

Dartmouth College Department of Computer Science Technical
Report PCS-TR97-304
Automated Parallelization of Discrete State-space Generation*

David M. Nicol Gianfranco Ciardo
Dartmouth College College of William and Mary
Hanover, NH 03755 Williamsburg, VA 23185

January 22, 1997

Abstract

We consider the problem of generating a large state-space in a distributed fashion. Unlike previously proposed solutions that partition the set of reachable states according to a hashing function provided by the user, we explore heuristic methods that completely automate the process. The first step is an initial random walk through the state space to initialize a search tree, duplicated in each processor. Then, the reachability graph is built in a distributed way, using the search tree to assign each newly found state to classes assigned to the available processors. Furthermore, we explore two remapping criteria that attempt to balance memory usage or future workload, respectively. We show how the cost of computing the global snapshot required for remapping will scale up for system sizes in the foreseeable future. An extensive set of results is presented to support our conclusions that remapping is extremely beneficial.

1 Introduction

Discrete systems are frequently analyzed by generating and examining their underlying state-space. While a valuable technique, its most serious weakness is that large complex systems may require the generation and analysis of tremendously large state-spaces. There is relatively little locality of reference in the generation process, which means that ordinary virtual memory systems thrash once the generated state-space exceeds the available main memory. We are interested in extending the size of state-spaces amenable to generation by exploiting the aggregate memory available in a distributed-memory computer system. In earlier work we showed how this could be accomplished

*This work was supported in part by NSF grants CCR-9201195 and NCR-9527163, in part by a subcontract on the grant 96-SC-NSF-1011 to the Center for Advanced Computing and Communication, and in part by NASA Contract S1-19480 to the Institute for Computer Applications in Science and Engineering.

using a user-defined hash-function that statically partitions the graph among processors[2]. Given a generated state (typically a vector of integers), the hash-function identified the processor to which the state was statically assigned. While we showed that a well-tuned hash-function can effectively balance the workload partition and achieve reasonable execution time efficiencies as well, the method suffers from some obvious drawbacks. The user must have a thorough understanding of how the state-space will develop in order to craft a hash-function that both balances the graph partition, and keeps the number of graph edges that cross processor boundaries low. If ever in the course of generation just one processor is so burdened with states that it exhausts its available memory, the whole computation fails. If in the course of generating the graph it transpires that the generation process is not exploiting much parallelism, nothing can be done. Finally, the requirement of a user-defined function for every different modeled system makes development of a general distributed graph generation tool problematic.

In this paper we develop techniques that automate parallelization of state-space generation. Our methods include innovations in the mapping of states to processors, and policies for remapping workload. Our methods have been embedded in a general distributed state-space generation tool which itself has been integrated as the back-end of a stochastic Petri net modeling tool, **spnp**[4]. We examine the utility and costs of dynamic remapping for the tool executing on an IBM SP-2, considering effectiveness in both memory balancing and run-time parallelism. We find that our automated methods are much better than optimally hand-tuned static methods with respect to both memory balance and speedup.

The remainder of this paper is organized as follows. Section 2 provides background information on the mechanics of state-space generation and its distributed implementation, and sketches related work. Section 3 describes our method for automatically assigning states to processors. Section 4 then focuses on remapping. Section 5 describes some essential details regarding message-passing in our implementation. Section 6 reports on experiments we conducted, Section 7 develops some analytic results related to this work, and Section 8 gives our conclusions.

2 Background

Many discrete systems are analyzed formally by considering their full underlying state-space. Typically a “state” of the system is a vector of integers. For instance, in a queueing network the state is a vector of the queue lengths, one component per queue; in a Petri net the state is a vector of token counts, one component per place in the Petri net; in a system of communicating finite state machines the vector contains components describing the internal states of those machines. From a state, it is usually possible to transition other states. A transition may be caused by an external arrival to the system, or by the internal stochastic evolution rules of the model. In both queueing networks and Petri nets, for instance, the system remains in a state s for a random period of time and then chooses some new state s' that is reachable from s ; a job moving from one server

