Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

AN EXPERIMENTAL STUDY ON THE PERFORMANCE OF THE SPACE-TIME SIMULATION ALGORITHM

R. Bagrodia
K. M. Chandy
W.-T. Liao

August 1991
CSD-910061

An Experimental Study on the Performance of the Space-Time Simulation Algorithm*

Abstract

Most performance studies for parallel simulations only exploit spatial parallelism in the execution of a simulation model. This paper describes a performance study that used space parallelism, time parallelism as well as a combination of the two decomposition strategies to reduce the execution time for simulation models of simple stochastic benchmarks. Also, the time-parallel implementation of the model used a simple strategy to correct estimated future states of a process that effectively eliminated rollbacks. The paper presents speedup curves for a number of configurations and also analyzes the major cost components of the parallel simulation algorithm.

1 Introduction

A number of algorithms have been proposed for the execution of simulation programs on parallel architectures. Many empirical studies have been devised in the past few years to evaluate the performance of the algorithms for both deterministic and stochastic applications [RMMS87, Fuj88a, SS89, Fuj88b, JBWea87, CS89a, BL91, WL90, SS90]. With a few exceptions, most simulation algorithms and performance studies have only exploited spatial decomposition for parallel execution of the model: the physical system to be modeled is subdivided into smaller subsystems, each of which is modeled by a logical process. The logical processes are executed in parallel; when the synchronization overheads for the parallel processes are less than the parallelism in the model, reasonable speedups may be obtained. Recent simulation algorithms have suggested that some applications may also benefit by decomposition in the temporal domain such that a model is simultaneously executed over different time-intervals[CS89b, GL90]. In this paper, we examine both spatial and temporal decomposition of simulation models within the framework of the space-time simulation algorithm and present the results of an experimental study

*This research was partially supported by NSF under grant no. CCR 9157610 and by ONR under grant no. N00014-91-J-1605 that evaluated the effectiveness of the decompositions in reducing the execution time of a set of stochastic benchmarks.

The paper also studies the effect of state correction as an alternative to rollback and recomputation and examines its effect on the performance of optimistic algorithms. Parallel simulation algorithms typically subdivide the space-time region represented by the execution of the model and execute the sub-regions (which will henceforth be referred to as *areas*) simultaneously. The execution must be synchronized periodically to verify that the dependencies in the model have been correctly reflected in its execution. Consider two *areas*, r_a and r_b, such that execution of r_b depends on some event in r_a. If the two regions are executed concurrently, the execution of r_b will be incorrect. Rather than recompute r_b as required by traditional optimistic techniques, it may sometimes be possible to correct the final state of r_b so as to explicitly include the effect of the influencing events from the execution of r_a. State correction techniques for algorithms based on spatial decomposition have been described for the Maisie simulation language[BL90] and was also proposed by Lin and Lazowska[LL91] for time-parallel execution of simulation programs. To the extent that such correction is feasible, it may have a significant impact on improving system performance by reducing rollback (and indirectly, state saving) overheads for optimistic techniques.

For the stochastic benchmarks that were studied, the space-time algorithm yielded significant speedups using a spatial decomposition. However, temporal decomposition is harder to exploit, particularly if the implementation of the algorithm is transparent to the analyst. Temporal decomposition with rollbacks and recomputations do not yield significant benefit in reducing execution time of the model. To exploit temporal parallelism, it must be possible to predict the state of the model at specific times in the future. For most models, these predictions are invariably inaccurate, causing the corresponding computations to be recomputed when the simulation time in the model reaches the future values. In the absence of spatial parallelism, this yields no speedups and the overhead may actually cause the parallel implementation to be slower than the sequential one. However, the study shows that if temporal parallelism is used together with state correction, significant speedups are possible for the stochastic benchmarks that were used in the study. Finally, the paper also examines the practicality of combining space and time parallelism in the execution of stochastic models. Once again, for applications where state correction is feasible, significant speedups were obtained.

The rest of the paper is organized as follows: the next section gives a brief description of the space-time algorithm. Section 3 describes the experiments. Section 4 is a brief discussion of implementation issues for the space-time algorithm. Section 5 describes the performance results of the experimental study and section 6 is the conclusion.

2 Space-Time Algorithm

The Space-Time framework[CS89b] suggests that the execution of a simulation model be subdivided into a number of regions where each region represents the behavior of the system over some interval of the entire space-time region. Figure 1 shows a possible sub-division for a simple model that simulates a 1-dimensional system. A logical process (*lp*) is assigned to compute the behavior of each region. Each *lp* computes the behavior of its region using an iterative relaxation algorithm that is briefly described in the remainder of this section.

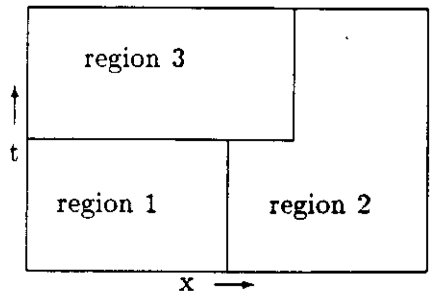

Figure 1: Space-Time decomposition

Let H be the upper bound on the time for which the system is to be simulated. Let $p_i^{x,y}$ refer to the lp responsible for the simulation of some physical process in the interval $[t_x, t_y)$, $t_x < t_y$; exactly one lp computes the behavior of a physical process for every t in [0,H]. A *precedence* relation, symbolized by $\leadsto$, is defined between two lp, where $p_i^{x,y} \leadsto p_j^{x,y}$ if and only if the state of $p_j^{x,y}$ depends on the state of $p_i^{x,y}$ or on some message received from $p_i^{x,y}$. If $p_i^{x,y} \leadsto p_j^{x,y}$, we say that $p_i^{x,y}$ is a *predecessor* of $p_j^{x,y}$ and $p_j^{x,y}$ is a *successor* of $p_i^{x,y}$. Note that although the exact predecessor or successor set for an lp cannot be determined a priori, a loose upper bound on these sets can typically be determined (a trivial bound is the entire set of lp in the system).

