CHAPTER III

Parallel Simulation: Spatial and Temporal Methods

3.1 Introduction

In the previous chapter, an efficient serial algorithm was developed to compute the dynamics of multiple manipulators that are cooperating to manipulate large loads. This algorithm, which determines accelerations, must be combined with a numerical integration routine in order to simulate the system. Because numerical integration is a discrete approximation of a continuous process, the dynamics computation must be performed hundreds or even thousands of times each second to achieve accurate results. On all but the fastest supercomputers, this computation, today, cannot be performed in real-time. One approach that has been taken to improve computational rates is to parallelize the dynamics computations. With significant speedups over serial implementation, real-time performance can be achieved and hardware-in-the-loop and human-in-the-loop applications can be implemented.

To achieve these rates, fine-grain parallel algorithms have been developed in most previous work. Some have required the use of special-purpose architectures to implement the fine-grain parallelism of the computations required for a single-chain system [18, 19, 20, 21, 38, 39]. Others decompose the computation into slightly larger blocks of concurrent tasks and schedule them on a number of tightly coupled parallel processors to produce speedup [22, 23]. These approaches, while designed to perform well on specific parallel architectures, are much less efficient when implemented on the more widely available general-purpose parallel and desktop multiprocessor systems. These systems typically have a limited number of loosely coupled processors that are not designed to efficiently handle the large amount of interprocessor communication and synchronization required by the fine-grain tasks.

Results from this chapter are to be published in "Parallel Dynamic Simulation of Multiple Manipulator Systems: Temporal vs. Spatial Methods," To appear in *IEEE Trans. on Systems, Man, and Cybernetics*, vol. 24, no. 8, August 1994 (with P. Sadayappan and D. E. Orin).

In contrast, the work in this chapter focuses on the development of parallel algorithms which run efficiently on existing general-purpose parallel computers, such as hypercubes, or multiprocessor workstations that are now available from SGI, Sun, DEC, and HP. This approach has distinct advantages over the fine-grained approaches because less time is required to develop the algorithms and no hardware development must be performed. In addition, the general-purpose architectures allow for greater flexibility in the approaches to parallelism to be investigated.

In the first stage of this research, the dynamic simulation algorithm for a single, open-chain robotic manipulator was effectively parallelized and the results were reported in [36]. The approach used a parallel block predictor-corrector (BPC) integration method to achieve *temporal* parallelism in which the dynamics of the system was computed for multiple points in time simultaneously. The purpose of this chapter is to extend this work to the simulation of multiple closed-chain systems and, in particular, multiple manipulator systems that are cooperating to manipulate a large load called the reference member. In this case, a second form of parallelism can be exploited which allows the dynamics of the individual chains to be computed in parallel. This form of parallelism corresponds to the system's structural parallelism and is called *spatial* parallelism. Both forms of parallelism can be combined by implementing this spatial parallelism *within* each temporal parallel task. In this case, the number of parallel tasks possible in the dynamic simulation is the product of the two.

In the next section, the computational structure of the dynamics algorithm for the multiple, closed-chain algorithm from Chapter 2 is briefly reviewed, and its spatial parallelism is discussed. In Section 3.3, numerical integration which is required to perform simulations is introduced. An improved form of the BPC integration method from [36], called the Sliding BPC (SBPC) method, is developed and its increased flexibility will allow varying amounts of temporal parallelism in the simulation algorithm while maximizing accuracy. Section 3.4 discusses how both forms of parallelism are combined and implemented on a general-purpose parallel computer and provides an analysis of the interprocessor communication and serial processing overhead required in the resulting algorithm. This algorithm is then implemented on a CRAY Y-MP8/864, and the speedup results are given in Section 3.5 for various parallel configurations of the simulation algorithm including temporal parallelism only, spatial parallelism only, and various combinations thereof.

3.2 Spatial Parallelism

The dynamics algorithm to compute the accelerations, the derivatives of state variables, in Table 1 is outlined as a flowchart in Figure 5. Emphasis has been placed on

