Load Balancing in a Heterogeneous Computing Environment*

Sridhar Gopal†
University of Wisconsin-Madison
gsri@cs.wisc.edu

Sriram Vajapeyam
Indian Institute of Science, Bangalore
sriram@csa.iisc.ernet.in

Abstract

Heterogeneous distributed computing is the tuned use of a network of machines of diverse architectures and computational power, by directing individual portions of a parallel program to the machine(s) best suited for their execution. Load balancing algorithms for an HCE attempt to improve the response times for parallel programs by adapting the process scheduling policies on individual machines. We propose a priority-based load balancing algorithm, the Priority algorithm, which dynamically adjusts the quality of service for the processes of a parallel program by detecting dependences among them. Process migration, the most successful means to balance load in homogeneous distributed environments, is undesirable in an HCE both because of architectural differences between machines and because it conflicts with the HCE goal of matching code to machines. We use simulation to evaluate the speedup of the algorithm on synthetic parallel programs under different background load conditions.

1. Introduction

Large scale homogeneous distributed computing, which comprises machines of the same architecture and similar computational power, has been adequate for regular applications that spend majority of their execution performing one type of computation. Sorting is an example of such an application. Irregular applications, such as protein sequence matching [12] and climate modeling [10], have a spectrum of computational requirements like large scale data acquisition, preprocessing, matrix computation, scalar processing and data visualization. A *Heterogeneous distributed Computing Environment (HCE)* that comprises diverse machines designed for executing different types of computation is better suited for such applications. An example HCE comprises a SIMD machine (for data acquisition and preprocessing), a vector machine (for highly parallel matrix computation), a general purpose workstation (for scalar processing), and a graphics machine (for visualization). Traditional high performance computing uses a vector supercomputer to execute irregular applications. However, such a high-end machine achieves only a small fraction of its peak performance on parts of the application not suited to it, and hence may not be the right target for irregular applications.

1.1. Heterogeneous computing

Heterogeneous distributed computing is the tuned use of a network of machines of diverse architectures and computational power, by directing individual portions of a parallel program to the machine(s) best suited for their execution. This intelligent use of the machines provides higher speedups for parallel programs and increases the utilization of the available computing power. Heterogeneous computing is now feasible because of the existing diversity of computing resources in local networks and the improvements in high-speed networking technology. Hundreds of institutions are using the Parallel Virtual Machine (PVM) to develop parallel programs that use a heterogeneous set of machines.

Heterogeneous Computing Environment (HCE) must not be confused with heterogeneous environments with clusters of workstations that use opportunistic load balancing techniques. Such a cluster typically comprises machines with the same computational facilities though they might be configured differently and have non-uniform connectivity to the network. parallel programs, in a cluster environment, are *not* targeted at the individual architectures of the different workstations. In this report, we focus on load balancing in HCE.

1.2. Scheduling in HCE

In a homogeneous distributed computing environment, the scheduler typically consists of two components, *viz.*, the load distributor and the local scheduler. The responsibility of the load distributor is to improve performance by correcting anomalies that arise in distribution of the load among nodes by process transfer/migration, while the local scheduler is the one that allocates local resources among the resident processes. Two distinct distributed scheduling strate-

*This paper was originally published with incorrect authorship in HICSS-29, 1996.

†Work done while at the Indian Institute of Science, Bangalore.

gies to improve performance have been evolved [2]. Load sharing algorithms attempt to improve the performance of a set of machines (and their processes), by assuring that no node lies idle while processes wait for service. Load balancing algorithms, on the other hand, go a step further by striving to equalize the workload among nodes (even when they are not idling). Process migration has been very successfully employed to improve response time for parallel programs in a homogeneous computing environment.