Given that the preceding set of lp is executed on a distributed architecture, the correct state of each lp is computed by using the following iterative strategy: given some state for its predecessor lp, an lp computes an *estimate* of its final state. During this computation, it generates a (possibly empty) sequence of messages for each of its successors. The message sequence is sent to each successor after a process has computed its final *estimated* state. When a process gets a message sequence from one of its predecessors that is different from the one it received in its previous iteration, the process recomputes its behavior. This procedure is repeated until

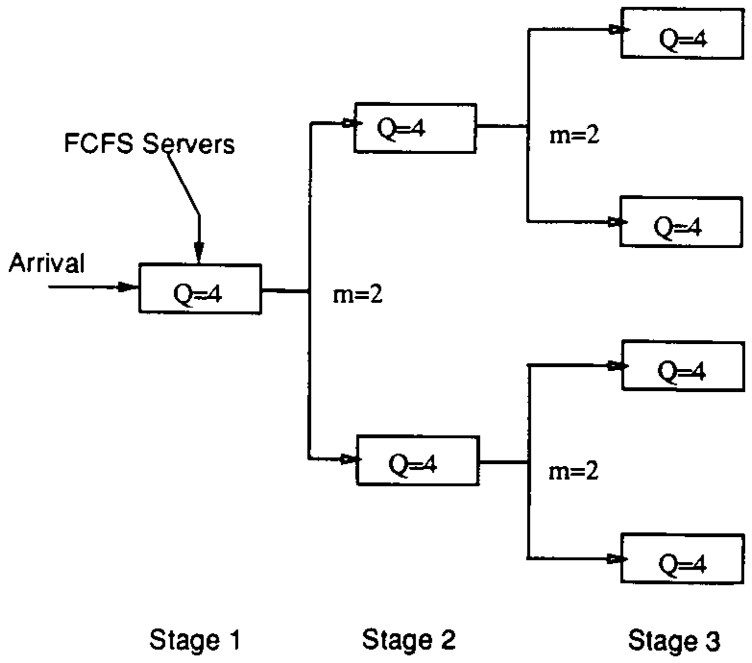

Figure 2: Feed Forward Network ($S = 3, m = 2, Q = 4$)

eventually the computation reaches a fixed-point where further execution of any process does not change its state, and the computation is said to have converged. A complete description of the algorithm and sufficient conditions for the convergence of the computation may be found in [CS89b].

3 Experiments

The experimental study used three types of queueing networks that have previously been used in performance studies of parallel algorithms. The first example is a feed-forward network (FFN) of the type shown in figure 2. This network consists of a number of stages, S, such that jobs exiting from stage s_i are fed into the servers in stage s_{i+1}. Each server in stage s_i has m successors, and a job exiting from the server may go to any of the m servers with equal probability. Jobs exiting from servers in the last stage exit the system. The figure shows the network with 3 stages where a server in each stage has two successors (S=3, m=2). In addition to S and m, other parameters of the system include the job arrival rate (λ) and the service rate at each server (μ), the total period of simulation T (or equivalently J, the total number of jobs that were serviced), and N, the number of processors used in the parallel implementation.

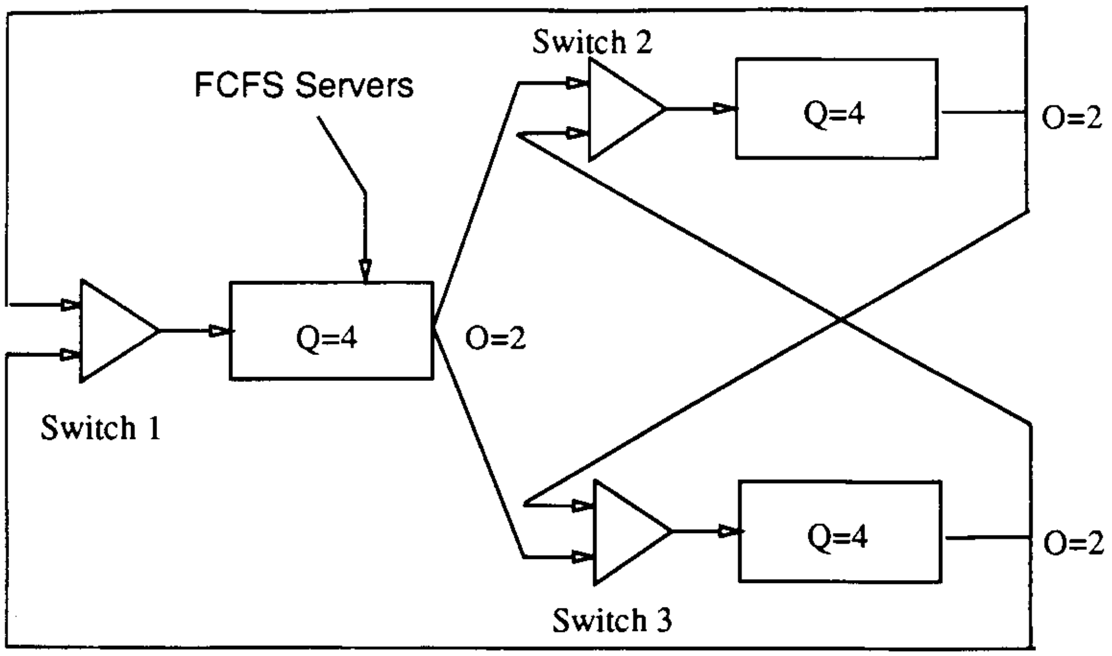

Figure 3: Closed Queueing Network ($N = 3, O = 2, Q = 4$)

The next benchmark is a closed queueing network (henceforth referred to as CQNF) that consists of N switches. Each switch is a merge process that feeds a tandem queue of Q fifo servers. A job that arrives at a queue is served sequentially by the Q servers and then goes to a fork process that routes arriving jobs on any one of its O ($O \leq N$) outgoing paths to a destination switch. The service time of a job at a server is generated from a negative exponential distribution, where all servers are assumed to have an identical mean service time. Each switch is initially assigned J jobs that continuously traverse the network for the specified simulation period. Figure 3 displays an instance of the network with N=3, O=2 and Q=4.

The third benchmark (henceforth referred to as CQNP) is similar to CQNF except that a job may belong to one of two classes: high or low, where jobs in the first class have a higher priority than those in the second class. Each fifo server is replaced by a priority preemptible server. Job classes introduce a additional parameter R, that refers to the percentage of jobs that have a higher priority. The preemptive nature of this application though increases the chances of rollbacks in the system and may benefit from a more frequent checkpointing of the system state.

4 Implementation: Simulation Algorithm

In this section, we first consider the implementation of the basic space-time simulation algorithm using spatial decomposition and rollback and recomputation to correct inaccurate predictions. We subsequently consider its implementation using temporal decomposition with state correction.

In implementing space-time with spatial decomposition, the run-time system must perform the following major tasks:

- subdivision of the space-time area into smaller regions.
- periodic checkpointing of each region.
- detection of timing anomalies in the processing order of events by a region and recomputation of the model from a checkpointed state to correct the anomaly.
- convergence detection to determine the time to which the simulation has been computed correctly.

Many different alternatives exist for each of the preceding tasks. The choice among these alternatives is not always straightforward and specific decisions may have a significant impact on the performance of the simulation algorithm. For example, the system may be checkpointed after every event or only for selected events. Frequent checkpointing increases state-saving overheads but may decrease the recomputation overheads when the system is rolled back. Similarly, the subdivision of the system into regions that are simulated concurrently can also impact the completion time of the simulation. In a subsequent section, we examine two extreme decompositions for a queueing network: the traditional decomposition where a single *lp* is used to sequentially compute the state of a physical component at different points in time and the time-parallel decomposition where the entire system is simultaneously simulated for different time-intervals.