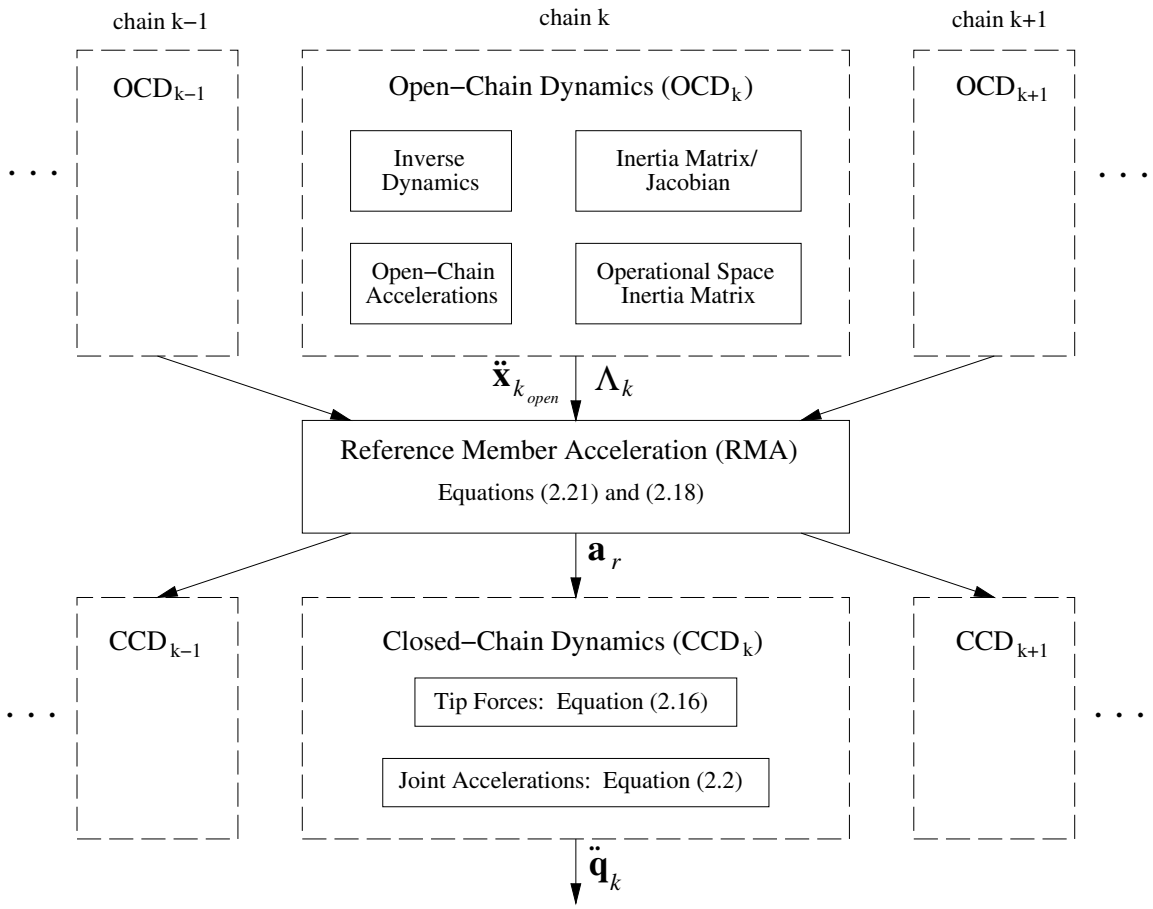

Figure 5: Multiple manipulator dynamics algorithm.

the fact that the computations within the Open-Chain Dynamics (OCD) and Closed-Chain Dynamics (CCD) steps for each chain in the system does not depend on the computations for any other chain in the system. Therefore, these computations can be performed in simultaneously which leads directly to the *spatial* parallelism in the computation. In the middle of this spatial parallelism, the Reference Member Acceleration (RMA) step is performed once for a given derivative evaluation, and represents serial computation that must be performed.

Figure 6 shows how this algorithm could be implemented on a general-purpose parallel computer for an m-chain system. Processor synchronization is required after the parallel OCD computation in order to collect the chains' operational space inertia matrices and open-chain accelerations before the serial RMA computation can be performed. In Figure 6 this is indicated by the *barrier* whose purpose is to

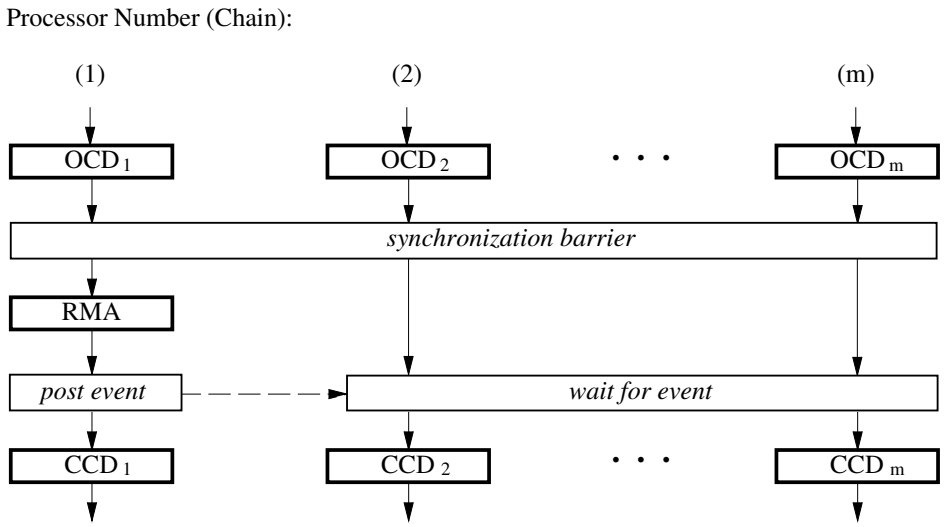

Figure 6: Spatial parallelism in multiple-chain dynamic equations (serial RMA computation).

hold the parallel tasks which have completed the OCD section until all of them have finished. After the serial computation is completed by one processor, it signals the other processors, via an *event*, to continue with the parallel CCD computations.

This computation is modified in Figure 7 to remove the event synchronization by allowing each task to maintain its own "copy" of the reference member information and perform its own RMA computation. While this introduces redundant computation, the execution (wallclock) time will not increase since it is performed during the time that the processors were waiting on one to complete the RMA computation. It can even reduce the execution time because a costly interprocessor synchronization point has been removed.

