[26] D. M. Nicol. The cost of conservative synchronization in parallel discrete-event simulations. *Journal of the ACM*, 40(2):304–333, April 1993.

[27] D. M. Nicol and D.R. O'Hallaron. Improved algorithms for mapping parallel and pipelined computations. *IEEE Trans. on Computers*, 40(3):295–306, 1991.

[28] D. M. Nicol and P.F Reynolds, Jr. Optimal dynamic remapping of data parallel computations. IEEE Trans. on Computers, 39(2):206–219, February 1990.

[29] P. F. Reynolds, Jr. Comparative analyses of parallel simulation protocols. In *Proceedings of the 1989 Winter Simulation Conference*, Washington, D.C., December 1989.

[30] L. A. Sanchis. Multiple-way network partitioning. *IEEE Trans. on Computers*, 38(1):62–81, January 1989.

[31] R. Sedgewick. *Algorithms*. Addison-Wesley, New York, 1988.

[32] H. Sellami and S. Yalammanchili. Efficient parallel simulation of marked graphs. In *Proceedings of the SCS Summer Simulation Conference*, pages 328–333, July 1992.

[33] H. Sellami and S. Yalammanchili. Consevative parallelsimulation of a class of multiprocessor simulation models. Technical Report TR-GIT/CSRL-93/02, Georgia Tech. School of Electricaaal and Computer Engineering, July 1993.

[34] T. Som. *Allocation of Processing Power in Optimistic Parallel Discrete-Event Simulation*. PhD thesis, Syracuse University, 1992.

[35] R. E. Tarjan. *Data Structures and Network Algorithms*. Society for Indusrial and Applied Mathematics, Philadelphia,PA, 1983.

[36] G. Thomas and J. Zahorjan. Parallel simulation of performance petri nets: Extending the domain of parallel simulation. In *Proceedings of the 1991 Winter Simulation Conference*, pages 564–573, Phoenix, Arizona, December 1991.

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

[13] D. R. Jefferson, B. Beckman, F. Wieland, L. Blume, M. DiLorento, P. Hontalas, P. Reiher, K. Sturdevant, J. Tupman, J. Wedel, and H. Younger. The Time Warp Operating System. *11th Symposium on Operating Systems Principles*, 21(5):77–93, November 1987.

[14] D. Kumar and S. Harous. An approach towards distributed simulation of timed petri nets. In *Proceedings of the 1990 Winter Simulation Conference*, pages 428–435, New Orleans, LA., December 1990.

[15] Sigurd L. Lillevik. The Touchstone 30 gigaflop DELTA prototype. In *Distributed Memory Computer Conference 91*, pages 671–677. IEEEPRESS, April 1991.

[16] B. D. Lubachevsky. Efficient distributed event-driven simulations of multiple-loop networks. *Communications of the ACM*, 32(1):111–123, 1989.

[17] T. Murata. Petri nets: Properties, analysis, and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[18] B. Nandy and W. Loucks. An algorithm for partitioning and mapping conservative parallel simulation onto multicomputers. In 6^{th} *Workshop on Parallel and Distributed Simulation*, volume 24, pages 139–146. SCS Simulation Series, Jan. 1992.

[19] G. L. Newhauser and L.A. Wolsey. *Integer and Conbinatorial Optimization*. Wiley and Sons, New York, 1988.

[20] D. M. Nicol, C. Micheal, and P. Inouye. Efficient aggregaton of multiple LP's in distributed memory parallel simulations. In *Proceedings of the 1989 Winter Simulation Conference*, pages 680–685, Washington, D.C., December 1989.

[21] D. M. Nicol and S. Roy. Parallel simulation of timed petri nets. In *Proceedings of the 1991 Winter Simulation Conference*, pages 574–583, Phoenix, Arizona, December 1991.

[22] D. M. Nicol and R. M. Fujimoto. Parallel simulation today. *Annals of Operations Research*. To appear.

[23] D. M. Nicol. *The Automated Partitioning of Simulations for Parallel Execution*. PhD thesis, University of Virginia, August 1985.

[24] D. M. Nicol. Parallel discrete-event simulation of FCFS stochastic queueing networks. *SIGPLAN Notices*, 23(9):124–137, September 1988.

[25] D. M. Nicol. Performance bounds on parallel self-initiating discrete event simulations. *ACM Trans. on Modeling and Computer Simulation*, 1(1):24–50, 1991.

Connection Machine CM-1 routing network on 16 processors of an Intel iPSC/860, and on a simulation of a slotted-ring architecture that is executed on up to 64 processors of the Intel Touchstone Delta. Significant performance benefits are observed, and the effectiveness of (and need for) dynamic remapping clearly demonstrated.

Acknowledgements

We thank Subhas Roy for his programming contributions.

References

[1] G. S. Alamsi and A. Gottlieb. *Highly Parallel Computing.* Benjamin/Cummings, Redwood City, CA, 1989.

[2] F. Baccelli and M. Canales. Parallel simulation of stochastic petri nets using recurrence equations. *ACM Trans. on Modeling and Computer Simulation*, 3(1):20–41, January 1993.

[3] S. H. Bokhari. Partitioning problems in parallel, pipelined, and distributed computing. *IEEE Trans. on Computers*, 37(1):48–57, January 1988.

[4] K. M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Trans. on Software Engineering*, 5(5):440–452, September 1979.

[5] G. Chiola and A. Ferscha. Distributed simulation of petri nets. *IEEE Parallel & Distributed Technology*, 1(3):33–50, August 1993.

[6] H.-A. Choi and B. Narahari. Algorithms for mapping and partitioning chain structured parallel computations. In *Proceedings of the 1991 Int'l Conference on Parallel Processing*, St. Charles, Illinois, August 1991.

[7] S. Eick, A. Greenberg, B. Lubachevsky, and A. Weiss. Synchronous relaxation for parallel simulations with applications to circuit-switched networks. In *Proceedings of the 1991 Workshop on Parallel and Distributed Simulation*, pages 151–162, Jan. 1991.

[8] G. A. Frank, D.L. Franke, and W.F. Ingogly. An architecture design and assessment system. *VLSI Design*, pages 30–38, August 1985.

[9] R. M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, October 1990.

[10] D. W. Glazer. *Load Balancing Parallel Discrete-Event Simulations.* PhD thesis, McGill University, May 1992.

[11] M. A. Iqbal and S. H. Bokhari. Efficient algorithms for a class of partitioning problems. Technical Report 90-49, ICASE, July 1990.

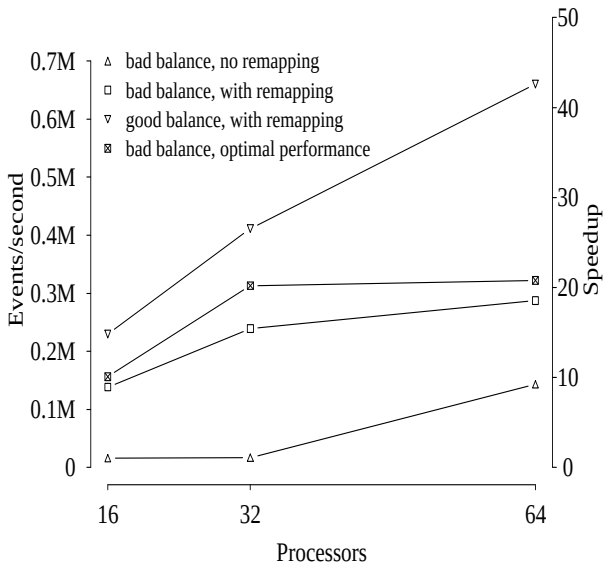


Figure 3: Performance on slotted ring network example

two LPs gets two workload LPs.

We also see that the dynamic remapping mechanism comes close to achieving the optimal performance possible, given the unbalanced workload. Performance of the balanced workload is nearly perfect for 16 and 32 processors; it falls away at 64 processors owing to the low number of events performed on each processor between synchronizations (50).

6 Conclusions

This paper studies the problem of automatically parallelizing the discrete-event simulation of large timed Petri-nets executing on parallel architectures. The methods we described have been implemented in a tool where one designs a Petri-net using a graphical tool, and then all remaining steps for parallelization are performed automatically.

We describe a synchronization algorithm and automated load balancing techniques, both static and dynamic. We present a new static mapping algorithm, and study its properties analytically. This algorithm is not restricted to TPN simulations, it applies to more general parallel computations. We study the effectiveness of our methods on the performance of a simulation of the

the gap between the two performance curves. On this problem the difference is less than 10%, a difference that is increasingly amortized as the problem size grows. With increasing problem size performance gets better, but it is clear that if the growth trend continues, by dimension 9 the event rate is close to its maximal level. The fact that this occurs at a speedup less than 12 is due to the cost of communication on the iPSC/860, which is quite high relative to the speed of the CPU. These same speedups on the more balanced Intel iPSC/2 are nearly 20% better (but the iPSC/2's CPU is a factor of 7 slower on this problem!).

We have also investigated our synchronization algorithm on various TPN models of mesh-based architectures; these results are reported in [21]. Like the CM-1 model, these are essentially self-balancing (at the time of that paper we had not yet developed the mapping and remapping methods). This study also shows increasing performance as the problem size grows, but also shows the dependence of good performance on a favorable computation to communication ratio. The cost of communication on the Intel iPSC/2, iPSC/860, and Delta architectures we've used is high enough to require substantial computation on average between communications.