Consider an HCE that comprises a set of machines, each one belonging to a unique architectural class and possibly representing a pool of machines of its class (i.e., each machine could represent a homogeneous distributed system). parallel programs are partitioned into processes, where each process spends the majority of its execution in one type of computation suited to a particular architecture. Scheduling these parallel programs consists of three parts, *viz.*, assignment, local scheduling and load balancing. Assignment or mapping comprises matching a process to a machine (or the pool it represents) that is best suited for its execution. Local scheduling on each machine may itself be a distributed scheduling policy in the pool it represents. In an HC environment, load sharing algorithms are not applicable because a machine is idle only if there are no processes assigned to it (or the pool it represents) or those assigned are not ready to execute. Migrating processes across architectural classes to share the load is not desirable and in fact, against the goal of executing a portion of application on the best suited architectural class.

1.3. Load balancing in HCE

As attempts at process migration across heterogeneous architectures have been abortive, it is challenging to define load balancing in an HCE. We define load balancing in an HCE as follows:

> To increase the utilization of the machines, the local scheduler in each machine not only uses information about local resources, but also uses the current global scenario of the system; in contrast, the local scheduler in a homogeneous computing environment is the same as in an uniprocessor (save for minor modifications). The latter acts *independent* of the higher level load distributor and does not use any global information.

We use an example to illustrate the opportunities for load balancing in an HCE. Consider an HCE that comprises two machines, A and B. Let PP be a parallel programs that has two processes P_a and P_b, assigned to A and B respectively. If the load on A is high and if P_b is the only process on B, it is possible for P_b to wait for P_a to finish its computation cycle and send in a synchronization message that enables P_b to proceed. If P_b waits for P_a often, machine B could spend a significant amount of time idling. Attempts at determining such dependences between processes of a parallel program amounts to finding its *task-precedence* graph. Also, this graph has to be determined dynamically as dependences are not always known ahead of run-time. Even if dependences can be determined statically, either the programmer or compiler has to specify the graph in a form that can be used by the runtime system. In this paper, we propose a priority-based dynamic load balancing algorithm, the *priority* algorithm, and evaluate the speedup obtained on synthetic workloads by simulation.

1.4. Related work

Heterogeneous computing opens up new challenges and opportunities in fields such as parallel processing, design of algorithms for applications, partitioning and mapping of application tasks, interconnection network technology and the design of heterogeneous concurrent programming environments [3, 8]. Various load balancing algorithms have been proposed for homogeneous concurrent environment and a taxonomy of algorithms can be obtained from [9, 5]. To the best of our knowledge there is no previous work on load balancing in a HCE. The priority algorithm proposed in this paper would be classified as a dynamic, distributed and cooperative algorithm by the taxonomy in [9, 5]. This algorithm was first reported in [4] in 1994 and represents early work in this area.

1.5. Overview

Section 2. discusses our characterization of workloads, and the experimental HCE that we simulate; it also gives details about the simulation methodology. Section 3. proposes the priority algorithm and section 4. describes the two parallel program models we use to evaluate the algorithm. The performance of the algorithm is studied with different background load conditions on the HCE. We then present results of the evaluation and conclude in section 5..

2. Framework

Heterogeneous computing has been a recent development and load balancing in such an environment has not been studied well [3, 8]. As there are no standard workload models and assumptions, we use synthetic workloads based on a reasonable characterization we present in this section. We draw heavily from the large volume of publications on homogeneous distributed systems. We also discuss the details of the simulation method we use to study the performance of the priority algorithm.

2.1. The experimental HCE

The experimental HCE consists of three machines *viz.*, a workstation, a vector machine and a SIMD machine. Each machine could represent a pool of machines of the same architectural class, and load balancing issues within a pool are to some extent modeled by process scheduling on the single machine. Each machine in the HCE executes single process jobs as well as processes of parallel programs. We model the network delay in transmitting messages between machines. We assume a maximum lag of one tenth of the least[1] quantum interval among all the machines in the system, for the smallest size message possible in the HCE. This assumption is necessary to simplify the implementation of the simulator as will become clear from its description.

The base case local scheduler in each machine uses the round robin process scheduling policy. Round robin scheduling is typically done in uniprocessor systems using two different queues, namely, the IO queue and the CPU queue. To make the base case local scheduler aware of parallel programs, we add a Communication queue that comprises ready processes that have recently received messages from their remote peer processes. Such a process is provided higher priority than those in the IO queue or the CPU queue to help resolve dependences as quickly as possible. Traditional UNIX-like scheduling is likely to slow down parallel programs by queuing critical parallel program processes behind serial jobs and less-critical parallel program processes. The quantum size and the clock frequency of each machine is specified as a part of the system configuration information supplied to the system. Table 1 gives details of the HCE used in this study.