The Maisie simulation language has been implemented on a Symult S2010 multicomputer using the space-time algorithm. Maisie is a message-based simulation language whose primary goal is to provide a notation that allows a programmer to describe the simulation model independently of the underlying algorithm. A Maisie program is a collection of entities, where each entity models one or more physical processes. Events in the physical system are modeled by message exchanges among the corresponding entities. A programmer designs a Maisie model of the physical system by describing the physical system using an appropriate set of entities. The simulation model may subsequently be executed using a sequential simulation algorithm, a conservative algorithm based on conditional events or the space-time algorithm. With respect to the space-time algorithm, the algorithm-independence feature of Maisie requires that each of the tasks described earlier must be implemented such that they are essentially transparent to the programmer. The multicomputer Maisie implementation was used to develop and execute a model of the shark's world problem on the the Symult and the experimental results were described in [BL91].

This paper examines the effect of using alternative decomposition, state-saving, and state correction strategies on the completion time of the queueing network simulation models described in the previous section. As such, the four major tasks described earlier were programmed explicitly in the simulation model. The simulation support facilities provided by the Maisie system were not used; rather Maisie was used simply as a convenient framework within which to express the parallel computations. In the next section, we describe how each of the FFN, CQNF and CQNP networks were modeled for parallel execution and compare the completion time of the parallel implementations with their sequential counterparts.

5 Results

Speedup is the primary metric of interest in these experiments. For a given configuration, speedup is defined as $S = T_{seq}/T_{par}$, where T_{seq} is the completion time for the simulation when executed using a sequential simulation algorithm and T_{par} is the completion time using a parallel implementation. Except where noted, each sequential run was executed on a single node of the Symult S2010 multicomputer whereas the parallel implementations used multiple nodes that are identical to the one used for the sequential executions. Other than the simulation algorithm, the model for the sequential and parallel implementations is an essentially identical Maisie program.

5.1 CQNF and CQNP Experiments

For the CQNF network, each merge process was modeled by a Maisie entity called *switch* and each tandem queue together with its associated fork process was modeled by another entity called *queue*. Thus each node of the physical network is modeled by one *switch* and one *queue* entity; for parallel implementations of the model both these entities are executed on a single processor. The *switch* entity is straightforward and is not discussed further. The *queue* entity works as follows: for each arriving job, the entity sequentially simulates service of the job at its Q servers and generates a message for one of its O neighbors. When k jobs have been serviced (for a suitably chosen k), the messages generated by the entity are sent to the corresponding neighbors and the final state of the entity is saved. If a subsequent message at a *queue* entity contains a job with an arrival-time that is smaller than the departure time of any of the jobs serviced in the previous iteration, the entity is recomputed from an appropriate state. The parameter k has a significant impact

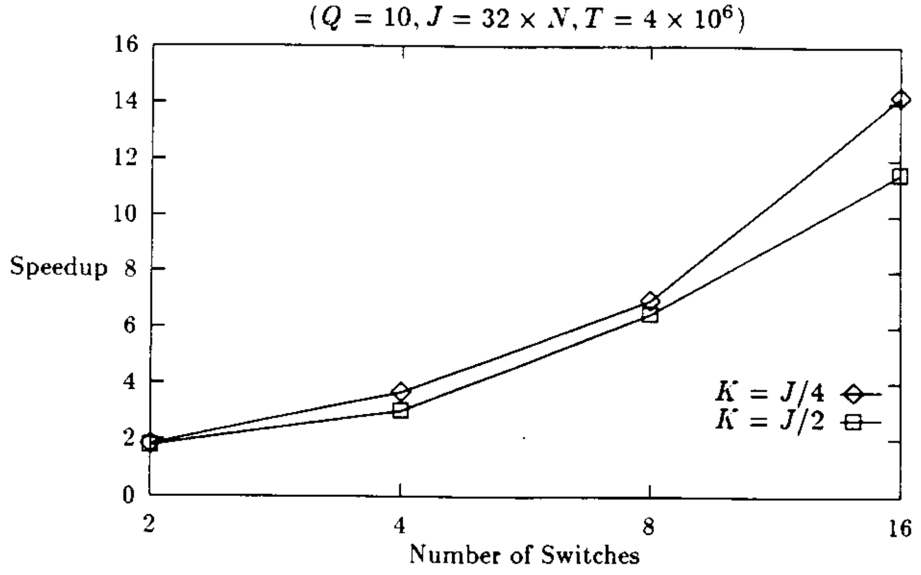

Figure 4: CQNF Speedup: number of switches

on the performance of the implementation. Other things remaining the same, the value of k affects the frequency of state saving, the amount of network message traffic, and the frequency with which the system convergence time is computed which also affects the frequency of garbage collection. In addition to the application parameters, the performance of this implementation was also studied as a function of k.

The convergence time for this model is computed by using an asynchronous convergence detection algorithm: each outgoing message from an entity carries its local convergence time and a message number that serializes the number of *correct* messages sent to each neighbor. Every entity periodically checks to determine the largest number corresponding to which messages have been received from all *queue* entities in the model and uses the value to compute the system convergence time.

The first graph (figure 4) measures the speedup obtained in the parallel implementations as a function of N. The other primary parameters were fixed at Q=10 and J=32*N. As seen from the figure, the speedup increases almost linearly with N for $N \leq 16$. Larger values of N are not included in this graph, as the 4M limitation on the main memory of each multicomputer node restricted the largest sequential implementations that could be executed. Modified configurations that were executed for larger values of N are discussed subsequently.