3.3 Numerical Integration and Temporal Parallelism

Up to this point, the dynamics algorithm, which computes the accelerations in a robotic system given the state of the system and the joint inputs, has been examined. This is only part of a simulation algorithm which couples a numerical integration routine with this dynamics computation to update the state information for the next point in time. In this context, the dynamics constitutes the "derivative computation" which uses the state information, positions and velocities, provided by a numerical integration routine and the input torques and forces. In turn, the

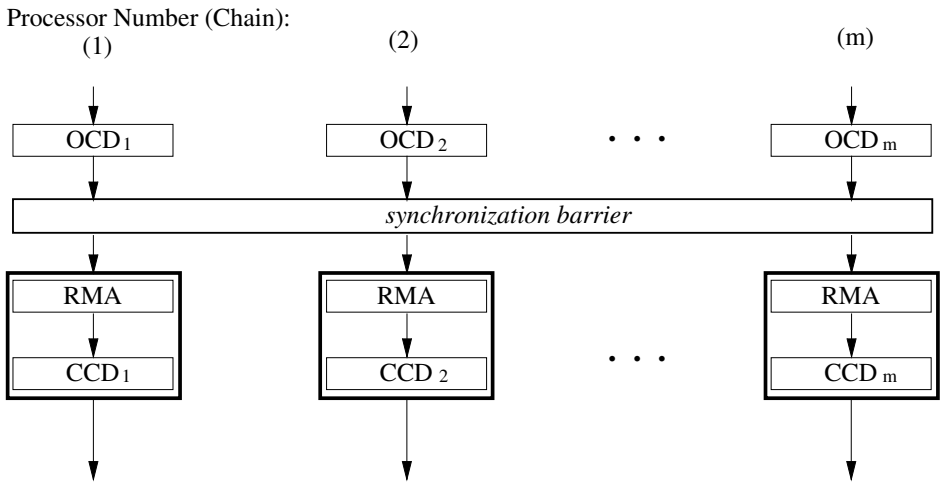

Figure 7: Spatial parallelism in multiple-chain dynamic equations (redundant parallel RMA computation).

dynamics provides the numerical integration routine with a new state derivative, velocities and accelerations, which is integrated to obtain the updated state.

Many different methods are available to perform this numerical integration, and the most popular are predictor-corrector and Runge-Kutta methods. In this section, temporal parallelism is investigated which is more readily achieved with the class of predictor-corrector methods called PECE. To describe this form of parallelism, a common serial version of the PECE method is reviewed. Then, a new approach for predicting and correcting the states in parallel, called the Sliding Block Method, is presented.

3.3.1 Serial PECE Method

The PECE methods can be divided into two steps regardless of the order of the method (unlike Runge-Kutta methods). In the first step, the state at the next desired point in time is *P*redicted based on past values and the derivative is *E*valuated based on this prediction. Then, this state is *C*orrected followed by another derivative *E*valuation. In this chapter, fifth-order methods are used to provide an adequate tradeoff between accuracy, which tends to increase with order, and numerical stability, which tends to decrease with order. Figure 8 shows the quantities needed and computed in both steps of the serial method.

The predictor step is shown in Figure 8(a). It computes a linear combination of five (for fifth-order) past derivative vectors, $\mathbf{f}_{-4}, \ldots, \mathbf{f}_0$, and the state at the most

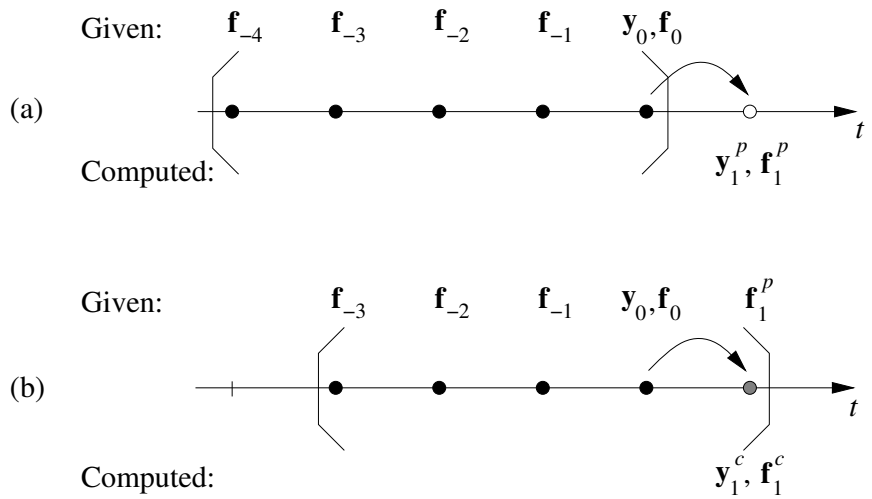

Figure 8: Serial B1PC5 integration: (a) predictor step, (b) corrector step.

recent point in time, $\mathbf{y}_0$ to predict the state of the system at the next point in time, $\mathbf{y}_1^p$. This is used by the dynamics algorithm to compute the derivative vector (joint and reference member accelerations), $\mathbf{f}_1^p$, at this point in time. Note that in this discussion regarding numerical integration, the derivative of state vector, $\mathbf{f}$, should not be confused with a spatial force vector which is used in the discussions of the dynamics algorithms.

To correct the state in the second step of the method, the quantities shown in Figure 8(b) are used. The "oldest" derivative vector is dropped and the linear combination used to compute the corrected state, $\mathbf{y}_1^c$, now includes the predicted derivative vector. The same state vector at time step zero, $\mathbf{y}_0$, is used rather than the predicted one at time step one, $\mathbf{y}_1^p$, for reasons of stability and accuracy. However, the corrected derivative vector is then computed using $\mathbf{y}_1^c$.

In the next iteration of the method, the values are shifted and four old derivative vectors, $\mathbf{f}_{-3}, \ldots, \mathbf{f}_0$ and the new corrected derivative, $\mathbf{f}_1^c$, become the five derivative vectors used in the next predictor step along with the newest corrected state vector, $\mathbf{y}_1^c$. The block of points to be used has "slid" forward by one step in time while maintaining the same number of points required to achieve the desired order of the method. This method is usually called PECE5, but will be referred to as B1PC5 in this chapter to indicate a one-point Sliding-Block Method which utilizes fifth-order predictor-corrector formulas to contrast it from the multi-point block methods presented next.