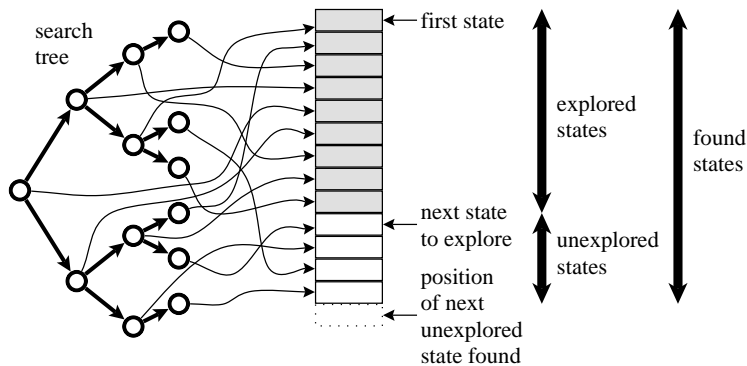


Figure 1: Organization of explored, unexplored, and found states.

to another changes the vector of queue lengths describing a queueing network, a transition firing changes the marking of a Petri net. In the following, by “state-space” we mean the directed graph whose nodes are the reachable states and whose edges describe transitions between states, the edge being directed towards the state resulting from the transition represented by the edge.

The state space for a system may be infinite—a single queue with infinite buffering can conceivably build up beyond any pre-specified queue length given a rapid enough set of arrivals. We concern ourselves here with the problem of generating finite state-spaces, although the methods will work equally well for infinite spaces if one includes a rule for terminating their expansion.

State spaces are typically built using breadth-first generation. Initially, some starting state is placed on the list of “unexplored states”. The starting state is also used to initialize a tree that will organize the “found (up to that point) states” for an efficient search. The generation process then enters a loop that terminates only when the list of unexplored states is empty. At each loop iteration, the first unexplored state is removed and added to the list of “explored states”, and (using rules specific to the problem domain) all states reachable from it in one transition (i.e., children) are generated. Each of them is then searched in the tree; if a child is found to have been generated already, then only a new graph edge is introduced, rooted in the child’s parent and directed towards the existing node. If a child is not found in the tree, then it is first inserted into the tree, before adding a state-space edge directed to it from its parent, and the child is placed at the end of the list of unexplored states. Figure 1 illustrates the relationship between explored states, the next state to explore, and the next location to place a newly generated state. The search tree is extended and balanced through manipulation of pointers, the states themselves do not move within the illustrated layout. We stress that the order in which states are found is in no way related to the lexicographic order used in the search tree.

The application-specific part of state-space generation can be separated from the common aspects through a well-defined interface, such as that we describe in [2]. One benefit is the immunization of the system modeler from details concerning parallelization. The techniques we describe in this paper follow in a similar vein and have been implemented in a manner that supports integration

with any modeling front-end that correctly uses the interface.

Different analyses are conducted once the graph is generated, depending on the application. In models of communication protocols one looks to see whether any of the reachable states represents deadlock. Livelock is also possible and is detected by identifying transient states (ones that may become unreachable once left). Resource requirements can be extracted from an understanding of the relationship between state expression and resources. Parallel algorithms for performing these types of analyses are discussed in our earlier paper [2]. When the state-space represents an ergodic continuous-time Markov chain one typically solves a set of linear equations (the global flow-balance equations [22]) that associate a stationary probability with each node in the graph. Methods for solving linear equations in parallel are standard and will not be discussed here except to say that good performance can be expected if the graph clusters well and the states are evenly balanced among processors. Indeed, we anticipate that Markov analysis will be one of the principle applications for our methods; our emphasis on balancing memory utilization derives in part from our recognition that balanced memory is required to balance the workload of the linear system solution phase.

One aspect of distributed state-space generation sets it apart from other types of computations that are usually distributed—the need to determine whether a newly generated state has been generated before. For, discovery that state B is a child of state A establishes a unique and previously unknown link between A and B , but state B may also be a child of state C , and so may already be represented somewhere in the system. This is one of the key differences between serial and distributed state-space generation; a newly generated state must be sent to some processor responsible for that state, to determine whether this is a new or repeated instance of the state. Maintaining a global directory of states is not practical, instead we must use the state description itself to determine its location. Our previous approach was to hash the state vector into the range of integer processor identifiers, using a user-defined hash function. When a generated state maps to a different processor, the state and edge information is sent to the correct location.