Given a network of N switches, the amount of computation is determined by the

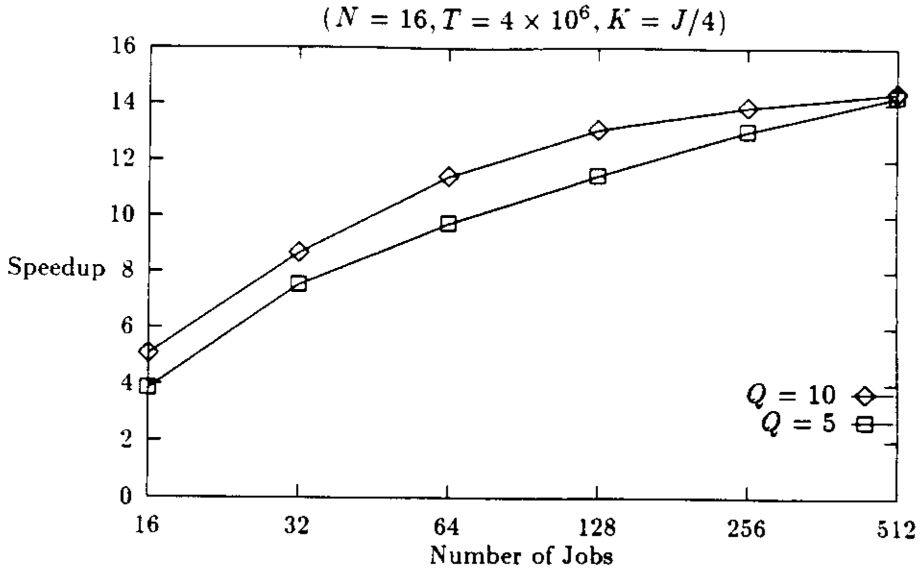

Figure 5: CQNF Speedup: jobs per switch

number of jobs (J) assigned to each switch as also by the number of servers in each tandem queue(Q). Figure 5 presents the speedups obtained for a network of N=16 fully connected switches as a function of the number of jobs in the system, for Q=5 and 10 respectively. Note that initially the speedup increases with J and then levels off to reach a peak speedup of about 14 for J=32*N. This behavior is expected as the initial increase in J offsets the message communication time in the network and once every node is fully utilized, further increases in J have no affect on the speedup. Note also that whereas for smaller values of J the speedup is greater for larger Q, the value of Q is less relevant as J is increased. This is intuitively plausible, as for smaller J, the larger Q implies that each node is being better utilized. As J increases, each node is fully utilized even for the configuration with a smaller Q. The experiments that are reported in this graph assumed an iteration count (k) of J/4. The behavior of the network as a function of k is discussed subsequently.

To permit networks with larger N to be executed, memory requirements of the model were reduced by changing the topology. Rather than use fully connected networks, each switch was assumed to be connected to N/2 other nodes in the system. Furthermore, the number of jobs were reduced from J=32*N (which yielded the optimum speedups for the experiments in figure 5) to only 4*N. This allowed us to execute simulations of upto 64 switches, and although the speedup is sub-optimal, it was still close to 35 for N=64. Figure 6 gives the speedup as a function of N, *where*

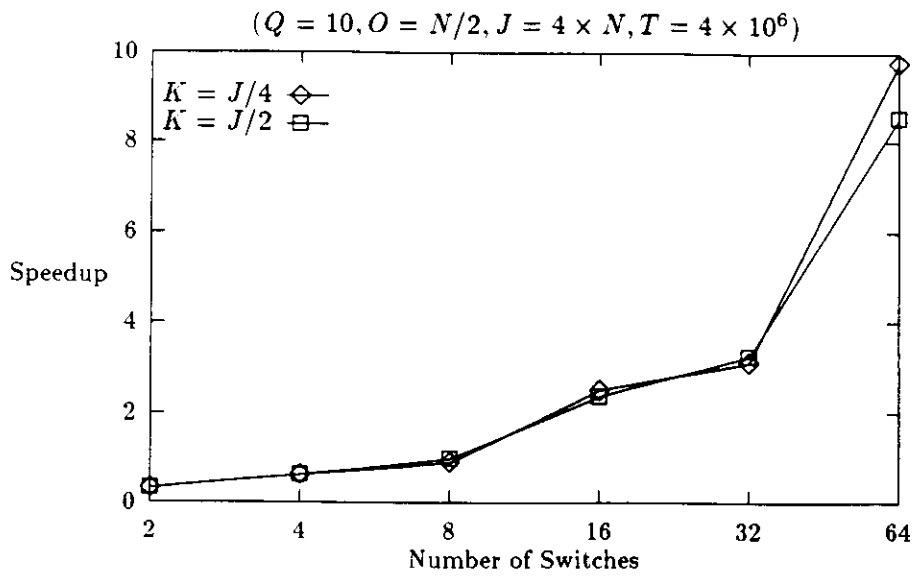

Figure 6: CQNF: Speedup over Sun Sparc-IPC

the sequential implementations were executed on a Sparcstation IPC, a configuration that is considerably more powerful than the single node of a Hypercube. As seen from the figure, significant speedups were obtained (upto a factor of 10) even when compared against a superior sequential implementation.

The final set of experiments were designed to measure the overhead due to the simulation algorithm itself. As mentioned earlier, the iteration count k has a significant impact on the frequency of checkpointing, convergence detection and other tasks performed by the run-time system. Figures 7, 8 and 9 present the speedup, overhead and number of rollback events as a function of k for two configurations that differ only in the number of jobs in the network. The overhead costs consist primarily of checkpointing, garbage collection, determining recomputation points in the queue of checkpointed states and the message delivery costs (both hardware and software). Note that rollback costs are not included as a separate cost as they indirectly contribute to the preceding categories. For J=32*N, the speedup increases initially and then decreases. The intuitive explanation for this behavior is that as k decreases, it initially causes the simulation to synchronize more frequently, thus reducing the number of rollbacks. Also, the normal increase in overhead caused by the lower k is more than offset by the decrease due to the reduced number of rollbacks, resulting in overall improvement in the speedup. However, as k is decreased further, rollbacks are not reduced significantly whereas the other overhead