We use the slotted ring model to better explore the benefits of dynamic remapping. As a test case for unbalanced workload, we created a model with 64 workload LPs where the first 6 and last 47 LPs generate events at a rate of 50 events per unit simulation time, while the remaining 17 LPs generate 500 events per unit simulation time. This particular assignment of workload stresses the static mapping algorithm, as adjacent heavy workloads are harder to distribute under its linear ordering constraints. However, the initial assignment estimates workloads based solely on topology, and is unable to distinguish between heavy and light workloads. Furthermore, the ring LP has many more places and transitions than a workload LP, but ends up having nowhere near the same event intensity. As a consequence we can usually expect the initial mapping to be very bad. Finally, the unbalanced workload model is simple enough to compute an upper bound on the speedup possible, by assuming the LPs are distributed optimally (an easy calculation by hand), communication costs are free, and the initial mapping is perfect.

Figure 3 plots the results of simulating this model on 16, 32, and 64 processors of the Intel Touchstone Delta. All runs generated approximately 10 million events. The vertical axes are as before, this time with the event execution rate expressed in millions of events per second. We plot three sets of data associated with "bad balance", the unbalanced workload described above. For the purposes of comparison we also plot data associated with a "good balance" model where all LPs have the same weight (50 events per unit simulation time). In all these runs communication with the ring LP is infrequent. This allows us to isolate the effects of load imbalance from communication costs. We still do have inescapable communication costs due to synchronization, which occurs every unit of simulation unit.

On all runs where remapping was employed, remapping was chosen very shortly into the run, as it was quickly evident that a new mapping based on event count measurements was superior. Although theoretically possible, no subsequent remappings were performed. The necessity of dynamic remapping is clearly seen by examining performance when remapping is disabled. The jump in performance at 64 processors is a consequence of the mapper initially always using as many processors as are available. The worst that can happen (which did) is that the one processor assigned

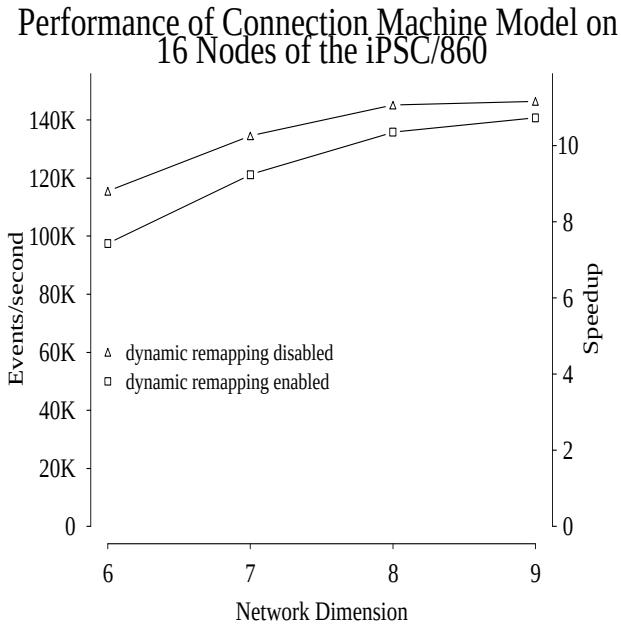


Figure 2: Performance on Connection Machine global router example

were given parameters to cause the greatest amount of interprocessor communication. Each problem size was executed long enough to simulate over ten million events, and was simulated both with remapping logic enabled, and disabled. The dynamic remapping mechanism never caused the initial mapping to be abandoned, even though at some individual remapping assessments it appeared (temporarily) that some gain might be achieved by rearrangement. This illustrates the essential safety offered using the Bayesian filter.

The left vertical axis demarks the aggregate average number of events executed per second (in units of a thousand). The right vertical axis delineates the corresponding speedup, measured using an optimized serial code that employs the same Petri net event processing logic, but uses a splay-tree priority list to manage events, and suffers no unnecessary overheads inherented from the parallel version. All speedup measurements are computed using the measured serial rate on a six dimensional problem, as the larger models would not fit in one node's memory.

The overhead of gathering workloads and projecting remapped performance can be seen as

distribute it to the others. The load time is large, and so many processors (which are shared with many users and which are charged for) are idle during the loading. A more sophisticated implementation could use the concurrent file system.

mimics handshaking that ensures a buffer is available for a message before it is sent. The dimension d of the hypercube parameterizes this model. A model with 6 dimensions has over 10,000 places and 10,000 transitions. A model with 8 dimensions has over 100,000 places and transitions.

The slotted ring model is comprised of some N LPs that model fine-grained workload, we call these *workload generators*. A workload generator is basically a loop within which a set of tokens circulate. Every pass through the loop (five events) the workload generator randomly decides whether to communicate over the ring. The ring itself is an LP, hence the communication topology of the system is a tree with one root (the ring) and N leaves. The simulated communication is non-blocking and serves simply to generate some simulation workload (a round-trip for a message) for the ring LP. We control workload intensity by varying the firing time on loop transitions. The smaller the firing time, the more simulation work per unit simulation time is required. This model clearly demonstrates the need for dynamic remapping, because reasonable workload estimates derived from the topology alone fail miserably to balance the load. A model with 64 workload LPs and one ring LP has 4672 places and an equal number of transitions.

The timing delays in both models are based on realistic disparities between computation and communication times. This has a definite impact on the synchronization protocol. For example, certain "slow" transition firings in the CM-1 network model are two orders of magnitude larger than the smallest delays on firings that cross processor boundaries. There is a very sizable lag (in simulation time) between the last "fast" transition to fire, and the firing of the slow transition. It would be disastrous to simply advance time by the minimum boundary firing time amount each window, for many windows would contain no events. However, the technique of adding the least on-processor event time-stamp to the minimum boundary firing time effectively skips over these periods.

The simulations were conducted on the YAWNS (Yet Another Windowing Network Simulator) parallel simulation testbed [20, 26], implemented on the Intel family of multiprocessors. We present data from runs executed on the Intel iPSC/860, and upon the Intel Touchstone Delta [15], a large-scale multiprocessor also based on Intel's i860 CPU. The time spent in the pair/match/mapping algorithm was dominated by the I/O time to read the network description, and so proved to be inconsequential.

The CM-1 routing network example defines an LP naturally as the submodel associated with one router node. The problem communication topologies thus forms a hypercube. Since all LPs are structurally identical, any topological measure of workload will assign the same workload to each LP, and the merge/match/map algorithm maps it optimally under the assumptions of uniform communication and execution costs. Moreover, under the homogeneous model assumptions, the average execution cost for every LP was identical. No long term load imbalances could be expected to develop, so that dynamic remapping should probably not be used.

Figure 2 plots the measured performance of the CM-1 model on sixteen processors of the Intel iPSC/860[1], as a function of the hypercube dimension ($d = 6, 7, 8, 9$). The various random decisions

[1] We have run this model on the Intel Delta as well, but it is impractical to do so on large models. The present implementation allocates all processors, then has one processor read the model description from a single disk and

same remapping decision as all the others. We attempt to distribute workload information with relatively little communication, as follows. In a first step we compute the maximum and minimum LP loads throughout the system, using a software combining tree that operates on vectors (e.g., a global reduction min on vectors). Such reductions are supported by fast library routines on Intel multicomputers. Next, every processor discretizes the range of LP workloads into 16 levels. Then, for each of its LPs, a processor generates a 4-bit code describing which of those levels best describes the LPs load. These 4-bit codes are inserted into an array, initially empty, of codes for all LPs. The processors engage in a global bitwise-OR reduction on this table, which serves to distribute the code information to every processor. From these codes the processors can now reconstruct the same approximate workload levels as any other processor, and execute the remapping logic based on these estimates. Every processor computes which LPs it must shed, and which ones it will receive. The actual disengagement of an LP from one processor and integration into another involves some work related to bundling and unbundling state information, and keeping the processors data structures up to date.

We have chosen to ignore communication costs in the mapping step, for two reasons. First, the number of different communication values is quadratic in the number of LPs, which implies a great deal of information to gather and distribute at run-time. Intuition suggests that if the linear ordering is successful in keeping closely communicating LPs together, then ignoring the communication costs while mapping should not grossly affect performance. This does beg the issue of recovering from a bad linearization.

Three areas of the scheme above bear further investigation. The bit-vector approach to distributing load information is efficient for small-to-medium numbers of LPs, after which the communication cost can be overly high. Also, certain parameters may need dynamic adjustment, particularly the frequency of computing tentative mappings, and the decay parameter for workload averaging. Finally, dynamic re-linearization needs investigation.

5 Empirical Study

Our empirical study considers TPN models of a slotted ring network, and of the Connection Machine CM-1 global routing network. Our models do not attempt to accurately capture all aspects of behavior; instead they are intended to be representative of large TPN problems to which one might apply parallel processing.