As automatic parallelization cannot depend on a user defined hash function, a different approach is needed. We see two requirements. First, the location of any state must be determined quickly, without interprocessor communication. Furthermore, we must map states so that a child state is likely to be mapped to the same processor as its parent, otherwise we will be overwhelmed by inter-processor communication. Second, we need to support dynamic remapping of workload, and descriptions of known states. The remapping problem is hampered by the fact that future execution requirements are unknown, and unknowable, because the structure of the as-yet-undiscovered portion of the state-space is not known.

Another dimension of the remapping problem is to determine *what* to remap, *how* to remap, and *when* to remap. These issues are developed in the following sections, after we discuss related work.

Early work on load-sharing had a world view of independent jobs in a distributed system.

The main interest was in job migration policies [23, 6, 13] that best utilized system resources. The algorithms were asynchronous, workload moved without global synchronization and without a global view of the system; key issues were how transfers are initiated, and what information is used to govern those initiations. More recent work has a different view of workload, but has continued in the asynchronous vein [24, 12]. A synchronous view of remapping was taken in work on decision policies that focus on when to remap [18, 19, 20]; balancing the delay cost of remapping against the anticipated performance gain is the essence of these policies. Globally synchronous remapping techniques are developed in [5, 25]. These methods iterate; at each iteration, pairs of processors balance workload between them, and ultimately some global sense of balance is achieved without any of the processors ever having had a global view of the system.

Our application and parallelized branch-and-bound computations share the problem that at a point when one remaps, future workload is unknown. Various methods for approaching parallel branch-and-bound have been studied [11, 10, 14, 21]. A critical difference between our application and branch-and-bound is that we must store the state-space we explore, that is not typically done in branch-and-bound. A branch-and-bound algorithm may generate a given subproblem (and then all of its descendents) multiple times; we do not. Unlike branch-and-bound, when we generate a state, we must determine whether that state has been generated before. We have semantic reasons for achieving some level of locality between our states, branch-and-bound is not so constrained.

Our approach is to exploit the high connectivity that modern interconnection networks provide, and compute remappings synchronously, and directly. Each processor has a picture of the global load distribution, and when implementing a remapping, a piece of workload or data goes directly from its source processor to its target processor. Direct methods have been considered in the context of large scale parallel processing systems [7, 1, 15], but these do not take a global view. While our approach is pragmatic and driven by available technology, we will argue, at the end of Section 7, that it scales up to moderately large parallel systems and so has some future value as well.

3 Automated State Mapping

To help understand the intuition behind our methods we first briefly review the mechanics of balanced trees. As our implementation uses AVL trees [9], our description applies specifically to these, as well as certain other commonly used search trees. One uses an application-specific comparison operator such that for any state vectors A and B , either $A < B$, $A > B$, or $A = B$. Given a tree and some state vector C , we determine whether C is in the tree by comparing it with the tree’s root, R . If $C = R$ we are done, otherwise we recursively search the left subtree of R if $C < R$, and the right subtree otherwise. If we find that C is not in the tree, it is inserted into the tree at the “exit point” of the last comparison. That is, if state vector B was the tree member against which C was last compared, then B becomes C ’s parent in the tree; C is B ’s left child if