Table 1: HCE configuration

Machine	Clock frequency (MHz)	Quantum (clock cycles)
SIMD, S	400	40000
Vector, V	500	50000
Workstation, W	150	15000

2.2. Workload characterization

We start with the task-precedence graph, a common representation, of a parallel program. Each node in this directed graph represents a task, a unit of computation, and each edge represents a dependency from the node at its tail to the node at its head. A task can start execution only after all its parent tasks have completed. An example graph is shown in Figure 1. Tasks with similar code structure are grouped together to form a process of the parallel program. In contrast, traditional parallel program partitioning results in a set of processes, each of which performs computation with multiple varying code structures. A process is assigned to a machine that is best suited for executing the predominant code structure of its component tasks. Tasks with different code structures are differentiated by boxes, circles and triangles in the figure.

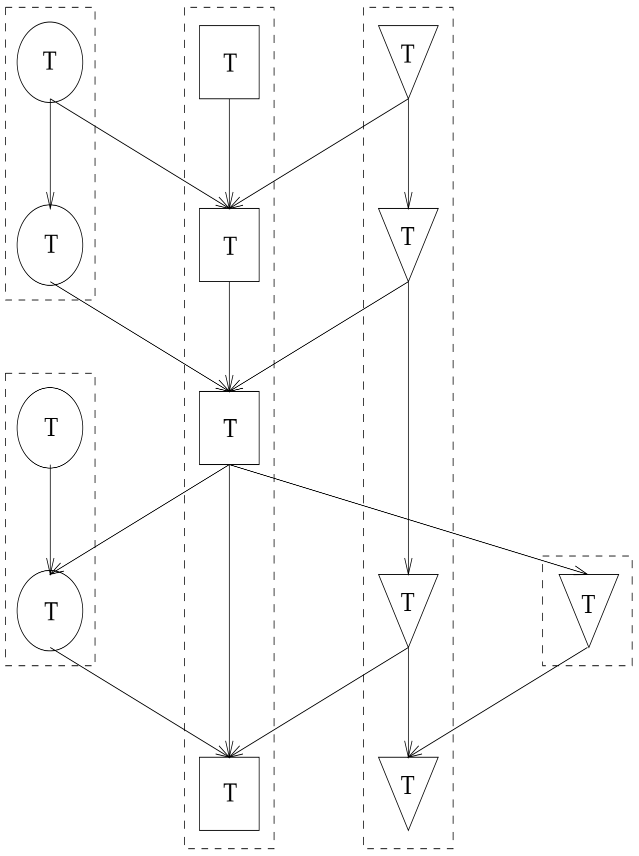

Figure 1: Workload characterization - an example

Each process of a parallel program is given a unique name and dependences between the component tasks are modeled by passing messages between the processes. The message sends are assumed to be non-blocking; a process starts executing the next assigned task, if any, after sending a message. The computation performed by a task is described by alternating two independently and identically distributed exponential sequences, one that models the *CPU* time and one that models the *IO wait* time. For IO-bound jobs, the ratio of the mean of the *CPU* time sequence to that of the *io wait* time sequence is distributed between 0.025 and 0.45. For CPU-bound jobs, this ratio varies between 0.65 and 0.95. The length of these sequences and the mean values depend on the total service demand of the parallel program and its parallelism.