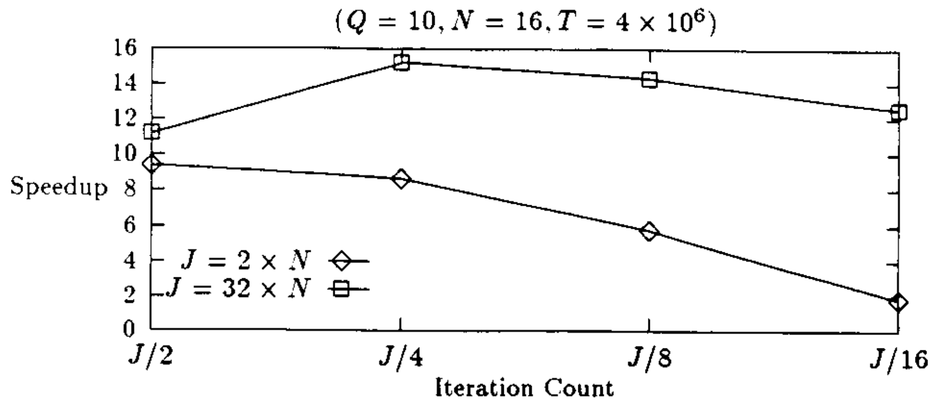

Figure 7: CQNF Speedup: iteration count

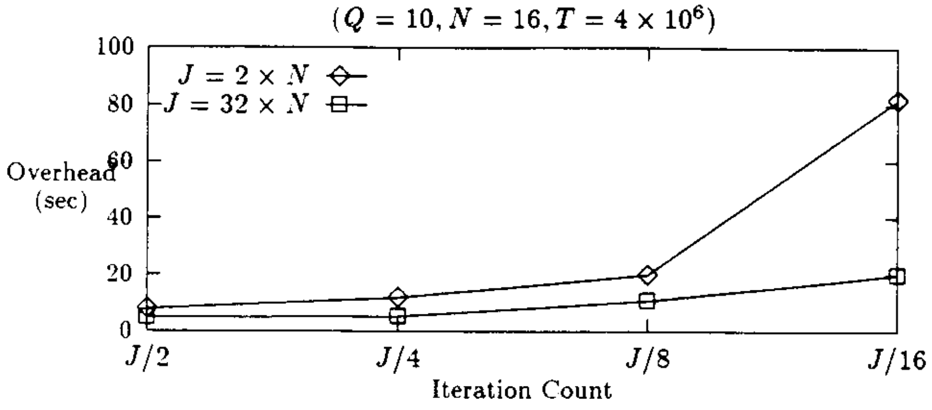

Figure 8: CQNF Time: system overhead

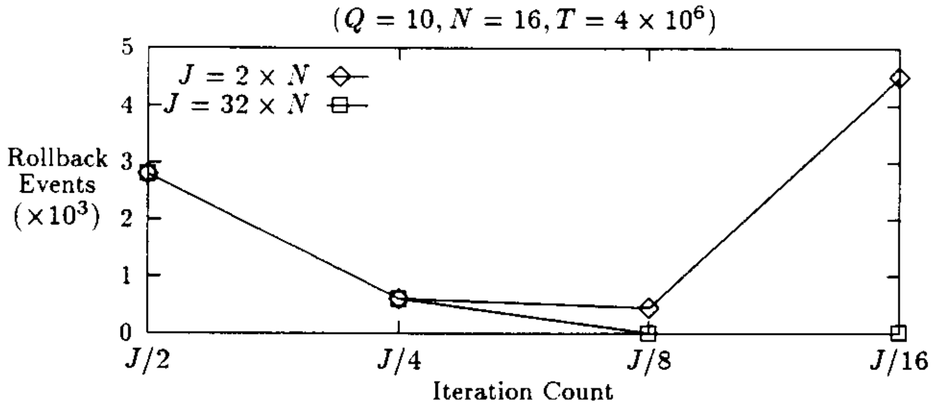

Figure 9: CQNF: rollback events

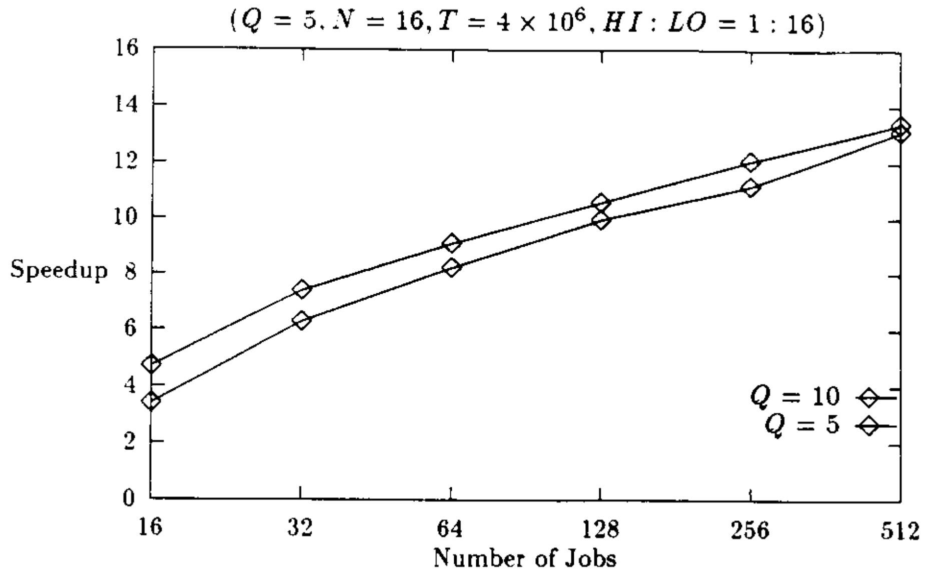

Figure 10: CQNP Speedup: jobs per switch

increases causing a net decrease in speedup. This explanation is in fact supported by the graphs in figures 8 and 9 respectively. For J=2*N, the speedup decreases monotonically, dropping significantly for the lower values of k. This is accompanied by a sharp increase in the overhead costs and also in the number of rollbacks. In examining the contributants to the overhead, it was found that although all costs increase, the major increase is due to higher message delivery costs. Note that for this configuration, the lowest value of k implied that each queue processed exactly 2 jobs before synchronizing with its neighbors. This caused a significant increase in the total number of iterations needed for the simulation, and consequently increased the simulation overheads. As the amount of useful work done per iteration is relatively small, this configuration yielded a minimal speedup of 2.

The implementation of the CQNP benchmark was very similar to the CQNF implementation, where a *queue* was used to model the Q servers and the fork process and a *switch* entity modeled the merge process. The speedup curves for the CQNP application were similar to those obtained for the CQNF experiments. For a low ratio of high to low priority jobs, the speedup curve as a function of the number of jobs (J) is displayed in figure 10 where the speedup is seen to increase as J is increased, and is independent of the number of servers (Q) for larger values of J. Figure 11 shows that the speedup also improves almost linearly as the number of switches is increased; note that the number of nodes used for the parallel implementations is