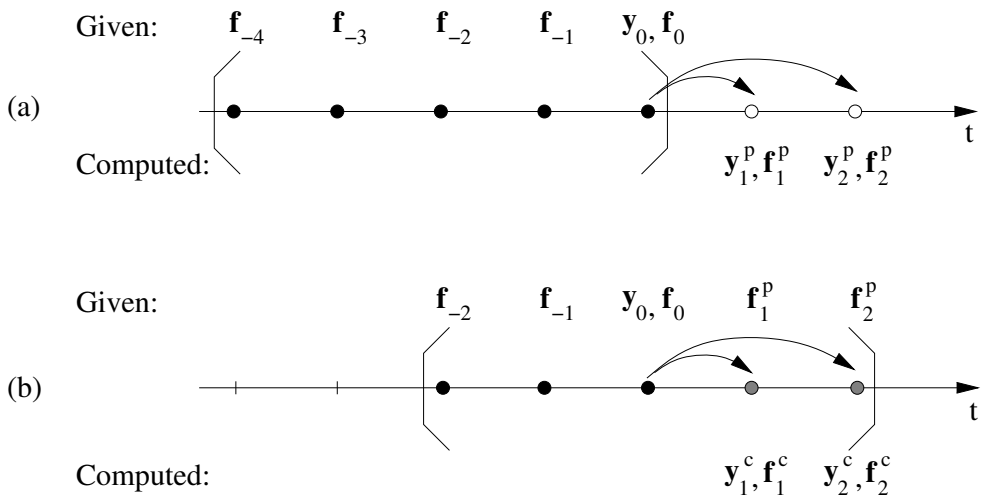

Figure 9: Two-point parallel B2PC5 integration: (a) predictor step, (b) corrector step.

3.3.2 Parallel Sliding-Block Methods

Temporal parallelism can be achieved by performing the computations described above for more than one new point in time. This concept is illustrated in Figure 9 for a two-point method called B2PC5 which can be implemented on two processors in parallel. Each processor has access to the past state and derivative vector information, and with an appropriate linear combination (as determined by the method of undetermined coefficients [40]), computes the predicted state for one of the two points in time. Each processor can then perform the dynamics computations for the system independently to determine the derivative vectors, $\mathbf{f}_1^p$ and $\mathbf{f}_2^p$.

Before the corrector step can take place, however, the processors need to exchange the new derivative information because a linear combination including both of these values is needed to compute the corrected state as shown in Figure 9(b). Note that in this step, *two* old values have now been discarded in favor of the new predicted information to maintain fifth-order accuracy, and it is the corrected information at these five points that will be used in the next iteration of the method.

Any amount of temporal parallelism can be achieved by increasing the number of new points to be computed. For reasons of accuracy and stability which decrease with increasing points, the limit for the number of temporal parallel tasks is chosen as one less than the order of the method. For the fifth-order method considered in this work, this limit corresponds to the four-point B4PC5 method shown in Figure 10. This method is also equivalent to the the parallel block predictor-corrector

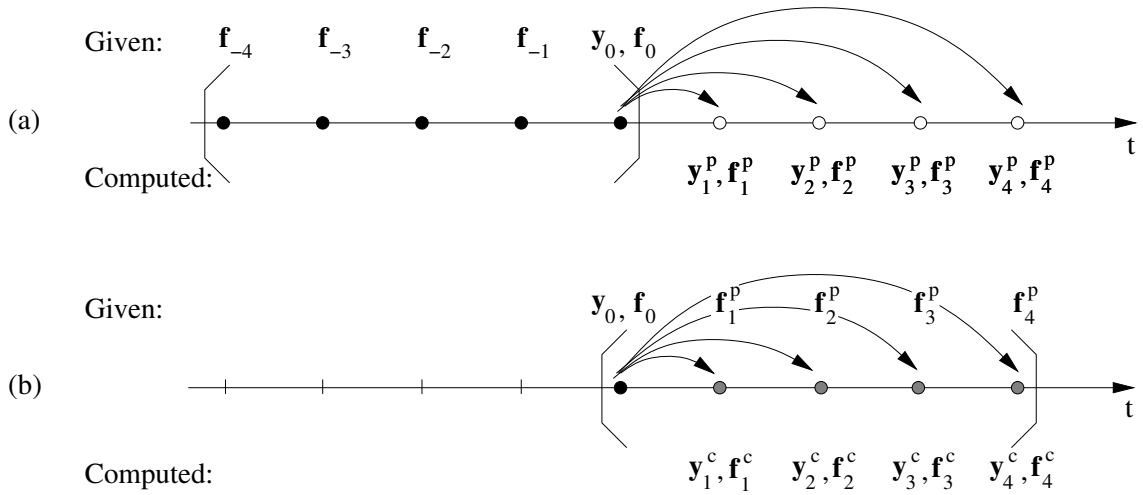

Figure 10: Four-point parallel B4PC5 integration: (a) predictor step, (b) corrector step.

(BPC4) method formulated by Birta and Abou-Rabia [41] and used in [36]. In their approach, however, the number of parallel tasks was always one less than the order of the method to be used so no freedom was given between choosing the desired order of the method *and* the amount of parallelism. The sliding block method presented in this section overcomes this drawback.