The CM-1 network model is based on the description in [1]. It captures the dimension-by-dimension structure of message-passing, the effects of limited buffer space, and the interaction between a router and the mesh of processors that directly access it. Every node of the global network serves 16 PEs; a PE signals its decision to communication (made randomly) by placing a token in a specified location. For each message, a dimension is chosen and the message is enqueued to be sent across that dimension. When a message arrives at a new PE, a random decision is made to either absorb the message (modeling its terminal arrival), or to send it through another dimension, chosen uniformly at random among all dimensions higher than the one through which it came. The model explicit mimics the petit and grand cycle nature of the CM-1, and explicitly

4.2 Dynamic Remapping

Our approach to dynamic remapping has essentially been laid out before, in [28], with an emphasis on physical computations that exhibit distinct phases. The issue there is to determine with sufficient confidence that a phase change has occurred and that performance will benefit from remapping. The general approach is to periodically consult an "oracle" that judges whether it is worthwhile to remap now. The oracle's decision is not immediately acted upon though, it is used to update (via Bayes' Theorem) a *gain probability* that performance will improve by remapping now. The optimal decision policy was shown to be a threshold policy—if the gain probability is larger than some step-specific threshold, then one ought to remap. As computation of the optimal decision thresholds proved to be impractical, a heuristic was proposed to use a constant, high, threshold.

We apply this work to the present context, as follows. Periodically, just prior to the beginning of a window's processing the processors all coordinate to make a remapping decision. The decision is based on measurements of the average processing cost undergone so far by every LP. As the simulation runs, a processor keeps track of the number of events executed so far on behalf of each LP. The processor also keeps track of the total time spent so far processing events, by measuring the time spent by a routine which in one call processes all the events done by a processor in a window. With these figures we compute the average time spent by each LP processing events in a window; the average may be exponentially decayed to allow sensitivity to time-varying averages. The averages become the LP weights for the static mapping algorithm. It makes a great deal of sense to balance based on these per-window averages, since windows are separated by barrier synchronizations. Furthermore, given these averages, a prospective mapping, and knowledge of the simulation's termination time (in simulation time) we may estimate the remaining time required to complete the simulation under the assumed mapping. Our approach then is to have an oracle routine compute the best mapping given the present LP workload averages, then predict the expected time to remap (with an estimated remapping cost of 1 second) and finish the computation under the new mapping. The oracle similarly predicts the expected finishing time if one does not remap. The oracle recommends remapping if its projections suggest a performance improvement of at least 10%. This judgement is used to update the gain probability. The purpose of the 10% padding is to protect from underestimating the remapping cost (which isn't known until it is observed). Since we have observed that the initial mapping can be truly inferior, we modified the heuristic so that a remapping is performed automatically if the oracle judges the new mapping to be twice or more faster than the old. In our experiments we have seen remapping triggered both by the gain probability crossing the threshold (which requires 2-3 consecutive positive oracle judgements), and by the twice-as-good rule. The remapping logic is not disabled after the initial remapping; if the initial remapping decision turns out to be very wrong it is still be possible to correct it. Similarly, if the workload has a time-varying average, then the decayed sample averages can reflect this, and trigger a remapping.

Our implementation of this policy deserves comment, as it would be easy to implement the logic inefficiently. The basic idea is to provide every processor with an estimate of every LPs workload, and then have every processor execute exactly the same code and make exactly the

Proof: Consider a processor assigned any x LPs. The proof of lemma 4 shows that the sum of edges between LPs on that processor is no greater than $(x/2)\log x$; hence there are at least $kx - 2x\log x$ edges to LPs on other processors. The function

$$f(x) = e_w x + kx - 2x\log x$$

is thus a lower bound on the cost of assigning x LPs to a processor. Note that the bound is achieved if the set forms a hypercube. Considering x as continuous, we have

$$f'(x) = e_w + k - 2\log x - \frac{2}{\ln 2}.$$

Note that $\log x$ is maximized when equal to k, hence f is increasing over $x \in [0, 2^k]$ if $e_w \geq k + 2/\ln 2$. If $x_1, x_2, \ldots x_{2^j}$ are workload assignments ($x_i \geq 0$, $\sum_{i=1}^{2^j} x_i = 2^k$ then the function

$$g(x_1, \ldots, x_{2^j}) = \max\{f(x_1), f(x_2), \ldots, f(x_{2^j})\}$$

is a lower bound on the bottleneck cost of the assignment. Since f is increasing, g is minimized when the maximum x_i is as small as possible—that is, when the x_i's are identically 2^{k-j}. This situation is achieved when the LPs are partitioned into 2^j hypercubes, furthermore the value of g then is also exactly the bottleneck. If G is enumerated naturally, the pair/merge/map algorithm will produce this assignment. ∎

Other situations where the pair/merge/map approach finds optimal solutions occur as a result of the definition of the bottleneck cost. Any solution that minimizes the bottleneck can be embedded in a linearization. For example, given the optimal mapping we can renumber the LPs assigned to processor 1 starting at 1, then carry over the enumeration to LPs assigned to processor 2, and so on. However, a large number of linearizations are equivalent in the sense that the chain mapping algorithm will find the optimal bottleneck on them. For example, any permutation of the processor ordering does not affect the bottleneck cost, and does not confuse the chain mapping algorithm. Likewise, within the LPs assigned to a processor there is an insensitivity to their ordering within the processor. The net effect is that given an optimal solution and an associated linearization π^{opt}, there are a number of permutations of π^{opt} that will not affect the sets of LPs that are co-resident. Given any one of these linearizations the chain mapping algorithm with discover the optimal bottleneck.

Lemma 7 *For any mapping problem involving m LPs and P processors and minimized bottleneck cost b, there are at least $P! \times \Gamma(m/P)^P$ different linearizations upon which the chain mapping algorithm will discover a solution with cost b.*

Proof: Suppose that a solution minimizing the bottleneck value b assigns n_i LPs to processor i, for $i = 1, 2, \ldots, P$. From the discussion above there are at least $P! \times \prod_{i=1}^{P} n_i!$ different linearizations for which the chain mapping algorithm will produce a solution with bottleneck cost b. In the continuous domain, $\Gamma(n_i) = n_i!$, and for fixed $\sum_{i=1}^{P} n_i = m$ the product $\prod_{i=1}^{P} \Gamma(n_i)$ is minimized when $n_1 = n_2 = \ldots n_P = m/P$. ∎

which completes the induction. Now observe that when $x = 2^i$ the bound is met, and is met by sets that themselves form hypercubes (which have $i2^{i-1}$ edges contained within them). Now at every step i the pair/merge algorithm merges hypercubes of dimension $i-1$ into hypercubes of dimension i (a consequence of G being ordered naturally). Thus $S_j(\pi)$ is maximized for each j. ∎

Rings and hypercubes are special cases of k–ary n–cube networks. We believe our results might be generalized to such networks where k is a power of two.

4.1.2 Chain Mappings

Suppose that some linearization of the LPs is given. The most general formulation of the remaining mapping problem allows *any* two LPs to have non-zero communication costs. A dynamic programming formulation solves the problem in $O(Pn^2)$ time, P being the number of processors. To see this, let $C(j,p)$ be the optimal bottleneck cost achievable mapping LPs 1 through j onto p processors. Then the principle of optimality asserts that

$$C(j,p) = \min_{i<j}\{\max\{C(i,p-1), \sum_{k=i+1}^{j} e_k + \sum_{k=i+1}^{j} \sum_{m\le i, m>j} w_{km}\}\}.$$

Note that this solution permits non-adjacent LPs to have non-zero communication costs, whereas previous solutions are restricted to the alternate case. Not surprisingly, under the more constrained assumption the algorithms have lower complexity.

There are instances of the mapping problem where the pair/merge/map approach will find the minimal bottleneck mapping. We show this for rings and hypercubes with unit communication costs and large enough common execution costs e_w for each LP. The second result enumerates the many equivalent linearizations derivable from a given optimal mapping. This result reflects the resilency of restricting our attention to chain mappings.

The first conclusion is obvious.

Lemma 5 *Let (G,E) be a ring, enumerated naturally, with unit edge weights. Suppose every vertex has common weight e_w. Then for any power-of-two number of processors P, the pair/merge/map algorithm minimizes the bottleneck over all possible partitions of the ring into nonempty P sets.*

Proof: Under any mapping, every processor has a communication cost of at least two. The linearization produced by the pair/merge algorithm gives every processor a communication cost of exactly two. The chain mapping algorithm will map no more than $\lceil |G|/P \rceil$ LPs to any processor, yielding a bottleneck cost of $2 + e_w\lceil |G|/P \rceil$, which is optimal. ∎

The second conclusion shows that in balanced hypercubes, for moderate values of e_w it is optimal to assign equal sized hypercubes of smaller dimension to each processor—as does pair/merge/map.

