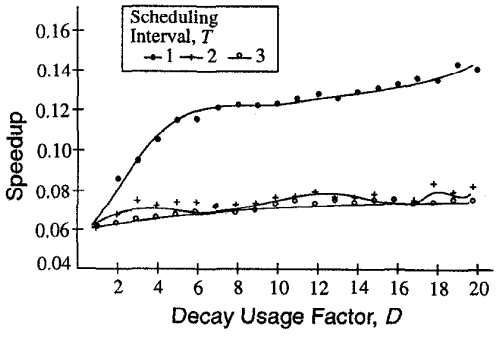

Figure 5. Variation of speedup with decay usage factor D (B = 500, Background Load A) for Program Model I.

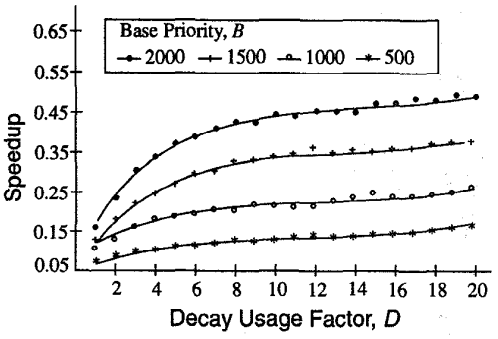

Figure 6. Variation of speedup with decay usage factor D (T = 1, Background Load A) for Program Model I.

From the graphs in Figure 6, we find that speedup increases proportionally as B is varied from B=500 to B=2000. This shows that the time taken for a high priority process to come down to the base priority level is controlled not only by the rate at which the numeric value of its priority increases but also the range of priority values that a high priority process can have. Note that higher numeric value implies lower priority. A process with *priority* = 0 has the highest priority.

Figure 7 shows that the algorithm performs better for Program Model I, when the background load is a parallel program with a random task precedence graph, that is, with load condition C. The performance improvement is the poorest for background load condition A consisting of edit jobs alone. This is attributed to the fact that the edit jobs require better service and are given more priority as compared to computation-bound jobs by the local scheduler which uses the base case scheduling policy (discussed in Section 3).

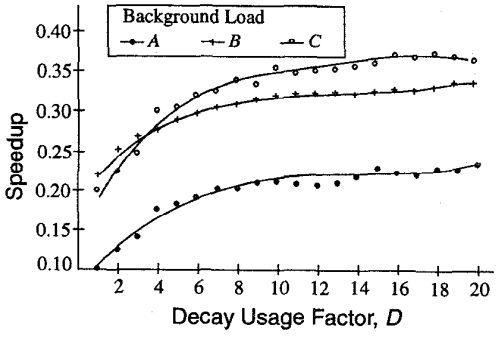

Figure 7. Variation of speedup with decay usage factor D (B = 1000, T = 1) for Program Model I.

Proceedings of the 1996 Hawaii International Conference on System Sciences (HICSS-29)
1060-3425/96 $10.00 © 1996 IEEE

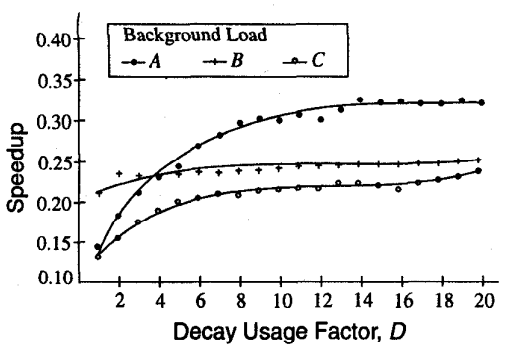

Figure 8. Variation of speedup with decay usage factor D ($B = 1000$, $T = 1$) for Program Model II.

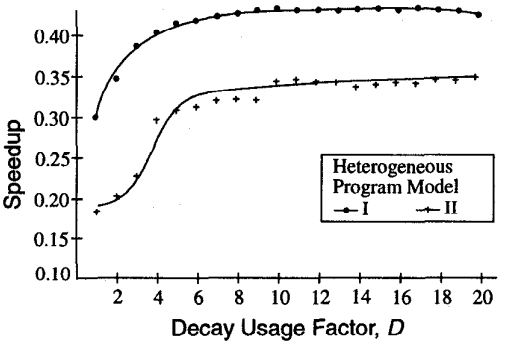

Figure 9. Variation of speedup with decay usage factor D ($B = 2000$, $T = 1$) for Program Model I and II.

It is interesting to observe from Figure 8 that the algorithm performs better for Program Model II, when the background load is a set of edit jobs alone. The performance is poorest for background load condition *C*. This is because of the fact that the effects of the additional level of bottleneck in Program Model II are not felt in the case of background load condition *A* as the vector and SIMD machines are free except for the only tasks they run. In the case of load condition *B* or *C*, these two machines are quite loaded and the additional level of bottleneck results in the poor performance of the algorithm.

From Figure 9, we find that the performance improvement for Program Model II is lesser compared to that for Model I. This is because of the additional dependencies in Model II that require speeding up of execution of the tasks of the vector machine also.