The structure of the temporal parallelism for a single iteration of the methods presented in this section is shown in Figure 11. The PE (compute *P*redicted state and *E*valuate the derivative) and CE (compute *C*orrected state and *E*valuate the derivative) blocks correspond to the predictor and corrector steps in these methods along with the derivative evaluations described in the previous section. There are p parallel tasks which correspond to the number of points computed by the method during a single iteration, and a single column of blocks represents the computation of the solution of the entire m-chain system for a specific point in time on one of p processors. Barriers are used to synchronize the processors between the predictor and corrector steps so that the state and derivative information may be exchanged. An event is also used so that a small amount of serial processing may be performed to shift the derivative and state information in preparation for the next iteration.

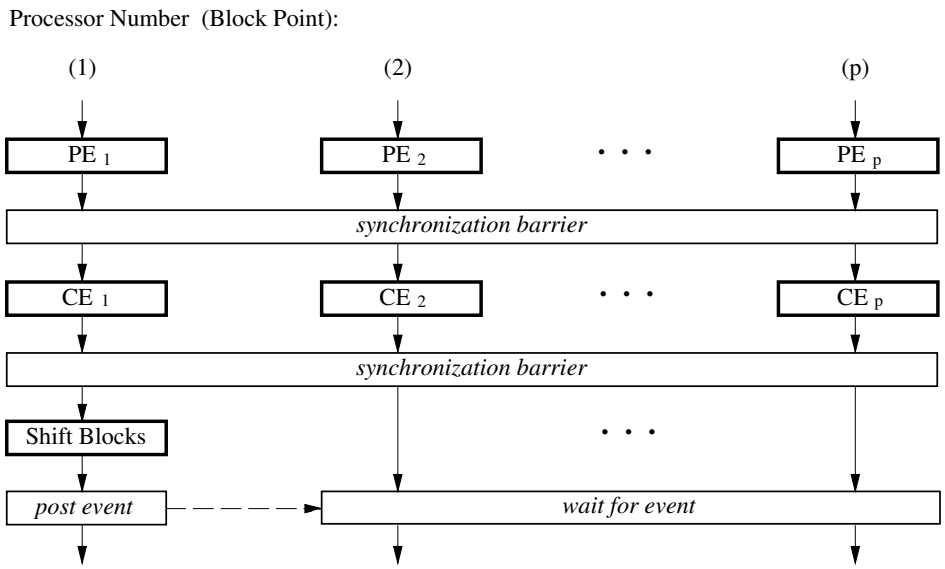

Figure 11: Temporal parallelism in dynamic simulation.

3.4 Combined Parallelism and Task Partitioning

Both forms of parallelism, spatial and temporal, can be combined by implementing the spatial parallelization of the derivative evaluations in Figure 7 within the individual PE/CE computational blocks of Figure 11. Note that the predictor and corrector equations can also be spatially parallelized so that a step to compute the state (either predicted or corrected) of the appropriate chain in the system can be added to each OCD block of Figure 7. A straightforward implementation yields the structure shown in Figure 12.

A drawback to this implementation is the large number of tasks that the event and barriers are required to synchronize after each step of the integration which can lead to a large amount of overhead in some parallel systems. A modification can be made at these points to relax the synchronization requirements and improve performance. Because the integration of the state of each chain depends only on its own values at all of the block points, these barriers (referred to as temporal barriers) and events are broken apart so that each one only synchronizes with the processors assigned to the same chain. As a result, these temporal synchronization points and the Shift Blocks task have been spatially parallelized. Figure 13 shows the resulting structure which consists of $m \times p$ parallel tasks.

This figure also shows the synchronization commands that are used in the implementation of this algorithm on the CRAY Y-MP8/864. The simulation code

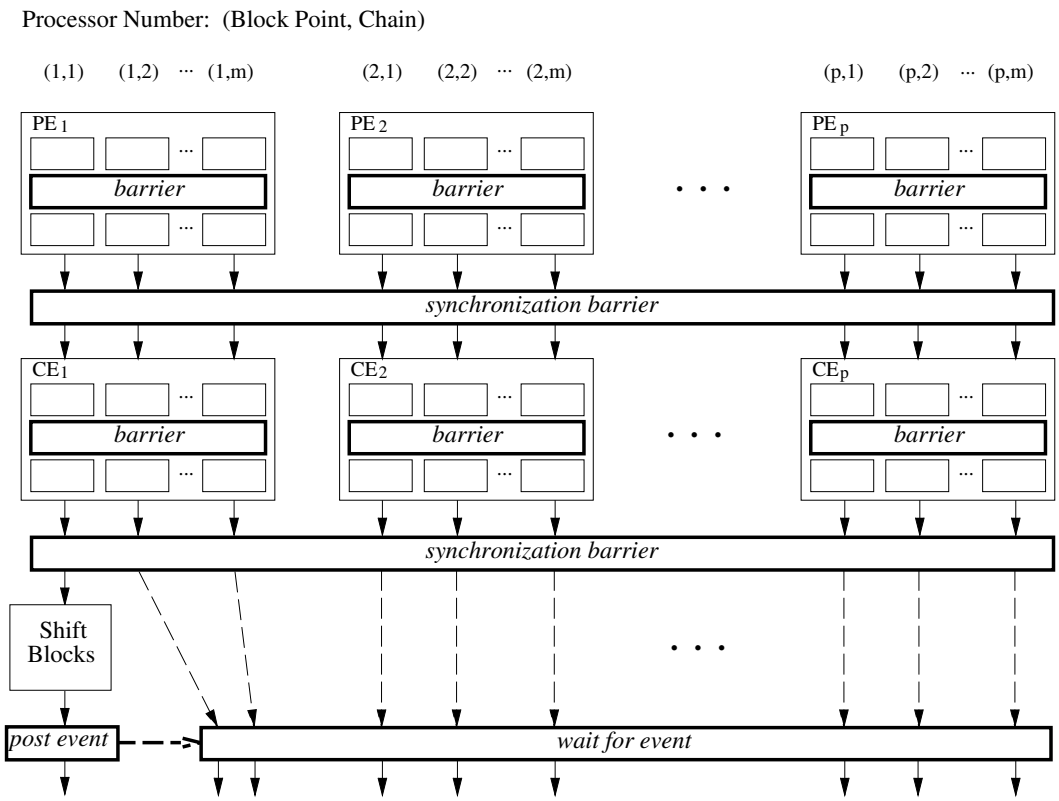

Figure 12: Structure of combined spatial and temporal parallelism.