[1]In this study, due to simulation-time constraints, the quantum interval has been kept the same for all machines.

```
machine  V                  machine  W
name     PPa.1              name     PPa.2
block    6011               run      554
run      7239               block    50272
block    6008               run      495
receive  W PPa.2            block    7022
run      14528              send     V PPa.1 Msg
block    1255               receive  V PPa.1
send     W PPa.2 Ack        run      1042
```

Figure 2: A sample parallel program

In this work, we keep the service demand (in clock cycles) of all processes in a parallel program to be the same. The two exponential sequences are interspersed with sends and receives that are used to exchange messages (which can be either synchronization messages used to model the task dependences or partial results being exchanged) between processes. Figure 2 gives an example parallel program comprising two processes. The *run* and *block* instructions specify the amount of time (in clock cycles) a task executes or blocks. The *send* and *receive* instructions specify the source or destination process and the architectural class of the process (in our model, an architectural class maps to a single machine). The send instruction also specifies the message to be sent to the destination process. For every send, there is a matching receive done by the destination process.

2.3. Method

We simulate the execution of the above workload in parallel over a network of machines. We model each machine in the HCE by a UNIX process, the proc_mgr. A front-end process, the master, initiates these proc_mgrs on different machines in the computing network used for simulation. A job is submitted to the master and based on the process assignment information submitted with the job, the master splits it into different processes and issues them to the different proc_mgrs for execution.

As it is unrealistic to assume a global clock to time-stamp messages in a distributed environment, synchronization protocols for distributed simulation such as the Chandy-Misra protocol or the Time-warp protocol [1, 7] have been proposed. To ease implementation of the HCE simulator, we use a custom synchronization protocol based on barriers between the proc_mgrs. Messages received by a proc_mgr at a barrier are those sent by the other proc_mgrs since the previous barrier. This delay in the message transmission is used to model a random communication overhead in the HCE. Like most of the literature on distributed scheduling, we do not account for the overhead involved in context-switches, scheduling, and the load balancing algorithm itself.

3. The Priority Algorithm

The *priority* algorithm tries to improve the response time for parallel programs by determining the task-precedence graph dynamically. It is a scalable, distributed and cooperative algorithm. Each machine in the HCE comprises a load balancing agent as part of the local scheduler that implements the priority algorithm.

As we have modeled dependences with a set of non-blocking sends and blocking receives, whenever a process P_a blocks for a process P_b, the best option to reduce the waiting time of P_a is to schedule P_b without preemption until the dependency is removed, that is, until the message is sent to P_a. An analogy can be drawn from the scheduling of IO-bound processes on UNIX. A UNIX scheduler gives higher priority to processes that have recently performed I/O over those that are CPU-bound. A UNIX process can be modeled as a parallel program consisting of two processes, one that performs I/O alone and another that executes on the CPU alone. When an IO-task finishes execution on an I/O processor, it waits for a compute-task to be serviced on the CPU and vice versa. To enable the IO-process to resume execution earlier, the UNIX scheduler increases the priority of the compute-process on the CPU.

The situation in HCE is quite different in that we do not have two processors, but several machines and available parallelism in jobs can be more than two. Conflicts arise in situations when two processes in a single machine have to be scheduled simultaneously to quickly resolve two dependences. The priority algorithm addresses such situations by using a High Priority (HP) queue of ready processes on top of the ready queue managed by the local scheduler. We use the convention that higher numeric value for priority implies lower priority. The HP queue contains processes arranged in the descending order of their priorities.

3.1. Definition

The load balancing component of the distributed scheduling consists of a load balancing agent in each of the machines. Each of these load balancing agents performs the following:

- Whenever a process P_a blocks for a message from a process P_b, a message is sent to the remote machine, where P_b executes, requesting it to speed up the execution of P_b.

- On receipt of this message, the load balancing agent in the remote machine enhances the priority of process P_b. If P_b is already present in the ready queue, it is moved up into the HP queue immediately; otherwise, it is added to the HP queue once it becomes ready.

- Processes in the HP queue are scheduled before any process in the ready queue managed by the local scheduler. As a process executes, its usage factor (U) increases depending on the duration it uses the CPU.
- The priority of processes in the HP queue is reduced to the base level priority, B, by the *Decay Usage Scheduling* algorithm [6]. Any process having a lower priority than B leaves the HP queue and joins the ready queue.
- The Decay Usage scheduling algorithm has three parameters given by:
    a. D, the decay (CPU) usage factor,
    b. T, the scheduling (or decay) interval when U is decayed using D, and
    c. R, the amount by which U is increased for a process per clock cycle used. This parameter is different for each machine and in this study we have made it proportional to the clock frequency of the machines 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 HP range is context-switched, its priority is updated using the formula given below and it is re-inserted into the HP queue.

$$\text{New priority} = \text{Old priority} + (\text{decayed})\text{CPU usage}$$

- After every T quanta, the load balancing agent decays the CPU usage of all processes whose priorities are above the base priority, B. Hence, the decay operation is done once every T quanta and not once every T context-switches.