Lemma 6 *Let (G,E) be a hypercube, enumerated naturally, with unit edge weights, and $|G| = 2^k$. Suppose every vertex has common weight $e_w \ge k + 2/(\ln 2)$. Then for any power-of-two number of processors $2^j \le 2^k$, the pair/merge/map algorithm minimizes the bottleneck over all possible partitions of the hypercube into up to 2^j pieces .*

$O(nB\log(2^{i-1}B))$; the cost sum over all $\log n$ steps is

$$\begin{aligned} O\left(\sum_{i=1}^{\log n} nB\log(2^{i-1}B)\right) &= O\left(nB\sum_{i=1}^{\log n}(\log B + \log 2^{i-1})\right) \\ &= O\left(nB(\log B)\log n + nB\log^2 n\right) \\ &= O\left(nB\log^2 n\right). \end{aligned}$$

On certain graphs the pair/merge linearization is optimal in the sense that the linearization it finds simultaneously maximizes $S_j(\pi)$ for all j. We demonstrate this result for rings and hypercubes. In both cases the communication topology is very regular, and we assume unit edge weight costs. This situation corresponds to an initial mapping of a homogeneous Petri net model of such architectures, prior to run-time measurement of execution and communication costs. This is still an important problem, because even with measured costs the model may well be uniformly weighted. This in fact was an unexpected consequence of our CM-1 router example.

LP enumeration has a definite affect on the matches made, and a little care is required for our optimality results to hold. For the specific cases we consider we suppose the LPs to be enumerated in a "natural" way. We presume a ring is enumerated so that adjacent vertices in the enumeration share a communication edge; we presume the usual enumeration of a hypercube where vertex i and j share an edge if and only if the Hamming distance between i and j is exactly one.

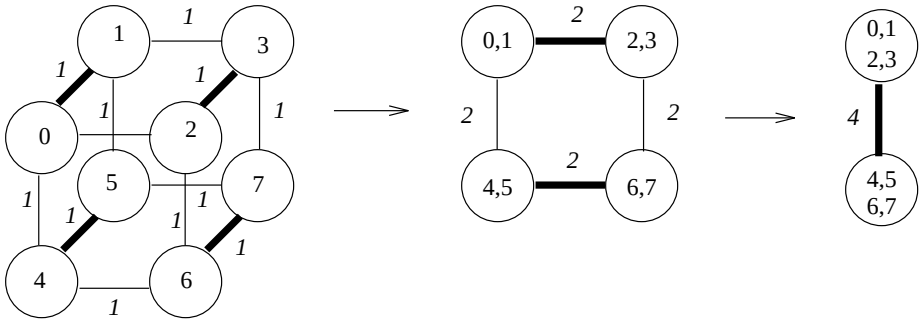
Lemma 3 *Let (G,E) be a ring, enumerated naturally, with unit edge weights, and $|G| = 2^k$. Then for all $j = 1, 2, \cdots, k$, any linear ordering π produced by the pair/merge algorithm maximizes $S_j(\pi)$.*

Proof: For any integer j, it is obvious that the partition into 2^{k-j} pieces maximizing the number of edges between vertices in a common partition element is obtained by grouping the first 2^j vertices together, then the next 2^j, and so on. This is precisely the grouping defined by the pair/merge algorithm. ∎

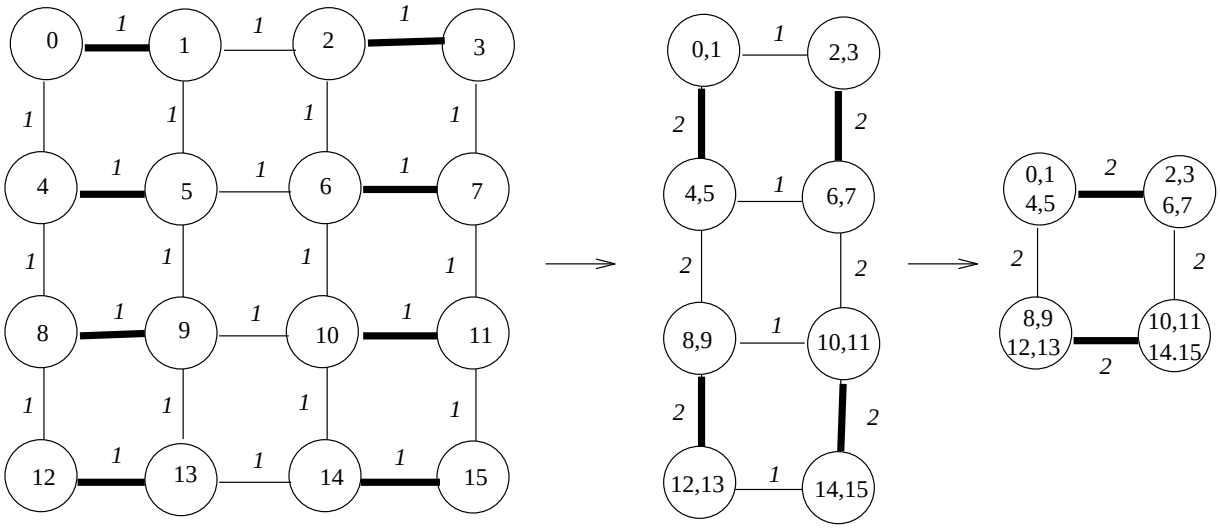
Lemma 4 *Let (G,E) be a hypercube of dimension k, suppose that G is enumerated naturally, and that all edges have unit weight. Then for all $j = 1, 2, \cdots, k$, any linear ordering π produced by the pair/merge algorithm maximizes $S_j(\pi)$.*

Proof: We first induct on x to prove that the number of edges between members of any subset of x vertices is no greater than $(x\log x)/2$. The base case of $x = 1$ is trivially satisfied. Suppose then that the claim is true for any subset of size $x-1$ or smaller, and choose any subset A with x vertices. Split A evenly into two subsets A_1 and A_2. The number of edges between vertices in A is the sum of the edges on A_1 plus the edges on A_2 plus the edges between them. There are at most $\lfloor x/2 \rfloor$ edges between them, and by the induction hypothesis the sum of edges on A_1 and on A_2 is no more than $(x\log(x/2))/2$. Therefore the number of edges on A is no more than

$$\lfloor x/2 \rfloor + \frac{x\log(x/2)}{2} \leq \frac{x\log(x)}{2},$$



Match/Merging a Hypercube with unit edge weights



Match/Merging a Mesh with unit edge weights

Figure 1: Behavior of match/merge algorithm on a hypercube, and mesh

requires $\Omega(n^2 \log n)$ time. However, these costs drop to $O(nB)$ and $\Omega(nB \log B)$ if an LP communicates with no more than B others. This makes a real difference when n is large, and $B \ll n$. It is also true for our CM-1 router example.

The cost of a matching step is dominated by the cost of computing and sorting inter-LP communication costs. The first step exacts an $O(nB \log B)$ cost. At the second step the number of super-LPs involved is halved, but in the worse case the number of connections a super-LP has doubles. This gives the second step a cost of $O(nB \log(2B))$. In general the i^{th} step costs

Proof: Let π^{opt} be a linearization that maximizes the sum of weights between adjacent LPs, and renumber the LPs with respect to π^{opt}. Let S_{even} be the set of edges of the form $(i, i+1)$, for i even, and let S_{odd} be the set of edges of the form $(i, i+1)$ for i odd. Both S_{even} and S_{odd} are matchings, hence the sum of edges in either is no greater than $S_1(\pi^h)$. This gives

$$\omega^{opt} \leq 2S_1(\pi^h) \leq 2\omega^h,$$

from which the result follows. ∎

This result is essentially the same as a similar one for a TSP heuristic based on a matching step [19].

To accelerate solution time (possibly at the expense of solution quality) we normally use a $O(n)$-time approximation to the maximal weight matching step, called a "stable" matching. This is one in which it is not possible to find matched pairs (A, B), (C, D) such that

$$\max\{w_{AC} + w_{BD}, w_{BC} + w_{AD}\} > w_{AB} + w_{CD}.$$

Such an algorithm is obtained from a modification to the stable marriage problem [31] for bipartite graphs. The sense of the original stable marriage solution is to loop over all "males", each one attempts to become engaged by proposing to the "females" in decreasing order of preference. If a suitor finds a previously engaged debutante such that the debutante prefers the new suitor to her engaged, the old engagement is broken and a new one forged, and the jilted suitor is left to pick up and continue his search for a mate.

Restated in our context, every LP orders all other LPs with which it communicates. Higher communication implies higher preference. We cannot immediately apply the stable marriage algorithm, as we lack distinct sexes. A minor modification to the algorithm has the effect of selecting sexes: once an LP becomes engaged playing the role of a debutante, it is not later considered to be a suitor. This effectively separates the LPs into two equal sized groups; the matching found will be stable with respect to the sex roles discovered during the process.

The modified method is called the *pair/merge* algorithm. The matchings it produces are identical to those of the match/merge algorithm on certain networks (e.g., rings, meshes, hypercubes) with unit edge costs, provided the match/merge algorithm is encoded with suitable tie-breaking rules.

The effects of pair/merge using stable matching on a 3-dimensional hypercube and on a 2-dimensional mesh are illustrated in Figure 1. Ties between equal weighted edges are resolved in favor of LPs with lower index numbers (this is an important aspect of the match process when matching balanced graphs). We see that the hypercube with equal weight edges collapses along dimension lines, and that the mergings in the mesh alternate between dimensions. Linearizations based on these processes are clearly dealing with the global structure of the graph rationally.