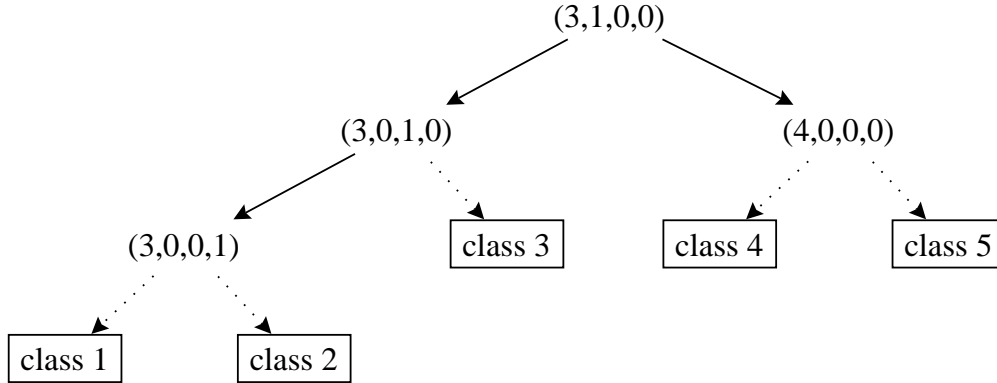


Figure 2: Balanced tree and definition of classes via exit points

$C < B$ and is its right child otherwise. The tree may balance itself following the insertion. We will say that a state in the tree has a *left exit point* if it has no left child; a *right exit point* is similarly defined.

For state-space generation with vector-valued states, it is typical to use lexicographical ordering to compare two states. If $A = (a_1, a_2, \dots, a_n)$ and $B = (b_1, b_2, \dots, b_n)$, then one determines the ordering exactly as one would if A and B were n -digit numbers, scanning from left to right looking for the first coordinate position in which A and B differ, using the comparison of the values in that position to determine the vector ordering.

We replace the hash function of our earlier work with a classifier based on balanced tree mechanics. We will map (and remap) classes, not individual states. Classes are defined with respect to a *control set* of states organized as a search tree. The same control set is built by each processor during an initialization phase, and does not change throughout the subsequent graph generation phase. The exit points of its tree are enumerated. Any state found during the breadth-first exploration is classified by searching for it in the control set's tree. If found in the control set, it is considered to be in class 0. Otherwise, the identity of its exit point defines its class, and its current processor is retrieved from a class-to-processor lookup table. Figure 2 illustrates a control set tree with four nodes and the induced five class definitions.

To keep communication costs down, it is desirable that a child and parent frequently map to the same class. At the same time, we need states to be spread among classes to give us flexibility in remapping. Our choice of control set and its organization attempts to achieve these conflicting goals. A poor choice for the control set might map nearly all states into a very small number of classes (a fact we discovered the hard way); observe that, in Figure 2, any state with a 0, 1, or 2 as its first component is mapped into class 1. Understanding the tree search as being nearly equivalent to a hashing function, we see that the control set needs to capture characteristics of the whole state-space in the same way as a hashing function does. To acquire these characteristics we create the control set from random walks.

At initialization, each processor generates a number of random walks though the state-space.

States hit by a random walk are saved. Each random walk begins at the initial state; given the last state visited by the walk, its children are generated. One of the children is selected uniformly at random to be the next state in the random walk and its siblings are discarded¹.

To further avoid any systematic over-clustering of states, we randomize the lexicographical ordering. Using synchronized random number generator seeds, each processor generates the same permutation vector $\mathbf{idx}[]$ to randomize the ordering of state components. Thus, $\mathbf{idx}[0]$ holds the index of the first state component examined in the lexicographical comparison, $\mathbf{idx}[1]$ holds the second index examined, and so on. We found this a valuable tool to protect us from the potential problems arising from correlation of states' components (consider—hashing on a vector of components whose values are highly correlated effectively diminishes the “spreading” of the hashing function. Lexicographical ordering is like hashing in this regard.) Indeed, the “natural order” in which they would be considered if we were not permuting them (e.g., the order in which the places are specified in a Petri net model), often reflects the modeler's thinking of the system, and it is not likely to be truly “random”. This is a case where a truly uninformed decision is instead desirable. Indeed, experiments where we generated orderings that encourage locality by placing components likely to change in a transition at the end of the lexicographical ordering were too successful, in that they created classes too large to move. Until our mechanism can dynamically split classes (which we discuss later) it seems that randomization is the most sensible approach.

Despite the extensive randomization, this method of class definition will induce a certain level of locality among states in a common class. For a moderately sized control class, class definition is likely to be entirely determined by comparison with indices early in the comparison sequence. There is a reasonable chance that a child will reside in the same class as its parent for the simple reason that the difference between the parent and child vectors is usually restricted to a few state components, and if classes are determined by entirely different state components, the child and parent will be in the same class. We formalize this intuition in Section 7 where we derive a lower bound on the probability that a child resides in the same class as its parent.