4. Evaluation

We use two parallel program models to evaluate and study the performance of the priority algorithm.

4.1. Parallel program model I

The first model comprises a parallel program with two processes, one on the vector machine (V) and the other on the workstation (W). The model captures a realistic situation: a process on a general purpose machine like a workstation, which is heavily loaded, is the bottleneck in improving the response time of the parallel program. This model executes many iterations of the load cycle shown in dashed box in Figure 3.

Each load cycle comprises some computation done concurrently by both the processes, followed by a sequential computation performed by P_W. The sequential computation is modeled by a blocking receive done by P_V until it gets a synchronizing message from P_W, which is sent after the sequential computation is finished. This model is very different from a *fork-join* model used by [11] in that P_W proceeds after sending the synchronization message without waiting for P_V to finish its load cycle.

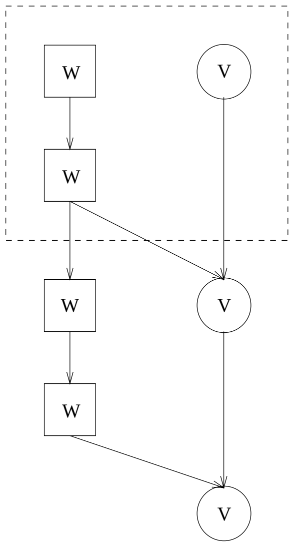

Figure 3: Load cycle of parallel program model I

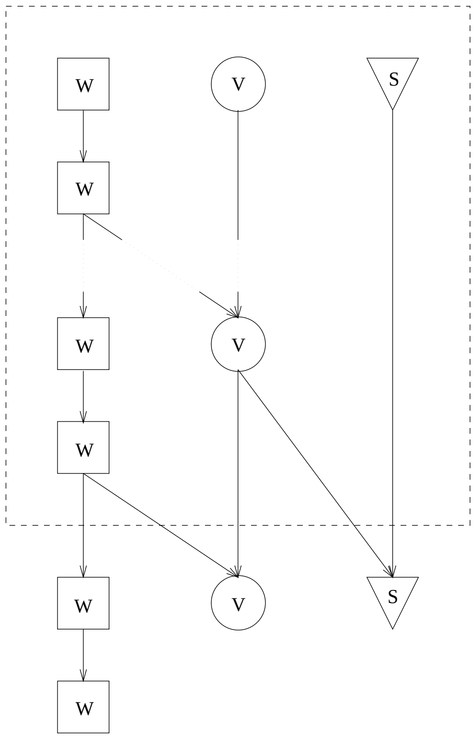

Figure 4: Load cycle of parallel program model II

Table 2: Details of parallel program models I and II

Parallel program model	Number of tasks			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

Table 3: Details of background load conditions

Background Load condition	Number of tasks			Number of messages	Distribution of mean ratios		
	S	V	W		S	V	W
A	0	0	6	0	-	-	0.45
					-	-	0.37
					-	-	0.29
					-	-	0.13
					-	-	0.09
					-	-	0.04
B	6	6	6	0	0.99	0.96	0.89
					0.88	0.75	0.63
					0.65	0.47	0.46
					0.54	0.30	0.29
C	6	6	6	518	0.32	0.18	0.14
					0.23	0.05	0.04

4.2. Parallel program model II

This model comprises a parallel program with three processes: one on the SIMD machine (S), one on the vector machine (V) and a third on the workstation (W). It extends the first model to capture two levels of bottlenecks. The process on the SIMD machine runs ahead of the one on the vector machine, which in turn runs ahead of the one on the workstation. The parallel program executes many iterations of the workload cycle shown in dashed box in Figure 4. Table 2 gives details about the two parallel program models.