Petri-net models of parallel architectures exhibit high connectivity locally, for instance, reflecting the interconnection pattern of a modeled parallel architecture. This feature has an impact on the algorithmic cost of linearization. If each of n LPs communicates with every other LP then there are $O(n^2)$ distinct inter-LP communication costs to calculate, and the computation of preferences

The ordering of merged two super-LPs ((A,B) vs. (B,A)) is specified in terms of inherited index numbers. This is somewhat arbitrary. While values of $S_j(\pi)$ are insensitive to this choice, other objectives (such as distributing workload evenly along the chain) are not. The match/merge process is depicted naturally by a binary tree whose leaf vertices represent original LPs, and where parent-child relationships reflect merging decisions. Given the merging decisions, each possible linearization is uniquely determined by a set of decisions that order each interior vertex's children. As there are $n-1$ interior vertices, then there are 2^{n-1} different linearizations derivable from a given set of match decisions. If the decision is made that child A precedes child B, the effect is that all LPs descended from A will precede all LPs descended from B in the ordering. It might be useful to delay ordering decisions until the tree is constructed, and then choose orderings to spread out the workload as well as to better localize the communications. We have not yet explored this problem, but feel it is worthy of future attention.

We can bound the deviation from optimal of the linearizations produced by this method. Let π^h be a linearization produced by our heuristic, and for every j let π_j^{opt} be a linearization that maximizes $S_j(\pi)$ over all linearizations π.

Lemma 1 *For all* $j = 1, 2, \ldots, \lceil \log |G| \rceil - 1$,

$$\frac{S_j(\pi_j^{opt})}{S_j(\pi^h)} \leq (2^j - 1)\frac{S_1(\pi^h)}{S_j(\pi^h)}.$$

Proof: Fix j, and let S_1 be the set of first 2^j vertices under π_j^{opt}, S_2 be the second set, and so on. The vertices in every set S_i have up to $2^j(2^j-1)/2$ number of edges between them, with nonnegative weights. These edges can be partitioned into 2^{j-1} disjoint sets $E_{i,k} = \{(u,v) | u = 2^i + e < v = 2^i + e + k, e = 0, 1, \ldots, 2^{j-1}\}$, for $k = 1, 2, \ldots, 2^j - 1$. $E_{i,k}$ is thus the set of edges in S_i between vertices that are k distance, without wrap-around. Each $E_{i,k}$ is a matching on S_i, although not necessarily a complete matching. It follows that for every $k = 1, \ldots, 2^j - 1$, $\cup_i E_{i,k}$ is a matching on the unordered graph (G, E), and hence has cost no greater than $S_1(\pi^h)$. There being $2^j - 1$ such matchings, we have

$$S_j(\pi_j^{opt}) \leq (2^j - 1) S_1(\pi^h).$$

The result is obtained dividing through by $S_j(\pi^h)$. ∎

Note that since $S_j(\pi^h) \geq S_1(\pi^h)$ for all j, we get a loose "pre-computation" bound of $2^j - 1$. This bound will be sharpened given measured values of $S_1(\pi^h)$ and $S_j(\pi^h)$.

Another feature of a match/merge linearization is that the sum of edge weights between adjacent LPs is not less than half of optimal.

Lemma 2 *Let* ω^{opt} *be the maximum possible sum of edge weights between adjacent LPs in any linear ordering, and let* ω^h *be the sum of such weights under a linearization produced by the match/merge algorithm. Then* $\omega^{opt}/2 \leq \omega^h$.

maximizes $S_j(\pi)$ for all $j = 1, 2, \ldots, \lceil \log |G| \rceil - 1$ reflects in part our desire that the more closely LPs communicate, the closer they are in the ordering.

We have developed a greedy approach called the *match/merge* algorithm, based on a maximal weight matching algorithm [35]. Given an undirected graph (G, E) with n vertices and m edges with nonnegative edge weights, a matching is a subset $M \subseteq E$ constrained so that no two edges in M share a vertex. A *maximal weight matching* maximizes the sum of edge weights possible in a matching. Sophisticated algorithms determine a maximal matching in $O(nm \log m)$ time [35]. Applied to our problem, a maximal weight matching identifies a good way of *simultaneously* pairing together LPs with high communication costs.

Our algorithm will ensure that LPs paired under the matching will be adjacent in the ordering, consequently any ordering π it discovers will always maximize $S_1(\pi)$. A single application of a matching algorithm will not linearize the LPs; we apply the algorithm repeatedly on increasingly smaller aggregated graphs. The graph size is reduced by merging paired LPs, into *super-LPs* (and any LP not paired in the matching is also considered to be a super-LP). Edges between super-LPs are defined naturally: an edge between super-LPs A and B exists if A contains an LP a and B contains an LP b such that a and b share an edge in the original LP graph. The weight of an edge in the super-LP graph is the sum of all edge weights of edges it represents, i.e., the total communication volume between LPs in A and LPs in B. Following this reduction, we apply the same matching algorithm to the super-LP graph. If super-LPs A and B are paired, then in our ordering the LPs represented by A and B will be adjacent in the sense that no LP from a super-LP other than A and B can lie between any two LPs in the concatenation (A, B). Following a pairing of super-LPs we can reduce the graph as before, and continue the process until the entire graph has been reduced to a single super-LP. The object at each step j of the algorithm then is to maximize $S_j(\pi)$, given the existing grouping into sets of size 2^{j-1}. The algorithm is described below.

1. Initialize n = number of LPs, index LPs from 0 to $n-1$. Initialize edge weights w_{AB} for all LPs A, B. ($w_{AB} = 0$ if A and B do not communicate; the graph is therefore considered to be complete).

2. Find maximal weight matching M.

3. Order the super-LPs: for every $\{A, B\} \in M$, if $\text{index}(A) < \text{index}(B)$ then $C = (A, B)$ else $C = (B, A)$; $\text{index}(C) = \min\{\text{index}(A), \text{index}(B)\}$.

4. Renumber the merged super-LPs to maintain their represent ordering, but to range over $[0, \lfloor \frac{n}{2} \rfloor - 1]$.

5. If some LP Z is not matched under M, then $\text{index}(Z) = \lfloor \frac{n}{2} \rfloor$.

6. For all (A, B), (C, D) matched above,

$$w_{(AB)(CD)} = w_{AC} + w_{AD} + w_{BC} + w_{BD}.$$

7. If n is odd then $n = \lfloor \frac{n}{2} \rfloor + 1$, else $n = \lfloor \frac{n}{2} \rfloor$. If $n > 1$ goto step 2.

4.1 Static Mapping

Suppose that the TPN has been partitioned into a set of n LPs, and consider the weighted LP graph naturally induced. We need to assign an execution weight e_i to each LP i and a communication cost w_{ij} between every pair of LPs i and j. e_i reflects the amount of computation time per simulation time unit expended on LP i, and w_{ij} reflects the communication time per simulation time unit expended on communication between LPs i and j. Before running the simulation these values are unknown. We initially guess then by setting e_i equal to the number of places and transitions in LP_i, and w_{ij} to the the number of arcs crossing between LPs i and j. The cost of parallel processing is best captured if we seek to minimize the load on the most heavily loaded processor, where the cost of a processor's load is defined to be the sum of the execution weights of its LPs, plus the sum of the costs of its LP's edges that are "cut" by the mapping (i.e., edges whose LPs lie on different processors). Formally, if $R_k(m)$ is the set of LPs assigned to processor k under mapping m, then the bottleneck cost of m is

$$\max_{1 \le k \le P} \{ \sum_{j \in R_k(m)} e_j + \sum_{j \in R_k(m), i \neq R_k(m)} w_{ij} \}.$$

Unlike many other objective functions in the mapping literature, this one explicitly considers parallelism in both computation and communication. Fast algorithms for finding optimal mappings are known when the LPs are arranged in a linear order, and the mapping satisfies the contiguity constraint [3, 11, 27, 6]. This means that the workload assigned to a processor must be a contiguous subchain of LPs in the linear order. If we are to use these techniques we must rationally order the LPs, attempting to force the highest rates of communication to be between co-resident or nearby LPs. Since the mapping algorithms themselves are not new, we focus on the problem of linearization. Following this we prove that the algorithm finds optimal mappings on balanced ring and hypercube graphs, and we explain why many different linearizations enable the chain mapping algorithm to find the optimal solution.

4.1.1 Linearization

We seek an ordering that concentrates highly communicating LPs close together *everywhere* throughout the ordering. A global view rather than incremental view of ordering seems appropriate. However, even the simplest quantifications of an ordering create a computationally intractable optimization problem. For example, the problem of maximizing the sum of weights between adjacent (in the ordering) LPs is a variation of the famous traveling salesman problem, which is NP-complete. We desire even stronger conditions on our linearization, in keeping with its eventual usage. The linearized workload is to be partitioned into contiguous subchains, so that communication between LPs in the same subchain is essentially free. To measure this, consider an ordering π and the partitioning of LPs under π into contiguous subchains of equal length 2^j (excepting the last one, which may be shorter). Let $S_j(\pi)$ denote the sum of weights on edges between LPs assigned to the same subchain, the sum being taken over all subchains. The larger $S_j(\pi)$ is, the less communication would occur between processors if the chain were partitioned as assumed. An ordering π that

2. Compute and distribute $w(s_i)$.

3. Move all messages in set *Buffered* with time-stamps less than $w(s_i)$ into set *Active*. Set the valid bit in each message in *Active*, and send copies of all messages in *Active* to recipient processors.