The processors collectively merge all saved states on all processors to define the control set, and build a tree over this set using the randomized lexicographical comparison function. Each processor then traverses the tree, evenly partitioning the member states among processors for exploration and evenly partitioning the classes among processors to build the first class-to-processor mapping table. No communication is involved: each processor has the same control set organized the same way, it executes the same algorithm as all the others to determine which states seed its exploration list, as a function of its processor identifier. The processors synchronize globally to terminate the initialization phase, and then begin the distributed generation.

A class-based mapping approach requires modifying the internal organization. A single tree per

¹We stress that if the graph represents a Markov chain, this differs from the probability that a given child is actually reached: we are now interested in strictly structural properties of the state-space, not in the stochastic and timing behavior of the graph.

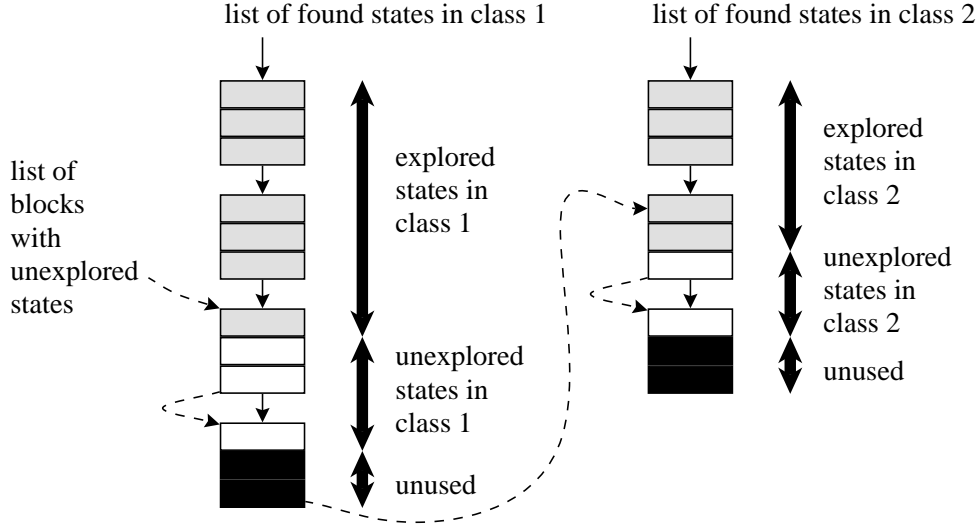


Figure 3: Block-oriented organization of class-grouped states in a given processor, assuming that it is assigned classes 1 and 2.

processor is inadequate, since it makes it inefficient to identify members of a remapped class and remove them. Instead we use an independent search tree for each class. We cannot pre-allocate memory for classes, so we partition memory into blocks of contiguous states that are allocated individually to processors.

The overall approach to distributed generation differs slightly from the hash-function based approach. First, the mechanism for identifying a state’s location is different, as we have described. Second, our internal method for uniquely identifying a state (for the purposes of edge description) is different. Earlier, the source of an edge was described by a counter-processor pair (e.g., the 12664th state on processor 23); since states remap, we now use a counter-class pair. A third difference is management of state exploration. Despite the partitioning into state-blocks, the states associated with a class can still be viewed as being ordered (logically) contiguously with a pointer to the next state to explore and a pointer to the next place to insert a new state. When new memory is assigned to a class, an entire new block is assigned, and the block is filled sequentially. All state blocks of all classes holding unexplored states are on a singly linked list; whenever a new state block is allocated it is attached to the end of this list. The state expansion loop works off the linked list, exploring each state in the head block until exhausted, after which the head block is removed, and processing is applied to the new head block.

When a processor exhausts its list of explorable states it engages in distributed termination logic. Of course, it may be that there is still ample workload elsewhere in the system, some of which will be remapped at the next remapping epoch. There are a variety of methods one might use to detect termination; as long as its cost is not intrusive, the choice matters little. We use the non-committal barrier synchronization [16] as that was easy to integrate into the distributed

generation logic.