is written in FORTRAN (Version 5.0 running under UNICOS Release 6.0) using macrotasking commands to implement the parallelism. The BARSYNC commands correspond to the synchronization barriers, and the argument given with this command is the number of parallel tasks that it must synchronize. Finally, the EVPOST-EVWAIT commands provide the mechanism by which some processors are suspended while others perform Shift tasks.

Since the CRAY has only eight processors, a robotic system with four chains and the use of the four-way temporal parallelism would result in more parallel tasks than processors. To overcome this problem a *reduced partitioning* mechanism was included in the code that allows the user to specify the number of desired parallel tasks along both the spatial and temporal dimensions of the simulation independently of the number of chains and block points. In order to maintain load balance, however, the specified partition sizes in both dimensions would have to be integer divisors of the total number of chains and block points, respectively. These numbers are denoted

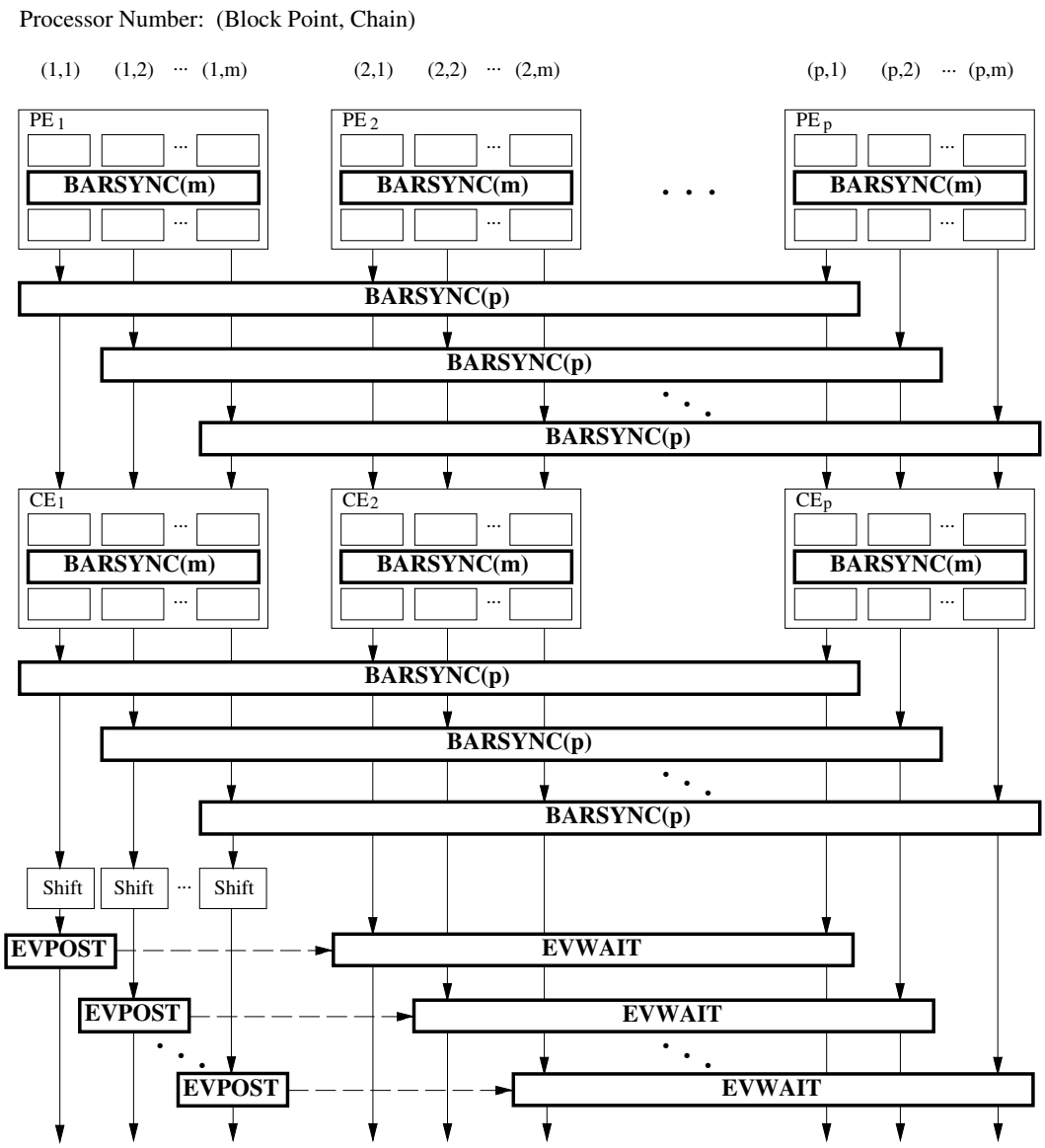

Figure 13: Structure of combined parallelism with reduced synchronization.

Table 3: Number of synchronizations and RMA computations per processor for various partitions.