4. Synchronize, and store received messages. Call this set *Received*. If any messages for this window are tentative then checkpoint state.

5. Convert all valid messages in *Received* into events.

6. Every processor independently up to time $w(s_i)$. Generated messages to be sent off-processor are placed in *Buffered*. Validity of messages in *Active* and *Buffered* is checked on **EndFiring** events. With any inconsistency, the message's valid bit is inverted; if the message is in *Active* a cancellation message is sent and the processor is considered to have *faulted*.

7. Synchronize, and determine whether any processor faulted. If not, terminate (if appropriate) or set $s_{i+1} = w(s_i)$, $i = i + 1$, *Active*=*Received*=$\emptyset$, remove invalid messages from *Buffered*, and goto step 2 .

8. (Faulty window processing) Process cancellation messages by inverting valid bits on cancelled messages in *Received*.

9. Synchronize. Any processor receiving a cancellation message recovers its checkpointed state, and goes to step 5. All other processors go to step 7.

We have not yet implemented this algorithm. Issues to be examined are the cost of state-saving, the number of resimulations that are required on average to determine the correct behavior, and optimizations that permit a processor to realize it is insensitive to a message cancellation. If its costs turn out to be low, then synchronous relaxation is an attractive method for exploiting more parallelism than our strictly conservative method.

4 Automated Mapping

Our approach to the mapping problem has two components. First, at load-time, the LPs must be assigned to processors without knowing either the distribution or intensity of workload. This is accomplished using a static mapping algorithm, operating on topological estimates of workload. Secondly, at run-time, the program monitors the simulation activity of each LP. Based on these measurements, the program periodically decides whether to redistribute the LPs. The decision is based on projected finishing times under the present mapping versus a proposed mapping (computed using the static mapping algorithm applied to the measured workload). In this section we develop the static mapping algorithm and analyze some of its properties. Then we describe the dynamic remapping decision policy that governs when remapping occurs, and its implementation. Performance data presented in Section §5 is taken from runs managed by these techniques.

detect when a tentative message was incorrect, because the sending processor will simulate the disenablement of the transition whose firing was already reported. That "error" is easily corrected by sending an event cancellation message after the erroneous **TokenArrival** message. The window is resimulated then. If more errors are discovered in the next pass, then they are repaired and the window is resimulated again. This process continues until the window is simulated without error. Convergence is assured because the correction to the earliest fault in an iteration cannot be undone by any later iteration. State-saving is required in order to support a window's resimulation. This, and the cost of repeating a window's simulation are the main overheads of the method.

Window properties help minimize relaxation's communication overhead. In the general synchronous relaxation method a correction may end up causing the transmission of a message that has never been seen before. This has nontrivial ramifications on the types of error corrections that have to be anticipated. However, we can use synchronous relaxation so that the only issue to be resolved by iteration is the validity of tentative inter-processor **TokenArrival** messages.

To reduce message cancellation costs, we modify the protocol so that any message with time-stamp s' is buffered until the simulation reaches the window $[s, w(w))$ containing s'. The message is sent just prior to simulating $[s, w(s))$. Holding back the message this way does not affect the value of $w(s)$ computed (because the **TokenArrival** message being withheld has a corresponding **EndFiring** event with the same time-stamp in the sending processor's event list). This arrangement permits a processor to cancel any tentative message locally (i.e., without interprocessor communication) in any window prior to the one containing its arrival time. Of course, if sent, the message might still be cancelled by some event between times s and s'. The processor P that originally sent the message to Q discovers the cancellation conditions, and sends a cancellation message after it. Receiving the cancellation, Q changes the validity status of the message, recovers its state at s, and resimulates the window.

The relaxation algorithm executed by each processor is presented below. We presume that every tentative message has a "valid/invalid" bit. It is possible for an event cancellation message to be itself canceled, so we define the effects of a cancellation on a tentative message to be an inversion of its valid bit. In the description below, set *Active* contains all tentative messages sent by the processor in the present window, and *Buffered* holds all known messages generated by the processor which have not yet been sent. Cancellation message generation is handled simply. During any iteration, the processing of an **EndFiring** event for transition t checks to see if (i) an associated tentative **TokenArrival** message in *Active* or *Buffered* was considered to be valid or not in the previous iteration, and (ii) whether any tentative **TokenArrival** message in *Active* or *Buffered* cancelled by this event was considered to be valid in the previous iteration. These checks are performed on status bits of messages in *Active* and *Buffered*. The effect of a conflict between the valid bit and the simulation state is to invert the valid bit. If the errant message is in *Active*, a cancellation message is sent to the message's recipient.

The relaxation-based algorithm's description follows.

1. For every initial token, insert a **TokenArrival** event in the appropriate processor's event queue, with time-stamp 0. Assign $s_1 = 0$, $i = 1$, *Buffered*=$\emptyset$, and *Active*=$\emptyset$.

The reason for this protocol's success on large models is intuitive. Imagine the simulation time line marked wherever an event occurs. This protocol slides a window of width $\delta_{\min}$ or greater across the time-line; processors execute independently during a window. If $\delta_{\min}$ remains fixed, as the model size grows the density of events in a window increases. The protocol's overhead is independent of model size, and so is amortized over an increasing number of events. The protocol can identify many events even when there is no minimum firing time. Applying the results of [26] to TPNs, we are assured that if firing times are random and exponentially distributed, then the number of events in a window grows without bound as the model size is increased.

3.3 Tokens Committed at Firing

We are also able to handle TPNs with a different firing rule: if a transition with firing time δ is first enabled to fire at time s, and remains enabled throughout time interval $[s, s+\delta]$, then (and only then) the firing occurs at time $s+\delta$ and token counts are adjusted at the transition's input and output places. Due to the influence of decision places, we cannot commit to the effects of firing a transition until the full enablement duration has elapsed. For instance, suppose transitions t_1 and t_2 share a input decision place p. A token arrives at p and enables both of them. The first to fire consumes the token at p, and so disables the other. A serial simulator would post **EndFiring** events for both transitions at the instant they become enabled. Then, the first **EndFiring** event processing will include a re-analysis of the status of all transitions that might have been disabled; the **EndFiring** event for each such is removed from the list. Consequently, in the parallel simulation one cannot safely pre-send **TokenArrival** messages for preemptable transitions.

The easiest solution is to prohibit errant messages from being sent. We simply constrain LP formulation further so that if t is a transition with a decision input place, then all of t's output places are assigned to the same LP as t. This rule ensures that erroneous **TokenArrival** arrival messages are never sent, since nothing can interfere with the firing of a border transition once it is enabled.

The solution above may cause a model to be so over-aggregated so that opportunities for parallelism are limited. If this is the case, it is possible to deal with the situation in a number of ways; however all require increased communication. One method is to have **TokenArrival** messages be sent as before, recognize that those from transitions with decision input places are *tentative*. Tentative **TokenArrival** messages are treated as "appointments" [24]. A processor Q receiving one with time-stamp s from processor P will not simulate any event with time-stamp s or greater until it is given permission by P. If P ends up canceling that **TokenArrival** event, it immediately sends a message to Q notifying it to cancel the event. If P ends up actually simulating the associated **EndFiring** event, then it sends a message to Q indicating the **TokenArrival** event is correct. Deadlock is avoided if one ensures that there is a positive delay $\delta > 0$ between when the receipt of a **TokenArrival** message to an LP can affect the behavior of any of its border transitions.

If event cancellations are rare, then we may be able reduce costs by using optimism, integrating our protocol with the method of *synchronous relaxation* [7]. The idea is to simulate a window as before, with pre-sent **TokenArrival** messages, which we assume are correct. We can always

3.2 Parallel Algorithm

The following is a brief overview of the protocol. Suppose that all simulation events in all processors up to (but not including) time T_{sim} have been simulated. Our protocol will compute a simulation time $w(T_{sim})$, such that all events with time-stamps in the *window* $[T_{sim}, w(T_{sim}))$ can be executed without further communication between processors. The openness of the upper window edge is deliberate, for reasons to be made clear. Off-processor messages generated in the course of processing **BeginFiring** events may be sent and received directly, or may be buffered and delivered at the end of the window processing. A received message is converted into a **TokenArrival** event and is inserted into the event-list. Messages sent between co-resident LPs are converted immediately into events. If programmed properly, there is insignificant additional overhead due to on-processor inter-LP communication. Processors synchronize globally upon simulating up to time $w(T_{sim})$, and the process repeats.

The time $w(T_{sim})$ must be chosen carefully, as follows. At the global synchronization point, each processor calls **BoundNextMsgTime()**; $w(T_{sim})$ is defined to be the minimum conditional bound returned, among all processors. The min-reduction can performed in $O(\log P)$ time on most multiprocessors, where P is the number of processors. We have proven elsewhere [26] that every future off-processor message the simulation will send has a time-stamp of at least $w(T_{sim})$. Thus, all inter-processor messages that the simulation will generate in the interval $[T_{sim}, w(T_{sim}))$ have already been identified, and converted into events. Every processor may therefore simulate its submodel up to (but not including) time $w(T_{sim})$ without danger of receiving a "late" message—the exclusion of time $w(T_{sim})$ from the window is now understood, since a message at that time might be generated by an event inside the window. This independence property leads us to the protocol given below.