The state-classification method may be extended to allow the control set to grow dynamically, an extension we hope to investigate in the near future. If it transpires that too many states are mapped to a class, one can “split” the class in an intuitive way. Since the class is organized as a balanced tree; the root of that tree is promoted to the control class, and the left and right subtrees of that root define new classes. The two new classes have nearly equal sizes since the tree is balanced. The promotion may cause the control set to be rebalanced, but such a balancing (at least with tree semantics) does not change any previously defined classes, and one is assured that the newly promoted state remains a leaf node in the control set.

4 Remapping

Remapping requires policies governing *when* to map, *how* to remap, and *what* to remap. The latter consideration depends on one’s objectives: a policy optimized to balance memory utilization is different from one optimized to reduce execution time, which is different from one optimized to remap only when one processor is in danger of exhausting its memory space. We will consider remapping all classes, based on known size (to balance memory utilization), and remapping only classes with unexplored states (to balance anticipated future execution costs); in the following discussion we simply speak of a processor’s “load” with the understanding that either notion applies.

A variety of remapping problems have been explored in the literature, see [26] for a survey. One important characteristic is whether the method is synchronous or asynchronous. We are driven towards a synchronous approach because of the overwhelming problem of keeping state location information up-to-date. At any time, any processor might generate any state, and need to send it anywhere. We did not wish to deal with the problems an asynchronous method would present with respect to mapping information. We are also driven towards a “direct” approach to remapping, where a source processor directly sends workload to the processor that ultimately receives it, without consideration of the communication network topology. This stands in contrast to iterative approaches where a piece of load may migrate several times in the course of a remapping. We are pragmatists; the machines we have available to us are moderately sized, and some of them have very high-bandwidth networks. Furthermore, on shared multiprocessors the subset of nodes (and the communication topology induced by the subset) upon which the program is executed varies from run to run. In this context it makes sense to directly compute remappings using globally disseminated load information, and to eschew any dependence on underlying communication topology. The mechanisms we use to accomplish the global dissemination are standard vector-valued reduction primitives that have long been found in message-passing libraries such as the Intel `nx` library, and MPI [8]. Only experimentation on very large parallel computers will definitively answer whether such methods are appropriate on such machines.

We formerly investigated two styles of remapping decision policies. Papers [19] and [20] balance

the delay cost of remapping against the performance degradation due to load imbalance and global blocking at the problem’s natural synchronization points. This policy focuses entirely on minimizing execution time, and we saw no clean way of extending it to encompass memory balance. Policies that weigh the cost of remapping against the anticipated benefit of remapping over the remaining lifetime of the computation are investigated in [18]. These policies principally account for the error that is natural in statistical measurements. On reflection we realized that the decision of whether to remap is fundamentally a choice of remapping frequency. Graph generation can be a long-lived computation with load constantly drifting out of balance. The question is not whether to remap, but how often to remap. Before inventing a policy for automating the remapping frequency we will experimentally vary it to determine how performance depends upon it. When we do balance, we attempt to balance nearly perfectly. There is compelling theoretical evidence that in a computation of this sort we should attempt to redress *any* rebalance; the model studied in [17] shows that under workload growth not dissimilar to ours, the maximum memory use by any processor is stochastically smaller the closer to perfect balance we can achieve with a remapping. However, as a means of controlling remapping overhead, we will provide mechanisms that back away from this most aggressive form of balancing, if needed.

Our first problem is to get processors coordinated so that they periodically remap. This application has no natural synchronization points, unlike many synchronous numerical methods (e.g. convergence checking). We create synchronization points with real-time clocks. We initially synchronize processors’ clocks and have each processor establish a Unix signal generator that causes a timer interrupt every Δ units of real time. The interrupt handler merely increments a counter; the graph generation code tests for changes in this counter before processing any new state (and in other critical places), and upon detecting a change engages immediately in a parallel reduction that serves both to synchronize the processors and to distribute load information. Δ is obviously a key variable; excessive overhead results from too small a value, while processor idleness may result from too large a value. Our experiments look at the effects of changing Δ .

We use a global vector OR reduction to synchronize processors. Each processor offers a vector with one component per processor, zero-filled except for the processor’s load value placed in its own component. All processors gain the complete load vector and then exercise the same logic to determine what to do: send load, receive load, or do nothing. Each processor begins by computing the average load μ . Next it rescans the load vector and defines the *on-load* group of processors whose loads are less than μ ; the *off-load* group is formed by processors with load greater than μ .