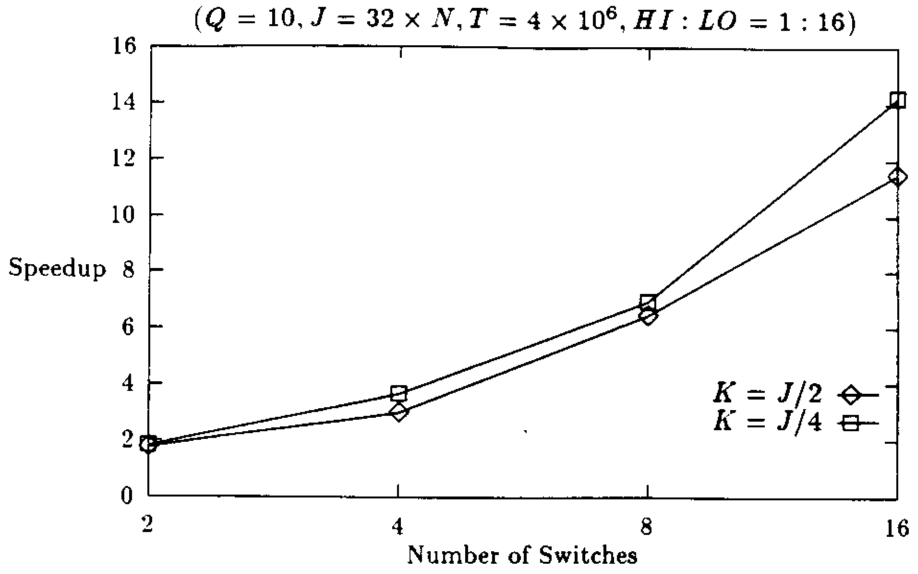

Figure 11: CQNP Speedup: number of switches

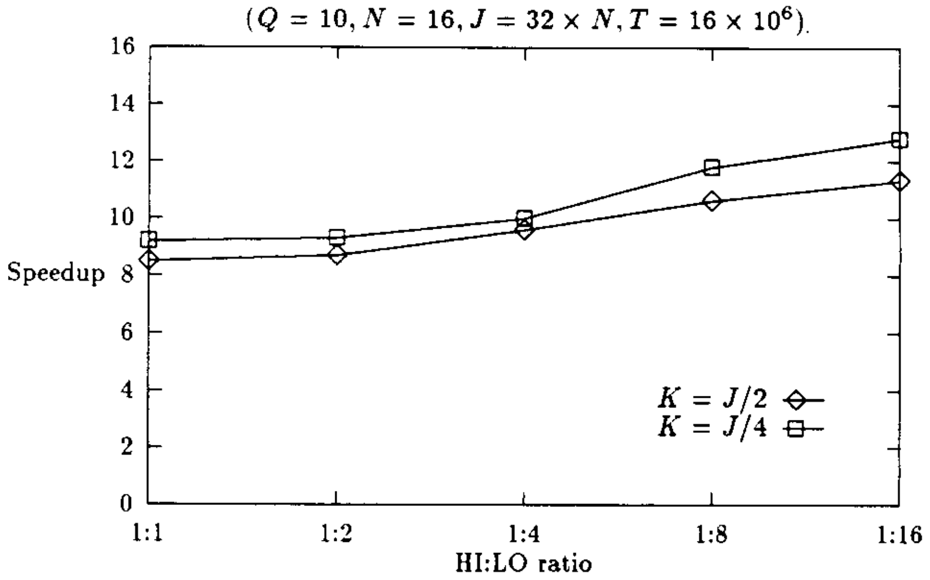

Figure 12: CQNP Speedup: High/Low job ratio

such that each switch and its associated queue is mapped to a single node of the multicomputer. Finally figure 12 displays the speedup as a function of the ratio of high to low priority jobs. Other things remaining the same, the speedup increases as the ratio is decreased. This is expected because as the ratio decreases there is less likelihood of a low priority job being preempted and as such a lower probability that the completion message for a low priority job is subsequently canceled by a rollback.

5.2 FFN Experiments

The FFN benchmark was used to measure the effectiveness of space parallelism, time parallelism and also a combination of the two methods. For the space parallel implementation, each stage of the network was modeled by a single entity. This ensures that every entity processes an equal number of messages and also simplifies the allocation of processes to processors. The frequency of checkpointing and convergence detection is once again determined by a suitably chosen k, which represents the number of jobs that are processed at a stage before they are forwarded as a packet to the next stage. The convergence detection is simpler for this application as the convergence time at each stage can be determined locally from messages received from the preceding stage.

For the time-parallel implementation, the simulation horizon H was subdivided into smaller intervals, each of duration T. The simulation model essentially consists of a number of *ffn* entities, each of which is responsible for simulating the entire network for a specific interval of T time units. Let entity ffn_i simulate the network in the interval t_i=[(i − 1)T, iT), i= 1,2 ...n, for n=H/T. The parallelism exploited by this implementation is the simultaneous computation of the state of the network at many distinct points in time. Each *ffn* entity is initialized to an identical state, where its servers are assumed to be idle. Subsequently, each entity simulates job arrivals and their service at its servers during interval t_i. The state of an *ffn* entity is represented by the departure time of each job from each of its servers. At the end of its simulation interval (at time i*T), entity ffn_i transmits the state of each server (either as idle, or if busy, the remaining service time) to entity ffn_{i+1}. Note that entity ffn_{i+1} has simultaneously simulated the network in the simulation interval t_{i+1} under the assumption that every server is idle at time i*T. Under the normal optimistic assumptions, if the final state transmitted by entity ffn_i is such that all its servers are not idle, entity ffn_{i+1} must be recomputed. As it is unlikely that the interval T can be predetermined such that the entire network is periodically empty, this recomputation will effectively sequentialize the parallel implementation. However, if the final state transmitted by entity ffn_i can be used to correct the state of entity ffn_{i+1} at time (i+1)*T without recomputation, significant speedups may be realized. In the parallel implementations reported in this paper, the state of ffn_{i+1} was corrected by adjusting the departure time of its jobs to account for

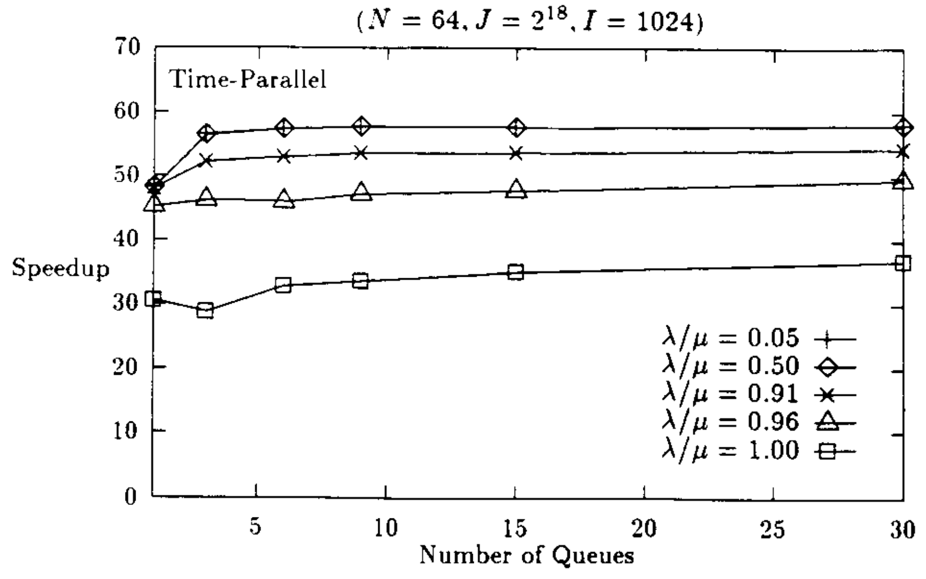

Figure 13: FFN Speedup: number of servers

the remaining service time of jobs that are still in service (or in the queue) at time i*T. If the final state of entity ffn_{i+1} after these adjustments is different from what was computed in the previous iteration, then further change in the state of entities ffn_{i+2} (and indirectly its successors) may be needed. Thus the computation proceeds in phases, where in the first phase all entities simultaneously simulate the network for duration t_i but only entity ffn_1 is guaranteed to have converged. In the worst case, the simulation may need to be executed for n phases before it converges. (Note that each phase need not be executed synchronously). However, for networks with low utilization, the implementation converges much faster.