1. For every initial token, insert a **TokenArrival** event in the appropriate processor's event queue, with time-stamp 0.

2. Set $s_1 = 0$, set $i = 1$.

3. For every processor, call **BoundNextMsgTime()**. Use a logarithmic time min-reduction to compute and distribute $w(s_i)$—the minimum value returned by any **BoundNextMsgTime()** call.

4. Every processor simulates up to time $w(s_i)$, independently of and in parallel with all other processors. Event processing is that of Section §2, save that **TokenArrival** events destined for off-LP places are passed as time-stamped messages. Messages between co-resident LPs are converted immediately into events.

5. The processors synchronize globally, then every processor accepts any remaining unreceived messages. A processor consumes a received message simply by inserting the described event into its event list.

6. Define $s_{i+1} = w(s_i)$, then increment i. Check termination conditions, return to step 3 if the termination conditions are not satisfied.

All places and transitions in a given LP will always be executed on the same processor, even if the LP's processor assignment changes. The problem of effectively aggregating a large netlist description of a TPN into LPs is difficult (but see [5] and [33] for nascent attempts), and perhaps unnecessary. Petri nets are commonly developed using graphical tools; these tools frequently let the modeler aggregate and then duplicate some subnet, e.g., a processor or an interface logic module. These are excellent candidates for LP aggregation, and it is a simple matter for a modeler to graphically communicate such aggregation to the tool. We have done exactly this in our tool **pntool**[21], that serves as the graphical front-end for our automated TPN parallel simulation testbed.

Our solution requires that two rules be followed when aggregating (and **pntool** enforces these rules).

- All input places for a transition are assigned to the same LP as the transition.
- Every transition t with an output place that is assigned to a different LP than t must have a non-zero firing time.

The first rule vastly simplifies the logic needed to decide when a transition may fire. Alternate synchronization schemes for timed Petri nets do not make this assumption [36, 14]. The second rule guarantees a lapse of simulated time between an event e on one processor, and any subsequent event that e may cause on another. With minor modifications this requirement could be relaxed, but with increased communication.

The event processing logic on every processor is that of Section §2, save that the code must sometimes send a **TokenArrival** message rather than generate a **TokenArrival** event. Our protocol controls these inter-LP message communications. It relies on two key activities: the *pre-sending* of **TokenArrival** messages, and the computation of lower bounds on the time-stamp of the next message an LP might send to an off-processor LP. We have already introduced the notion of pre-sending **TokenArrival** messages—these messages are sent as part of **BeginFiring** event processing, rather than **EndFiring** event processing.

Given that **TokenArrival** messages are pre-sent, one can always compute a lower bound on the time-stamp of the next message a processor will generate. Let δ_{min} be the minimum firing time among all *border transitions*, those with input and output places assigned to different processors. Let T_{sim} be a processor's simulation clock after completing some event's processing. The next message sent off-processor cannot have a time-stamp smaller than $T_{sim}+\delta_{min}$, for only **BeginFiring** events send messages, and every such message's time-stamp is at least δ_{min} larger. A potentially larger *conditional* bound can be constructed with very little extra cost by replacing T_{sim} with the least time-stamp on any event in the event list, say E_{min}. We take $E_{min} = \infty$ if the list is empty. The validity of this bound is conditioned on the processor not receiving a **TokenArrival** message with a time-stamp smaller than E_{min}. We have shown in [26] that bounds conditioned on the absence of further message arrivals suffice for our protocol. Our parallel solution assumes the existence of a routine **BoundNextMsgTime()** that finds E_{min} and returns the sum $E_{min} + \delta_{min}$. We turn next to a discussion of the protocol and its integration into the simulation algorithm.

to carry more than one token when it fires. All of these have important applications, and can be incorporated directly into the framework we propose.

In the section to follow we show how to implement this algorithm on a parallel computer.

3 Synchronization

We believe that parallel simulation will be practical primarily when large simulation models are distributed over a moderate number of processors. The usual use of discrete-event simulations is to construct confidence intervals from simulation output. Confidence intervals call for independent replications, and there is scarcely any easier way to exploit parallelism than to concurrently run independent replications. However, one rarely wants to run more than, say, twenty replications of a long-running simulation, because the width of a confidence interval decreases only in proportion to the inverse square root of the number of replications. Given a 500 node multiprocessor, one is more likely to devote 25 processors to each of 20 independent replications than one to devote an independent replication to each processor. Simulation is frequently used as an exploratory tool, aiding a decision based on one long run, which we wish to execute as quickly as possible. Thus we believe that techniques for parallelism have practical interest. We also believe that parallel simulation will be useful primarily on large simulation models. Small simulation models are simulated sufficiently quickly on workstations or PCs. Parallel architectures offer increased main memory size over conventional architectures, which helps to avoid thrashing virtual memory.

For the reasons outlined above we have concentrated on parallel simulation techniques suitable for large simulation models. We have studied a conservative synchronous approach to synchronization and demonstrated analytically that it can achieve good performance when the size of the simulation model is large [26, 25]. The solution we now develop for TPN simulations is an application of this approach to the TPN problem. We extend our previous work by tailoring the approach to work around TPN features that cause difficulty for parallelized simulation, to take advantage of TPN features that ease parallelized simulation, and to develop a new TPN-tailored relaxation based synchronization algorithm.

The remainder of this section is divided into three parts. The first provides general information about synchronization. The second discusses the basic synchronization itself, while the third part extends the method to TPNs where transition firings may be preempted.

3.1 Preliminaries

We assume that the TPN model is partitioned by the modeler into logically cohesive subnets we call *Logical Processes*, or simply, LPs. LPs are mapped to processors. Every processor maintains its own simulation clock, and an event list for every LP. A min-heap maintained over the minimal elements of each LP's event list allows us to treat the processor as having a single event list. One processor communicates with another by sending a time-stamped message. In our framework that message always reports the arrival of a token to some place, and the time-stamp records the arrival time.

- From time s to time $s + \delta$ the transition is considered to be *firing*;
- At time $s + \delta$ a token is added to each of t's output places.

We say that the transition firing is *enabled* at time s, and *completes* at time $s + \delta$. Note that tokens are committed to the transition firing at the time of the transition being enabled, not at the point when the transition actually fires. The interpretation that commits tokens only upon firing is also common; we will later discuss how to handle this as well.

We may construct a discrete-event simulation of a TPN whose events are **TokenArrival**, **BeginFiring**, and **EndFiring**, which denote the arrival of a token to a place, the beginning of a transition's firing, and the ending of a transition's firing, respectively. Assuming that the initial marking is implemented through the event list with **TokenArrival** events at time 0, the simulation may be executed using the following sequence.

1. Choose the least time event, at time T_{sim} and advance the simulation clock to time T_{sim}.
2. Execute the event, in one of the following manners.

   Case: TokenArrival Let p denote the associated place. Increment the token count at p. If the previous token count was non-zero, then the event processing is finished. Otherwise, this token's arrival may enable the firing of some transition. In this case, among all of p's output transitions, identify those now enabled to fire due to the token's arrival. Choose one of these uniformly at random, say t, and insert a **BeginFiring** event for t in the event list, with time-stamp T_{sim}.

   Case: BeginFiring Let t denote the associated transition, and let δ_t denote its firing time. Decrement the token count at each of t's input places. For every one of t's output places p', insert a **TokenArrival** event with time-stamp $T_{sim} + \delta_t$ into the event list. Finally, insert an **EndFiring** event with time-stamp $T_{sim} + \delta_t$ into the event list.

   Case: EndFiring Measure and record statistics.
3. Return to step 1 if termination conditions are not met.

It may seem curious to generate new **TokenArrival** events as a result of **BeginFiring** processing, instead of **EndFiring** processing. This reflects our deliberate choice to highlight lookahead—at the time a transition begins its firing we can predict exactly when tokens generated by the firing appear in their new places. Our parallel solution exploits this. Lookahead is not necessary in purely serial simulations.

As discussed in [17], there are a number of ways one can augment TPNs. Some arcs *inhibit* rather than enable transitions, meaning that the associated place must be empty for the transition to fire. Priorities may be assigned to decision place output arcs, to give some control over transition enablement. One may use random firing times, one may associate a probability distribution with the output arcs of a transition—on firing, one randomly chosen output place receives a token. Additional modifications include the association of colors with tokens, and allowance for an arc

First, one can rarely analyze the quality of a graph-partitioning solution, except to assert some local optimum condition. Secondly, the objective function does not reflect parallel communication. Our approach has analytic assurances, and seeks to minimize an objective function that better models execution time.

The contributions of this paper are two-fold. One of our contributions is a new heuristic for static mapping, aspects of which can be analytically quantified. In some cases we can bound the deviation of the results from optimal, in other cases we can prove optimality itself. We also extend our earlier work in synchronization and in dynamic remapping decision-making to the TPN simulation problem, and synthesize these adaptations with the new mapping algorithm. We demonstrate a tool that accepts a graphically designed TPN, then automatically maps, synchronizes, and dynamically remaps the simulation executing on a large scale parallel architecture. We report the results of a number of large TPN models, including one of the Thinking Machines CM-1 global routing network, and a slotted-ring parallel architecture. We report good performance, obtained automatically, on large scale parallel architectures. In one case we observe a speedup in excess of 43 on 64 processors of the Intel Touchstone Delta.