To determine how to remap each processor sorts the off-load group into decreasing load order, and sorts the on-load group into increasing load order. Thus the off-load group is arranged with the most able to contribute load at the front, while the on-load group is arranged with those most needing of states at the front. The i^{th} member of the sorted on-load group is paired with the i^{th} member of the sorted off-load group; if the on-load group is larger, we repeat the off-load group as needed. This is illustrated in Figure 4, an off-load processor may need to distribute load to more

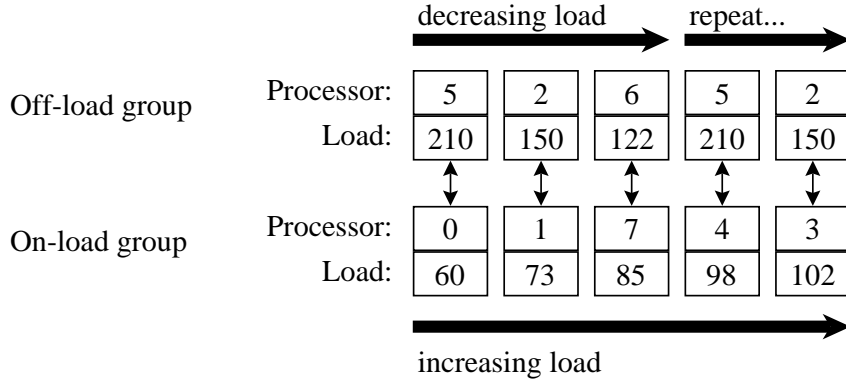


Figure 4: Pairing of source and destination processors

than one other processor.

Each off-load processor attempts to bring every one of its assigned processors up to (but not beyond) load μ without causing its own load to drop below μ or without spending too much time in transferring remapped classes. The latter constraint is enforced by computing at initialization the maximum number of states that a processor may send within a time-period equal to $0.1 \times \Delta$. This calculation requires some parametric description of message sending costs (and 10% is admittedly arbitrary), but experience with this check has proven its utility in backing away from performance crippling remapping schedules.

More specifically, an off-load processor creates a list of classes available to move, in decreasing sorted order of load. It then enters a loop within which, at each iteration, attempts to assign one class to each of its on-load assignments, scanning those on-load assignments in most-needy-first ordering. Each assignment attempt scans the list of available classes from front to back, choosing the largest as-yet-unassigned class that neither brings the target processor's load above μ , decreases the source processor's load below μ , nor increases the source processor's estimated remapping overhead beyond the limiting threshold. The loop is left once no further assignments are possible.

This method is purely heuristic. Since the classes have arbitrary sizes and are indivisible, the problem of finding any assignment that minimizes the makespan is intractable, let alone an optimal assignment that minimizes communication. We deliberately make no attempt to maximize on-processor graph edges, say, by considering inter-class connectivity. Our experience with mapping has always been that simple techniques usually provide most of the benefit possible, and we saw that measuring inter-class arcs would impose potentially large additional execution and storage costs. Very little can be said formally about this method; what *can* be said is that the method is fast, and makes every effort to redress imbalance in those processors that most sorely need more load.

Further communication is needed to prepare for the class transfers. The number of classes each off-load group processor will send to each of its targets is established with another vector OR

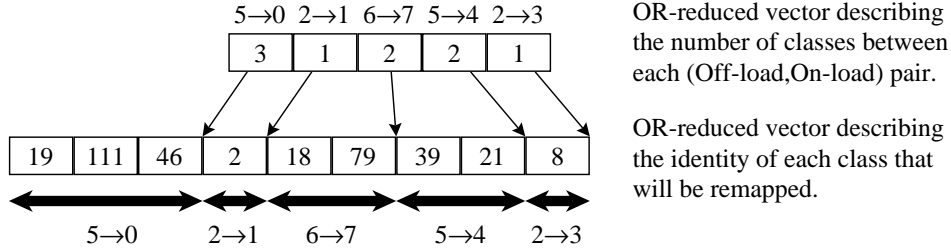


Figure 5: Global communication of remapping specifics. Example assumes the pairing of processor illustrated in Figure 4.