The first graph (figure 13) for this set of experiments studied speedup as a

$\lambda : \mu$	States Fixed	Time(sec)
1.00	26,785,820	81.48
0.91	1,478,436	55.02
0.50	18,563	51.54
0.05	106	51.48

Figure 14: FFN: Number of states fixed ($S = 1, Q = 30, J = 2^{18}$)

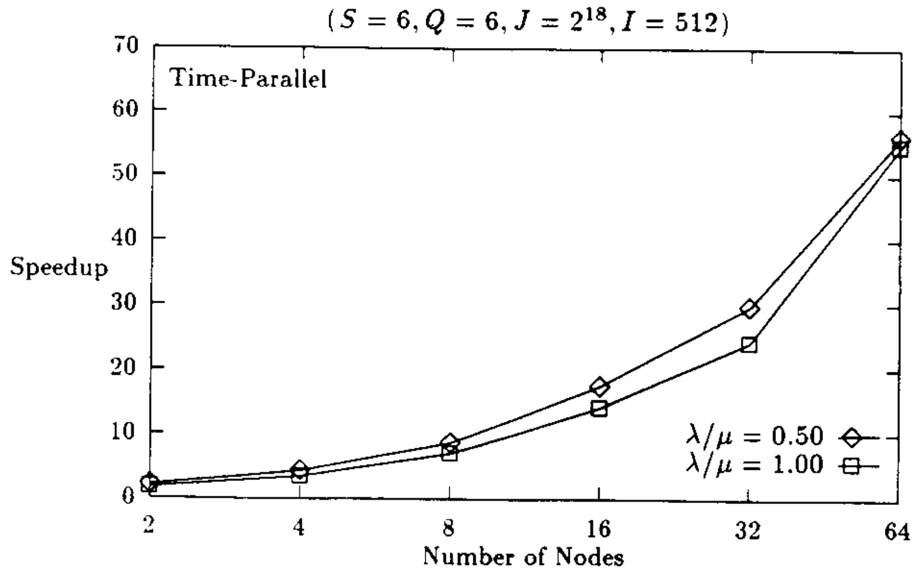

Figure 15: FFN Speedup: number of processors

function of network utilization for a simple network that consists of 1 stage. The simulation interval (T) assigned to each entity was chosen such that an entity is responsible for generating I=512 arrivals. The graph in figure 13 describes the speedup as a function of the number of servers(Q) for different $\lambda : \mu$ (R) ratios. Other things remaining the same, the speedup is better for a network with a lower R ratio. This is expected as a lower ratio implies that each server has many idle periods, which implies that when the state of some ffn_i is corrected, it is less likely to cause changes in its successor. The table in figure 14 corroborates this conjecture. It shows that whereas the configuration for R=0.05 needed only 18K states to be fixed-up, for R=1 almost 26M states needed to be corrected. This significant difference in the state correction activity is primarily responsible for the large variation seen for the speedups in the two configurations. For a ratio of 0.05, the speedup on a 64 node network was almost 58, which is close to optimal. It is relevant to mention that unlike the CQNF and CQNP experiments, the sequential implementation for the FFN experiment was a simple C function whereas the parallel implementations were written in Maisie. In our opinion, it would be hard to further improve the efficiency of the sequential model.

The second graph (figure 15) describes speedup as a function of the number of multicomputer nodes. As seen from the graph in figure 15, the speedup is almost linear with a value of 56 for 64 nodes. The tested configuration was a 6 stage

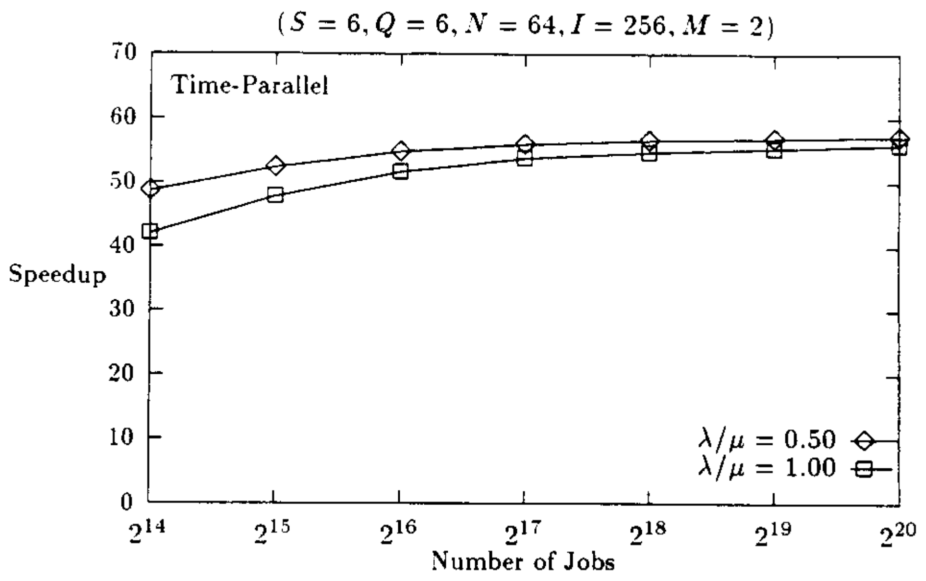

Figure 16: FFN Speedup: total number of jobs

network with a branching factor of 2 and a simulation interval (T) that generates I=512 arrivals.

The third graph (figure 16) studied speedup as a function of the total number of job arrivals. Given that each entity simulates the network for 256 job arrivals, a 64 node multicomputer would be fully utilized for a simulation horizon of 2^{14} job arrivals and further increasing the number of jobs will not improve speedup. However, as shown in the graph of figure 16 there is an initial increase in speedup of about 10% which finally levels off only after 2^{18} job arrivals have been simulated. If each node simulates the network for only one interval, then the node must remain idle once the simulation has converged over its time interval, even though it may not have converged over the rest of the network. However, if multiple (non-consecutive) time-intervals are allocated to each node, it may continue to do useful work for a longer time, improving the utilization and hence the speedup. The value of I also has some effect on the speedup, as seen from the graph in figure 17 where the speedup was within 10% of its maximum value as I was varied from a low value of 64 to a high of 8K with the speedup being at its maximum value for 256 or 512 depending on the $\lambda : \mu$ ratio for the network.

Finally, an implementation of the FFN network was also used to examine the consequence of combining space and time parallelism: for these experiments, each stage of the network was assigned τ nodes, where $\tau = N/S$, and N is assumed to be