The remainder of this paper is organized as follows. In Section §2 we discuss TPN semantics. Section §3 develops synchronization and simulation algorithms, and Section §4 discusses automated mapping algorithms. Finally, Section §5 presents our performance results. Section §6 gives our conclusions.

2 Background

A Petri-net can be viewed as a bipartite graph, with each node classified as either a *place*, or a *transition*. The usual graphical conventions depict a place by a circle, and a transition by a straight line. Places may direct arcs to transitions, and transitions may direct arcs to places. Each place that directs an arc to a transition t is known as one of t's *input places*; likewise, each place to which t directs an arc is known as one of t's *output places*. Input and output transitions are similarly defined with respect to a place. A place may hold any number of *tokens*; the tokens may move from place to place in accordance with the *transition firing* rule. A transition t may *fire* if each of its input places has at least one token each. The effect of t's firing is to remove one token from each of t's input places, and to add one token to each of its output places. The placement of tokens in places at any instant is known as a *marking*.

A *decision place* has more than one output transition. The arrival of a token there may fulfill the firing requirements of multiple transitions. However, only one of these transitions may fire, since the firing of the first such will remove the enabling token from the decision place. A standard means of resolving this dilemma is to non-deterministically choose which transition (among those able to fire) will actually fire.

An ordinary Petri-net has no notion of "time". A common variant of timed Petri-nets associates time with transition firings, as follows. Suppose the conditions to fire a transition are met at time s, and the *firing time* associated with that transition is δ. Then

- At time s, one token is removed from each of t's input places;

cipal contribution of this paper is to demonstrate that effective automated parallelization of TPN simulations is possible using a very conservative, very simple, synchronization scheme. The lookahead calculation is easy and automatic, and we provide a new automated mapping algorithm with a demonstrated ability to balance workload and keep communication overhead low. We also incorporate dynamic remapping logic, and observe how it substantially boosts performance. The speedups we present are an order of magnitude larger than any previously reported on TPN simulations.

The conceptual model of parallel simulations that is usually studied (based on the seminal work in [4]) precludes Petri-net semantics, an observation detailed in [36]. This conceptual model ascribes fixed communication channels between *logical processes*(LPs); time-stamped messages are exchanged via these channels, and an LP's simulation clock is advanced as a result of consuming a message. The solution described in [36] involves extension of this model to support Petri-net semantics. SIMD simulation using recurrence relations of a constrained class of stochastic TPNs is developed in [2]. In work more closely related to ours, Sellami and Yalamanchili [32] and [33] consider a conservative protocol to simulate "marked graphs", which are derived from a restricted class of TPNs. They too exploit model characteristics to optimize the synchronization protocol and to partition the marked graph model. The conceptual model we have most recently used [24, 26, 25] is simply that of communicating discrete-event simulations. Our model employs the same semantics of event list manipulation as does traditional serial discrete-event simulation, and so does not suffer from the limitations of the message-consuming model. However the specifics of our synchronization protocol require that some care be taken when partitioning a TPN among processors. In extreme cases these requirements may preclude any parallelization by our methods. We believe these cases are unusual, especially in TPN models of parallel architectures. One simple condition that ensures our protocol will work is if every communication between distinct "modules" (e.g. PEs, memories) in a simulated architecture is modeled with a transition having a non-zero firing time. This simply models the real world constraint that communication takes time. Finally, conservative and optimistic methods for timed Petri-net simulation are described in [5]. This paper also proposes rules for partitioning networks, based on topological properties. No large scale networks were considered, and performance results were limited to very small numbers of processors.

A simulation's workload cannot generally be predicted, even if the underlying structure is static. A automated parallel simulation must measure workload at run-time, and dynamic remap it when needed. Early work on the problem was developed in [23], which proposes to measure multiple trial runs, analyze critical path information from each, and cluster pieces of the simulation model based on aggregated critical path information. Later work on the Time Warp Operating System (TWOS) [13] employed multiprocessor scheduling heuristics; similar ideas are explored in [10] and [34]. These methods centralize the computation and distribution of new load distributions. Their rebalancing algorithms typically consider incremental movement of LPs in efforts to reduce the total communication cost.

Another approach is based on heuristic graph partitioning, e.g., [18], [30] and their references. Here one aggregates nodes (elemental pieces of the model) into equal-sized blocks, minimizing the sum of edge (e.g., communication) costs between separated nodes. This problem formulation has a large literature in the VSLI design community. We have chosen a different approach for two reasons.

1 Introduction

Timed Petri-nets (TPNs) are an important modeling tool used to study the behavior of various types of complex systems. While a great deal of study has gone into the analytic properties of TPNs (e.g., see [17] and its references), in practical settings TPNs are generally simulated. For example, simulation of TPN-related models is the basis for the performance analysis in ADAS[8], a tool designed specifically for parallel hardware and software performance evaluation. Discrete-event simulation of TPNs is thus an important modeling and analysis activity, known to require great computational effort. Parallel execution may decrease TPN simulation execution times; however, parallelized simulations will be adopted in general only if the parallelization is largely automatic, the topic of this paper. We describe methods for synchronizing processors and for load-balancing parallel TPN simulations. Our methods are implemented in a tool where one designs the TPN graphically, after which all parallelization is handled automatically. Our execution platform is the Intel family of multicomputers. We observe good performance (e.g., speedups greater than 40 using 64 processors) on large TPN models of parallel architectures, including nearest neighbor meshes, slotted rings, and Thinking Machines CM-1 global routing network.

Parallelized discrete-event simulation has been actively studied over the last ten years; the survey in [9] is an excellent introduction to the topic; a newer survey [22] highlights current areas of research interest. Synchronization remains a subject of high interest, owing to the complexity of the synchronization requirements imposed by discrete-event simulations. The difficulty arises because the simulation model is partitioned among processors, each of which maintains its own simulation clock. An event executed on one processor may affect a submodel assigned to another processor, necessitating an interprocessor communication. We therefore view a parallel discrete-event simulation as a collection of communicating (and synchronizing) discrete-event simulations of submodels. Consider: an event occurs at simulation time s on processor i, affecting the submodel on another processor j at time $s + d$. For instance, the event may model switch communication in a network simulation. If the processor j has already simulated past time $s + d$ it may have done so incorrectly, by neglecting to consider the effect of the message arrival at time $s + d$. Synchronization protocols deal with this problem. Two fundamentally different styles of protocols have been studied. *Conservative* approaches (e.g. [4, 16, 24]) ensure that a processor does not advance its simulation clock until it is certain that it will not bypass some simulation time at which another processor affects it. Conservative protocols are known to require *lookahead* in order to avoid deadlock, and to achieve good performance. Lookahead is the ability of a processor to predict its future behavior, as regards when next (in simulation time) it may affect another processor's submodel. *Optimistic* approaches ([12]) permit a processor to simulate ahead under the anticipation that another processor will not affect its submodel in the "past", but then correct these temporal errors as they occur. Optimistic approaches require state-saving and rollback to function properly. The notions of conservatism and optimism are not mutually exclusive; as observed in [29], the space of synchronization protocols is better partitioned using finer distinctions. This leads to protocols that combine elements of optimism and conservatism.

The synchronization approach we develop in this paper is conservative in all respects. A prin-

Automated Parallelization of Timed Petri-Net Simulations*

David M. Nicol†
Weizhen Mao ‡
Department of Computer Science
College of William and Mary
Williamsburg, VA 23187-8795

Abstract

Timed Petri-nets are used to model numerous types of large complex systems, especially computer architectures and communication networks. While formal analysis of such models is sometimes possible, discrete-event simulation remains the most general technique available for assessing the model's behavior. However, simulation's computational requirements can be massive, especially on the large complex models that defeat analytic methods. One way of meeting these requirements is by executing the simulation on a parallel machine. This paper describes simple techniques for the automated parallelization of timed Petri-net simulations. We address both the issue of processor synchronization, as well as the automated mapping, static and dynamic, of the Petri-net to the parallel architecture. As part of this effort we describe a new mapping algorithm, one that also applies to more general parallel computations. We establish analytic properties of the solution produced by the algorithm, including optimality on some regular topologies. The viability of our integrated approach is demonstrated empirically on the Intel iPSC/860 and Delta architectures using many processors. Excellent performance is observed on models of parallel architectures.

---

*A preliminary version of this paper appears in the Proceedings of the 1991 Winter Simulation Conference under the title "Parallel Simulation of Timed Petri Nets".

†This work was supported by the National Aeronautics and Space Administration under NAS1-19480 and NAS1-18605 while the author was in residence at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA 23681. Work was also supported in part by NASA Grant NAG-1-060, the Army Avionics Research and Development Activity through NASA grant NAG-1-787, NASA Grant NAG-1-1132, and NSF Grants ASC-8819373, and CCR-9201195.

‡This work was supported in part by NSF Grant CCR-9210372.