# CPU	Spatial Tasks, $\tilde{m}$	Temporal Tasks, $\tilde{p}$	Block Size	1 iteration:		4 points:	
				# Sync.	# RMA	# Sync.	# RMA
4	1	4	4	3	2	3	2
	2	2	4	5	4	5	4
			2	5	2	10	4
	4	1	4	2	8	2	8
			2	2	4	4	8
			1	2	2	8	8
8	2	4	4	5	2	5	2
	4	2	4	5	4	5	4
			2	5	2	10	4

by $\tilde{m}$ and $\tilde{p}$, and are used in place of m and p in Figure 13 when referring to the number of actual parallel tasks. In the general case, this mechanism would allow efficient parallel simulation of a robotic system where the product, mp, exceeds the number of physical processors on the target parallel computer.

3.5 Results

The partitioning mechanism enables the examination of the effects of various configuration parameters such as the number of chains, spatial tasks, temporal tasks, and integration block size, along with the number of physical processors on the performance of the parallel algorithm. As before, this analysis was carried out for a four manipulator system on both four and eight processor systems. The various ways of partitioning was found to have an effect on the number of RMA computations and interprocessor synchronizations that are performed by a single processor as shown in Table 3.

In this table, the various numbers of spatial ($\tilde{m}$) and temporal ($\tilde{p}$) tasks that lead to load-balanced partitions on four and eight processor systems are listed (note that $\tilde{m} \times \tilde{p}$ is equal to the number of processors for a load-balanced partition). In addition, all of the possible integration Block Sizes are included for each number of temporal tasks. In the fifth and sixth columns of this table, the RMA computation

Table 4: Speedup results for B4PC5.

# CPU	Spatial Tasks, $\tilde{m}$	Temporal Tasks, $\tilde{p}$	T (sec.)	S_p
1	1	1	0.2410	1.00
4	1	4	0.0637	3.78
	2	2	0.0668	3.61
	4	1	0.0731	3.30
8	2	4	0.0390	6.18
	4	2	0.0417	5.78

and synchronization requirements for one pass through the computation shown in Figure 13 are given. Only these two computational requirements are listed since all others (e.g., chain dynamics, predictor and corrector equations) remain unchanged. Since the different Block Sizes lead to the computation of different numbers of trajectory points (equal to the Block Size), the number of RMA computations and synchronizations required to compute four trajectory points are given in the last two columns. For B2PC5 and B1PC5 (Block Sizes of two and one, respectively), this requires two and four iterations of the computation shown in Figure 13.

3.5.1 Parallel Performance

It is evident from the last two columns of Table 3 that the B4PC5 integration method requires the smallest number of synchronizations in each of the partitions considered. In addition, maximizing the number of temporal parallel tasks while using the B4PC5 method also leads to the least amount of RMA computation for a given number of processors. With this result, simulations were first performed using the B4PC5 method and the partitions from Table 3, on four and eight of the CRAY's processors.

A system consisting of four PUMA 560 manipulators was modeled, and a test trajectory with a duration of one second was generated which consisted of lifting a 4kg object 0.8 meters straight up. Then an appropriate joint torque profile was computed which, when applied to the joints of the manipulators, would produce the desired motion. Using a fixed integration stepsize, these torques were used as input into the simulator. To get execution times as close to dedicated operation times as possible, a number of runs were performed during *low*-load periods on the CRAY, and the fastest times are reported in Table 4.

Examining these execution times for a given number of processors, it can be seen that they increase as the number of temporal parallel tasks, $\tilde{p}$, decrease. This is a direct result of the increased numbers of RMA computations that must be performed when $\tilde{p}$ is less than four (refer to Table 3). At the same time, the small variations in the amount of interprocessor synchronization seems to have much less effect on the relative performances of the different partitions. Consequently, the best speedups due to this parallelization, S_p, were achieved by using the greatest amount of temporal parallelism which were as high as 3.78 on four processors. Runs were also made while requesting all eight of the Y-MP's processors and a speedup of 6.18 was still achieved despite an inability to gain dedicated use of these processors in its multi-user environment.

3.5.2 Accuracy Performance

To this point, only the amount of computation required to evaluate a certain number of simulation points has been examined. The B4PC5 method offers a way to simulate the system using four processors in parallel. Because the method extrapolates farther from the known values in the predictor step, it is subject to larger truncation errors for a given stepsize. Consequently, the quality of these simulation results must be taken into account and compared with the smaller block size integration methods as well. To accomplish this, the same trajectory for the four PUMA system was used, and in this case, the error in the final position of the reference member was used as a measure of accuracy for the various integration methods. For the sake of brevity, only this value is reported and its accuracy was found to be representative of the accuracy of the other state variables in the system.

First, serial implementations of the one-, two-, and four-point Sliding Block Methods were used, and fixed integration stepsizes were used to carry out the simulation. The stepsize was varied over an appropriate range so that a profile of execution time versus error could be obtained, and the results are plotted in Figure 14. As expected, the larger Block Sizes lead to lower accuracy. This means that successively smaller stepsizes are required for B2PC5 and B4PC5 to achieve the same level of accuracy as B1PC5, and therefore, they must perform more iterations. This results in higher execution times over a wide range of accuracies.