4.3. Background load models

We evaluate the priority algorithm for the above two parallel programs under three different background load conditions. It is assumed that the HCE is in a steady state of background load before the parallel program is issued and that no jobs arrive after the parallel program is issued. This assumption is reasonable since most often high-end machines are operated in batch processing mode with the arrival rate being very low. Care has been taken to see that the background loads are sufficiently long not to affect the response time of the parallel program models due to startup and finishing overheads.

Background Load A comprises individual edit jobs. All these jobs are executed on the workstation, which is a general purpose machine unlike the vector or the SIMD machine. This background load is the simplest of all the three loads that we consider as it specifies no load on the vector and the SIMD machines. Background Load B consists of a mix of individual jobs belonging to various classes. It consists an equal number of jobs assigned to all the machines. Background Load C is a parallel program obtained by superimposing a random *task-precedence* graph over background load B. Table 3 summarizes the background loads.

4.4. Results

In this section, we present the simulation results for the two parallel program models under different background load conditions for various parameter sets of the priority algorithm. We define *Speedup* as follows:

$$\text{Speedup} = \frac{R_b - R_{lb}}{R_b}$$

where,

R_b	Response time for the parallel program alone for the base case.
R_{lb}	Response time for the parallel program alone with load balancing for a given parameter set.

Table 4 gives the response times of the two parallel program models under different background load conditions for the base case scheduling. We draw the following conclusions from the results obtained and attempt to explain the behavior of the algorithm.

Table 4: Response 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

4.4.1. Variation of speedup with decay factor. From Figure 5, we find that speedup increases gradually as D increase from $D = 1$ to around $D = 10$, when it starts to saturate. As D increases the time taken by a process inducted into the HP queue to leave the same is extended and hence the process gets better service for a longer time, thereby improving its response time.

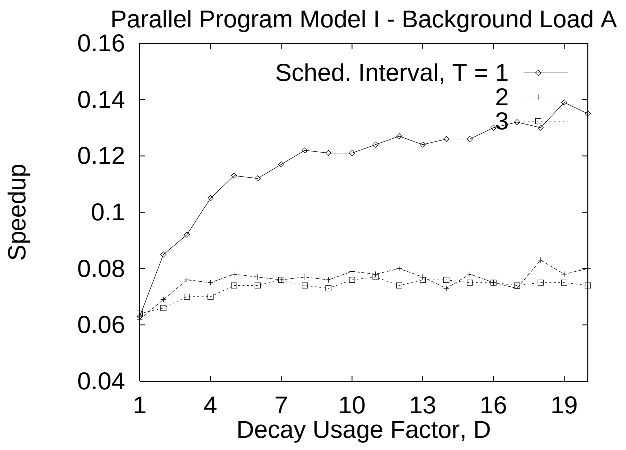

Figure 5: Variation of speedup for different T ($B = 500$)

4.4.2. Variation of speedup with scheduling interval. From Figure 5, we find that speedup decreases rapidly as T increases from $T = 1$ to $T = 3$, when it starts saturating. As 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 increase in the speedup can be attributed to the fact that the rate at which the priority of a process in the HP queue reduces to the base priority B is governed by how often the CPU usage field U of the process is decayed.

4.4.3. Variation of speedup with base priority. From Figure 6, we find that speedup increases proportionally as B increases from 500 to 2000. As the time taken for the priority of a process in the HP queue to reduce to the base priority level is controlled not only by the rate at which its value changes, but also by the range of priority values.