The above results show that the priority-based decay usage load balancing scheme can improve the mean of job response time by 30-60 percent even under the moderate background load condition. Load balancing can still be highly effective when a large fraction of the jobs that would otherwise be eligible for load balancing must be executed locally.

6. Conclusions

In this paper we have presented a priority-based decay usage load balancing algorithm for a heterogeneous computing environment. The algorithm determines the task precedence graph of the parallel jobs dynamically at run-time and assigns appropriate priorities to the processes to resolve the dependencies. The behaviour of the algorithm has been analysed for two different heterogeneous parallel program models under different background load conditions. It is shown that the improvement in the execution time can be gained by (a) changing the rate at which the high priority processes drop down to the `ready queue` managed by the local scheduler, and/or (b) increasing the base priority, thereby increasing the amount of CPU service that a high priority process can get before it is moved down to the `ready queue`.

Instead of using simple round-robin scheduling, our future work may incorporate a more realistic UNIX-like scheduling policy having multiple queues with feedback in the local schedulers of each machine. During context-switches, instead of recalculating the priority and inserting the process into the `high priority queue`, we just add the process to the end of this queue. The priorities of the processes are calculated only every scheduling interval, which in this case need not be an integral multiple of the quantum. This will give us a finer control the scheduling interval *T*.

References

1. R.F. Freund and H.J. Siegel, "Heterogeneous Processing," *IEEE Computer*, Vol. 26, No. 6, June 1993, pp. 13-17.
2. M. Ercegovac, "Heterogeneity in supercomputer architectures," *Parallel Computing*, North-Holland, Vol. 7, 1988, pp. 367-372.
3. V. Donaldson, F. Berman, and T. Paturi, "Program speedup in a heterogeneous computing network," *Journal of Parallel and Distributed Computing*, Vol. 21, 1994, pp. 316-322.

Proceedings of the 1996 Hawaii International Conference on System Sciences (HICSS-29)
1060-3425/96 $10.00 © 1996 IEEE

4. A.A. Khokhar, V.K. Prasanna, M.E. Shaaban, and C.-L. Wang, "Heterogeneous computing: Challenges and opportunities," *IEEE Computer*, Vol. 26, No. 6, June 1993, pp. 18-27.
5. J. Yang, I. Ahmad, and A. Ghafoor, "Estimation of execution times on heterogeneous supercomputer architectures," *Proc. 1993 International Conference on Parallel Processing*, CRC Press, St. Charles, Ill., Aug. 1993.
6. N.Bowen, C.N. Nicolau, and A. Ghafoor, "On the assignment problem of arbitrary process system to heterogeneous computer systems," *IEEE Transactions on Computers*, Vol. 41, No. 3, Mar. 1992, pp. 257-273.
7. A.S. Grimshaw, J.B. Weissman, E.A. West, and E.C. Loyot, "Metasystems: An approach combining parallel processing and heterogeneous distributed computing systems," *Journal of Parallel and Distributed Computing*, Vol. 21, 1994, pp. 257-270.
8. D.L. Eager, E.D. Lazowska, and J. Zahorjan, "Adaptive load sharing in homogeneous systems," *IEEE Transactions on Software Engineering*, Vol. 12, No. 5, May 1986, pp. 662-675.
9. T.L. Casavant and J.G. Kuhl, "A taxonomy of scheduling in general-purpose distributed computing systems," *IEEE Transactions on Software Engineering*, Vol. 14, No. 2, Feb. 1988, pp. 141-154.
10. X. Qian and Q. Yang, "An analytical model for load balancing on symmetric multiprocessor systems," *Journal of Parallel and Distributed Computing*, Vol. 20, 1994, pp. 198-211.
11. A.N. Tantawi and D. Towsley, "Optimal static load balancing in distributed computer systems," *Journal of Association of Computing Machinery*, Vol. 32, No. 2, Apr. 1985, pp. 445-465.
12. G.C. Sih and E.A. Lee, "A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 4, No. 2, Feb. 1993, pp. 175-187.
13. R. Mirchandaney, D. Towsley, and J. Stankovic, "Adaptive load sharing in heterogeneous systems," *Proc. International Conference on Distributed Computing Systems*, 1989, pp. 298-306.
14. K.G. Shin and C.-J. Hou, "Analytic models of adaptive load sharing schemes in distributed real-time systems," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 4, No. 9, July 1993, pp. 740-761.
15. M.H. Willebeek-LeMair and A.P. Reeves, "Strategies for dynamic load balancing on highly parallel computers," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 4, No. 9, Sept. 1993, pp. 979-993.
16. M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, San Francisco, CA, 1979.
17. A.S. Tanenbaum, *Modern Operating Systems*, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1992.
18. A. Ghafoor and J. Yang, "A distributed heterogeneous supercomputing management system," *IEEE Computer*, Vol. 26, No. 6, June 1993, pp. 78-86.
19. J.L. Hellerstein, "Achieving service rate objectives with decal usage scheduling," *IEEE Transactions on Software Engineering*, Vol. 19, No. 8, August 1993, pp. 813-825.

Proceedings of the 1996 Hawaii International Conference on System Sciences (HICSS-29)
1060-3425/96 $10.00 © 1996 IEEE