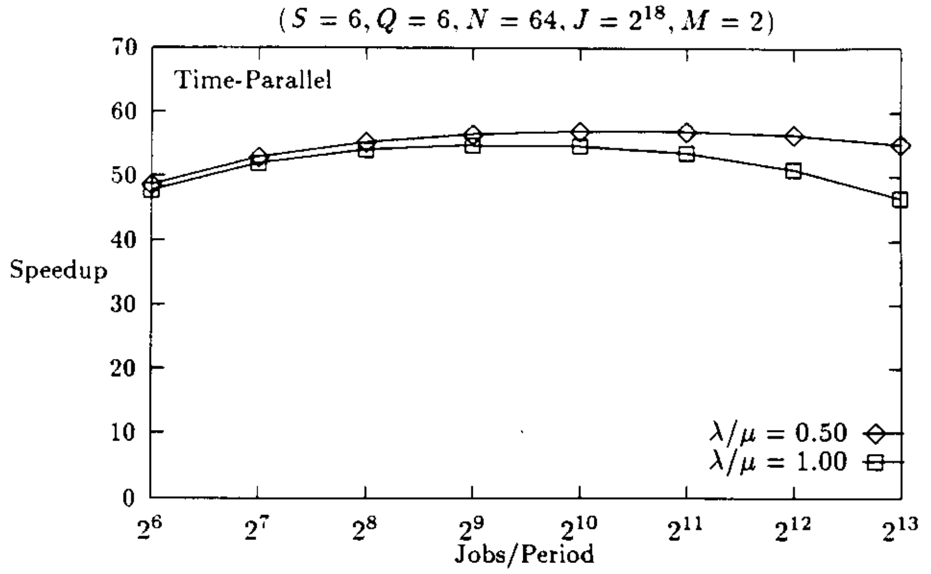

Figure 17: FFN Speedup: iteration count

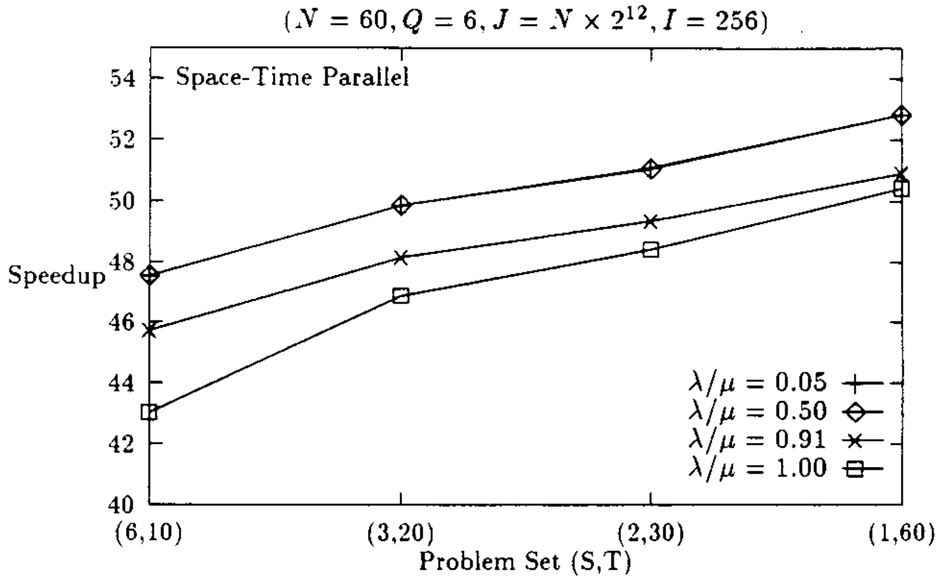

Figure 18: FFN: Space-Time Speedup

a multiple of S. A number of experiments were performed to measure the relative effectiveness of the time and space parallelism. Consider the following scenario: assume that N=S. A pure space parallelism would imply that each stage be assigned to a unique node. A pure time parallelism could cause the entire network to be simultaneously simulated for N different time intervals. A hybrid implementation would map multiple stages to a single node and use the additional nodes to exploit time parallelism. If N>S, the space parallelism can exploit at most S nodes, where the remainder can be used to exploit time parallelism. The performance of a FFN network with S=6, M=2 and Q=6 as a function of these mappings is presented in figure 18. Given a network of 6 stages and 60 nodes in the multicomputer, four different mappings were attempted: in the first case, each stage is mapped to a different node and is simultaneously simulated for 10 consecutive time periods, thus allocating a total of 10 nodes for each stage. In the subsequent mappings, the extent of space parallelism is decreased by mapping 2, 3 and eventually all 6 stages on a single node and using the extra nodes to increase time parallelism. As seen from the graph in figure 18, the overall speedup is best when the entire network is executed using only time parallelism.

6 Conclusion

This paper examined the effectiveness of different implementations of the space-time simulation algorithm in improving the completion time of of a class of stochastic simulation models. Almost all existing performance studies for parallel simulations have only examined the effectiveness of exploiting spatial parallelism for the execution of a model. In this paper, we described a set of experiments that used space parallelism, time parallelism as well as a combination of the two strategies to reduce the completion time for simulation models. In addition, rather than use rollback and recomputation to correct estimated future states, the time-parallel implementation uses a simple fix-up strategy that effectively eliminates rollbacks and provides near optimal speedups for a class of feed forward networks. The viability of state correction mechanisms in the implementation of other types of systems is currently being studied.

References

[BL90] R.L. Bagrodia and Wen-toh Liao. Maisie: A language and optimizing environment for distributed simulation. In *1990 Simulation Multiconference: Distributed Simulation*, San Diego, California, January 1990.

[BL91] R.L. Bagrodia and W.T. Liao. Parallel simuation of the sharks world problem. In *Western Simulation Conference*, 1991.

[CS89a] K.M. Chandy and R. Sherman. The conditional event approach to distributed simulation. In *Distributed Simulation Conference*, Miami, 1989.

[CS89b] K.M. Chandy and R. Sherman. Space-time and simulation. In *Distributed Simulation Conference*, Miami, 1989.

[Fuj88a] R. Fujimoto. Lookahead in parallel discrete event simulation. In *International Conference on Parallel Processing*, August 1988.

[Fuj88b] R. Fujimoto. Time warp on a shared memory multiprocessor. Technical report no. uucs-88-021a, Computer Science Dept., University of Utah, 1988.

[GL90] I. Greenberg, A.G. Mitrani and B. Lubachevsky. Unbounded parallel simulations via recurrence relations. In *1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 1–12, 1990.

[JBWea87] D. Jefferson, B. Beckman, and F. Wieland et al. Distributed simulation and the time warp operating system. In *Symposium on Operating Systems Principles*, Austin, Texas, October 1987.

[LL91] Y.B. Lin and E.D. Lazowska. A time-division algorithm for parallel simulation. *ACM Transaction on Modeling and Computer Simulations*, 1:73–83, January 1991.

[RMM87] D.A. Reed, A.D. Malony, and B.D. McCredie. Parallel discrete event simulation: A shared memory approach. In *Proceedings of the 1987 ACM SIGMETRICS Conference*, pages 36–39, May 1987.

[SS89] Wen-king Su and C.L. Seitz. Variants of the chandy-misra-bryant distributed simulation algorithm. In *1989 Simulation Multiconference: Distributed Simulation*, Miami, Florida, March 1989.

[SS90] L.M. Sokol and B.K. Stucky. MTW: experimental results for a constrained optimistic scheduling paradigm. In *Proceedings of the SCS Multiconference on Distributed Simulation*, pages 169–173, January 1990.

[WL90] D.B. Wagner and E.D. Lazoska. Parallel simulation of queueing networks: Limitations and potentials. In *Proceedings of 1989 ACM SIGMETRICS and PERFORMANCE*, pages 146–155, May 1990.