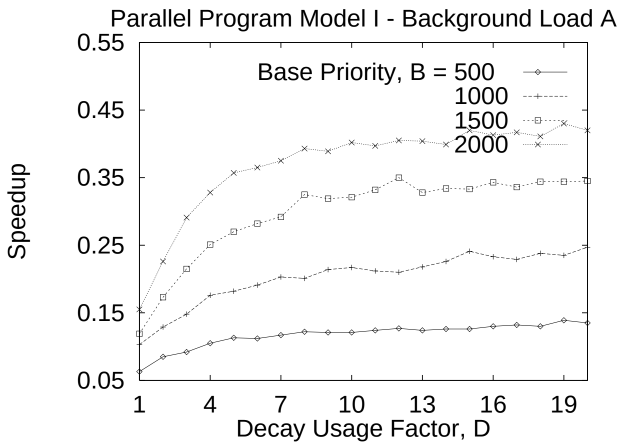

Figure 6: Variation of speedup for different B ($T = 1$)

4.4.4. Variation in speedup for different background load conditions. From Figure 7, we observe that the priority algorithm performs better for parallel program model I, when the background load is a parallel program with a random task-precedence graph (load condition C). The performance improvement is poorest for a background load consisting of edit jobs alone (load condition A). Load condition A comprises edit jobs that require better service and hence are given more priority as compared to the compute bound jobs by the local scheduler which uses the base case scheduling policy detailed in section 2.1..

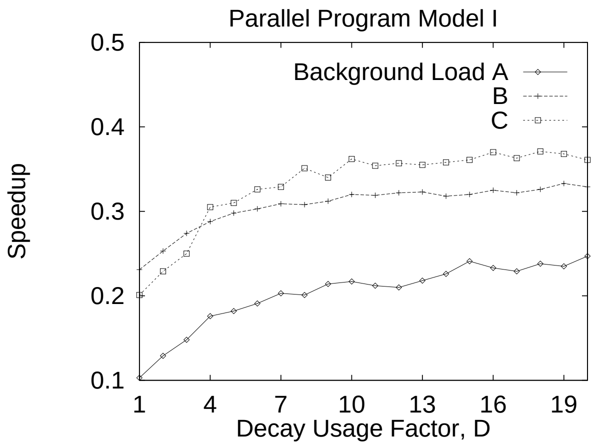

Figure 7: Variation of speedup for different load conditions ($B = 1000, T = 1$)

From Figure 8, we observe that the algorithm performs better for parallel program model II, when the background load is a set of edit jobs alone. The performance improve-

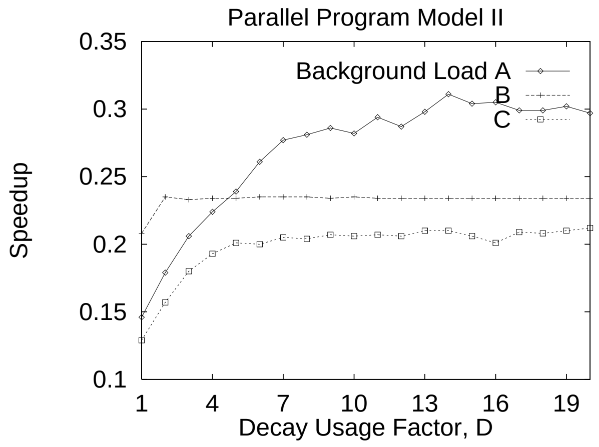

Figure 8: Variation of speedup for different load conditions ($B = 1000, T = 1$)

ment is poorest for background load condition C. The effects of the additional level of bottleneck in model II are not felt in the case of load condition A as the vector and SIMD machines are free except for the only task they run. However with either load condition B or C, these two machines are less idle and the additional level of bottleneck results in the priority algorithm performing poorly.

4.4.5. Variation in speedup between the two parallel program models. From Figure 9, we find that the performance improvement for parallel program model II is lesser compared to that for model I. The additional dependences in model II require speeding up of execution of the process on the vector machine also.