To quantify subsequent comparisons of relative performance between the different integration methods, an estimate of execution time, T, required to achieve an error of 10^{-6} meters was chosen as a benchmark. Table 5 lists the serial execution times for this benchmark from the data of Figure 14. This table shows speedups of less than one for B2PC5 and B4PC5 which we call their *algorithmic slowdown*, S_a. Using these results, the parallel speedups achieved with these methods, S_p, must be reduced by about one-half for B4PC5, and by one-fourth for B2PC5. The results are

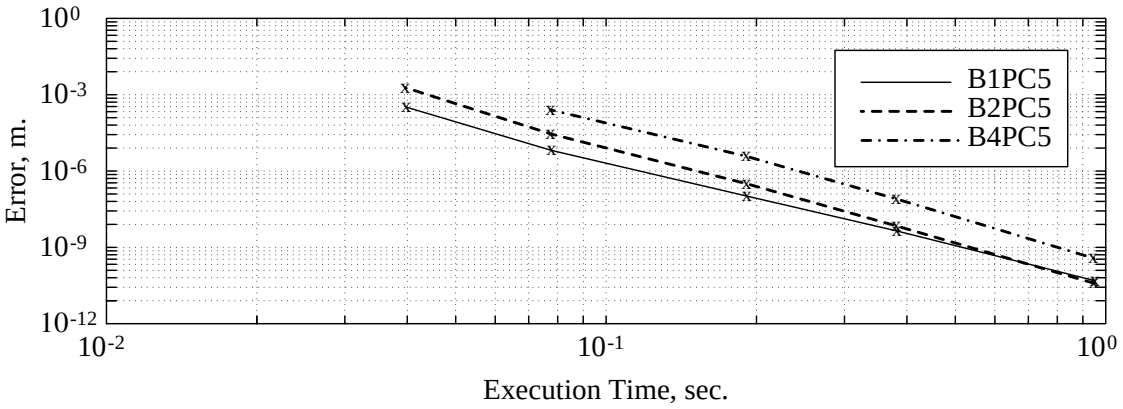

Figure 14: Serial accuracy performance of integration methods.

Table 5: Algorithmic slowdown with larger block sizes.

Block Size	T (sec.)	S_a
1	0.117	1.000
2	0.152	0.770
4	0.241	0.485

the speedups of these methods over the best serial method (B1PC5) for the given accuracy (10^{-6} m) and are a measure of their *accuracy performance*.

In Table 6, the accuracy performance data for all of the possible partitions on four and eight processors are presented. The same benchmark is used in which the execution times, T, are for the parallel simulations achieving the error of 10^{-6} meters. S_p is the speedup of the simulation over its own serial performance, S_a is the algorithmic slowdown associated with the Block Size, and S_T is the speedup over the best serial implementation (its accuracy performance).

These results lead to a much different conclusion than when the speedup, S_p, alone was considered. Table 3 showed that synchronization requirements increased for the computation of a given number of simulation points as the Block Size decreased. At the same time, the accuracy of the method increased requiring the computation of fewer overall points. The final result is that the *smaller* block sizes lead to better effective speedups. Since the system under consideration had four

Table 6: Accuracy performance for various partitions.

# CPU	Spatial Tasks, $\tilde{m}$	Temporal Tasks, $\tilde{p}$	Block Size	T (sec.)	S_p	S_a	S_T
4	1	4	4	0.0637	3.78	0.485	1.83
	2	2	4	0.0668	3.61	0.485	1.75
			2	0.0444	3.42	0.770	2.63
	4	1	4	0.0731	3.30	0.485	1.60
			2	0.0463	3.28	0.770	2.53
			1	0.0377	3.10	1.000	3.10
8	2	4	4	0.0390	6.18	0.485	3.00
	4	2	4	0.0417	5.78	0.485	2.80
			2	0.0295	5.15	0.770	3.97

chains, the serial B1PC5 method was the best on four processors and the resulting speedup was 3.1. On eight processors, where B1PC5 could not be used and still have eight parallel tasks, the B2PC5 method showed the greatest speedup at 3.97. From these results, it appears that B4PC5 has little or no advantage over the other integration methods. However, when a smaller number of chains is simulated, this method allows a greater number of parallel tasks to be generated than the other methods and would enhance the performance.

3.6 Summary and Conclusions

In this chapter, two methods for achieving effective parallelization for dynamic simulation of multiple chain systems on a general-purpose parallel computer were presented. One approach that was discussed in [36] for parallel simulation of a single chain system was based on temporal parallelism achieved with the use of a parallel numerical integration method. The topology of multiple chain systems and the resulting form of the dynamics algorithm introduces a second form of parallelism. Called spatial parallelism, this results from the ability to compute the dynamics of individual chains simultaneously.

Both forms of parallelism were combined in an effort to achieve greater speedups. For the cases where the number of resulting parallel tasks was greater than the number of available processors, a partitioning mechanism was implemented to divide the

computational task into fewer parallel tasks while minimizing the amount of interprocessor communication and maintaining a load balance. Runtime performance results were collected for a four PUMA system on four and eight processors of a CRAY Y-MP8 for all of the reasonable task partitions.

The results show that the best effective speedups were obtained using the integration method with the smallest block size while partitioning the computation so that the number of parallel tasks was equal to the number of processors. On four processors of the CRAY, the simulation exhibited the greatest speedup (3.1) with spatial parallelism only, and the use of the serial predictor-corrector integration method. The greatest speedup on eight processors (3.97) was also achieved with the use of full spatial parallelism. In this case, a two-point parallel integration method had to be used to achieve the eight desired parallel tasks. It also appears that the four-point integration method would be beneficial if sixteen processors were available.

An additional benefit to these forms of parallelism is that they do not preclude any of the previous work mentioned in the introduction that dealt with the fine-grain parallel algorithms for computation of the robot dynamics quantities. The special-purpose architectures required could be set up in parallel and used in conjunction with the methods discussed in this chapter. The resulting combination of parallel computation could be thought of as occurring in three dimensions, where the third comes from parallelism in the computation *within* each chain, and shows promise for even greater speedups.