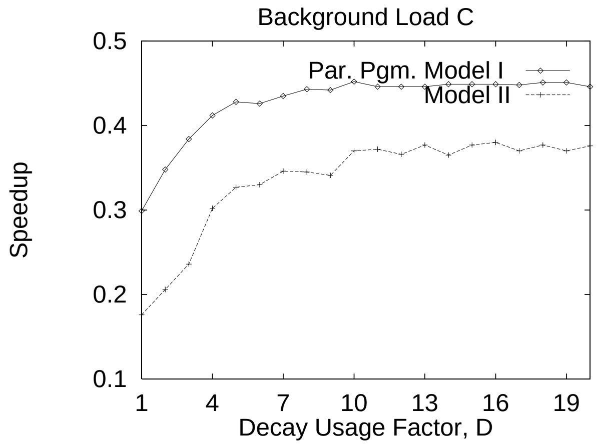

Figure 9: Variation in speedup for models I and II ($B = 2000, T = 1$)

5. Conclusion

Heterogeneous distributed concurrent computing intelligently uses the available machines in a high-end environment by matching code portions (tasks) of a parallel program with the machine(s) best suited for their execution. This intelligent management of resources not only improves their utilization but also improves the response time for the individual parallel programs by executing each task as fast as possible. Dependences among the tasks of a parallel program are enforced using synchronization messages. Partitioning a parallel program in this manner may not be beneficial if the different tasks are not load balanced. Also, interference from other workloads and other dynamic events could upset any balance thereby introducing a stall time corresponding to each dependence.

We proposed a dynamic load balancing algorithm, the Priority algorithm, that raises the priority of a process whenever one of its peer processes on a remote machine blocks waiting for a synchronization message from the former. As these messages correspond to dependences among the processes of a parallel program detecting these blocks amounts to determining the task-precedence graph of the parallel program. The priority algorithm uses a decay usage scheduling policy similar to the one used by traditional UNIX-like systems.

We evaluated the performance of the priority algorithm for two different parallel programs under different background load conditions. The two parallel programs were synthesized due to unavailability of workloads; they model I/O wait time which is typically not included in analytical models that are used to evaluate load balancing algorithms. We implemented a parallel simulator of an experimental HCE with three machines that also models the network delay involved in message communication.

References

[1] K.M. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computation. *Communications of the ACM*, 24(11):198–206, April 1981.

[2] D.L. Eager, E.D. Lazowska, and J. Zahorjan. Adaptive load sharing in homogenous distributed systems. *IEEE Transactions on Software Engineering*, SE-12(5):662–675, May 1986.

[3] R.F. Freund and H.J. Seigel. Heterogenous processing. *IEEE Computer*, 26(6):13–17, June 1993.

[4] Sridhar Gopal. Load balancing in a heterogeneous concurrent environment. Master's thesis, Indian Institute of Science, Bangalore, INDIA, January 1994.

[5] A. Goscinski. *Distributed Operating Systems: The Logical Design*. Addison-Wesley Publishing Company, 1 edition, 1992.

[6] J.L. Hellerstein. Achieving service rate objectives with decay usage scheduling. *IEEE Transactions on Software Engineering*, 19(8):813–825, August 1993.

[7] D. Jefferson. Virtual time. *ACM Transactions on Programming Languages Systems*, 7(3):404–425, July 1985.

[8] A.A. Khokhar, V.K. Prasanna, M.E. Shaaban, and Cho-Li Wang. Heterogenous computing: Challenges and opportunities. *IEEE Computer*, 26(6):18–27, June 1993.

[9] P.E. Kreuger. Distributed scheduling for a changing environment. Technical Report 780, Computer Sciences Department, University of Wisconsin - Madison, June 1988.

[10] Carlos R. Mechoso, John D. Farrara, and Joseph A. Spahr. Achieving superlinear speedup on a heterogeneous, distributed system. *IEEE parallel and distributed technology: systems and applications*, 2(2):57–61, Summer 1994.

[11] R. Nelson and D. Towsley. A performance evaluation of several priority policies for parallel processing systems. Technical Report 91-32, Computer and Information Sciences Department, University of Massachusetts at Amherst, May 1991.

[12] H. Nicholas, G. Giras, V. Hartonas-Garmhausen, M. Kopko, C. Maher, and A. Ropelewski. Distributing the comparison of DNA and protein sequences across heterogeneous supercomputers. In Anne Copeland MacCallum, editor, *Proceedings of the 4th Annual Conference on Supercomputing*, pages 139–149, Alburquerque, NM, USA, November 1991. IEEE Computer Society Press.