reduction, where off-load processors (possibly repeated as illustrated earlier) fill in their components of a zero-filled vector with these counts. All processors participate in the reduction, and take note of these counts as a means of interpreting the next communication, where identities of classes being transferred are communicated through another vector OR reduction. The length of this second vector is the total number of classes being transferred; each off-load processor writes into a zero filled vector the identities of the classes it is about to transfer. Indexing is supported by the earlier reduction that established transferred class counts, as shown in Figure 5, which assumes the pairing described in Figure 4. Following the reduction, every processor has the information needed to appropriately modify the class-to-processor mapping table, and does so. Each off-load processor then packages the the classes scheduled for transfer, sends them to their scheduled recipients, and enters a barrier. The recipients accept all scheduled classes, unpack them and create their data structures, and then enter a barrier. Any other processor simply enters the barrier. Emerging from the barrier the processors continue their state-generation activity until the next remapping check.

5 Message-Passing Details

Our application’s performance is affected significantly by how certain message-passing details are handled; because of this dependence we describe those details.

It is still the case on most message-passing systems that the startup cost of sending a message is large compared with the per-byte transfer cost. State vector lengths tend to be small (< 100 components), at least for systems one can exhaustively analyze. A processor therefore dedicates an output buffer for each other processor. As states are generated and “sent”, they are in fact just buffered until the buffer becomes full, at which point the set of states is finally sent. A processor with no further states to explore will preemptively flush its buffers. We use 3000 byte message buffers.

Our application runs under MPI, which takes care of all low-level message-passing logic. When a message arrives off the network, MPI either routes it to a memory block that an application declared for it, or stores it internally until the message is sought. Deadlock can result if messages are received faster than the application consumes them. It is necessary to implement flow control

at the application level, and to be sure that the application looks for new messages sufficiently often. Our code implements flow-control with acknowledgements. It allows a processor to send up to U unacknowledged state messages to each target processor; if it reaches this threshold, the application will not continue processing workload until an appropriate acknowledgement comes in. A state message is acknowledged as soon as it is received. The experiments we report use $U = 10$.

A counter is kept in the state exploration loop, enumerating the explored states. Every fourth state, the application calls a routine that probes MPI for any messages, and processes all messages so identified. Relatively frequent absorption of MPI-buffered messages is needed to keep senders from stalling at the flow-control threshold.

6 Experiments

These experiments are designed to reveal how well our automated state-to-processor mapping performs relative to our previous hand-tuned hash-function, and to determine how various aspects of the new method contribute to the overall performance. The experiments were conducted on the IBM SP-2, at NASA Langley Research Center. A shared multiprocessor, to run a job one submits a request for a dedicated subset of nodes with a specified size. The processors of the subset are dedicated to the application while it runs, but application communication traffic may contend with that of other applications.

For the purposes of comparison with our earlier approach, we will study a model of a Flexible Manufacturing System, illustrated in Figure 6, originally discussed in [3]. A key parameter to this model is the number of tokens k initially placed in places P1, P2, and P3; increasingly larger state-spaces are generated by increasing k . This Petri net has “timed” transitions (white boxes) and “immediate” transitions (black boxes). The state-space generator eliminates all vanishing markings, those with one or more immediate transitions enabled. When generating children caused by a timed transition firing, one or more immediate transitions may be enabled, and these will be fired (and any further immediate transitions then enabled will be fired, as so on) until all non-vanishing descendents are discovered.

The performance we report is speedup, relative to a specifically serial version. For $k = 5$, there are 152,712 states generated, with 1,111,483 edges between them in the state-space. A highly optimized serial code requires 381 seconds of processing time on one SP-2 node.

One point of interest is a comparison between the optimized serial version and the new code running on one processor. We find that the new code runs at 85% the speed of the serial code. In modifying the code to support remapping, we had to change the serial’s code techniques for minimizing *malloc* overhead. The code currently is unoptimized with respect to dynamic memory overhead, we suspect this is the principal cause of the difference in performance. Whatever the cause, an implication is that our 16 and 8 processor runs can achieve speedups of at best 13.6, and 6.8, respectively.

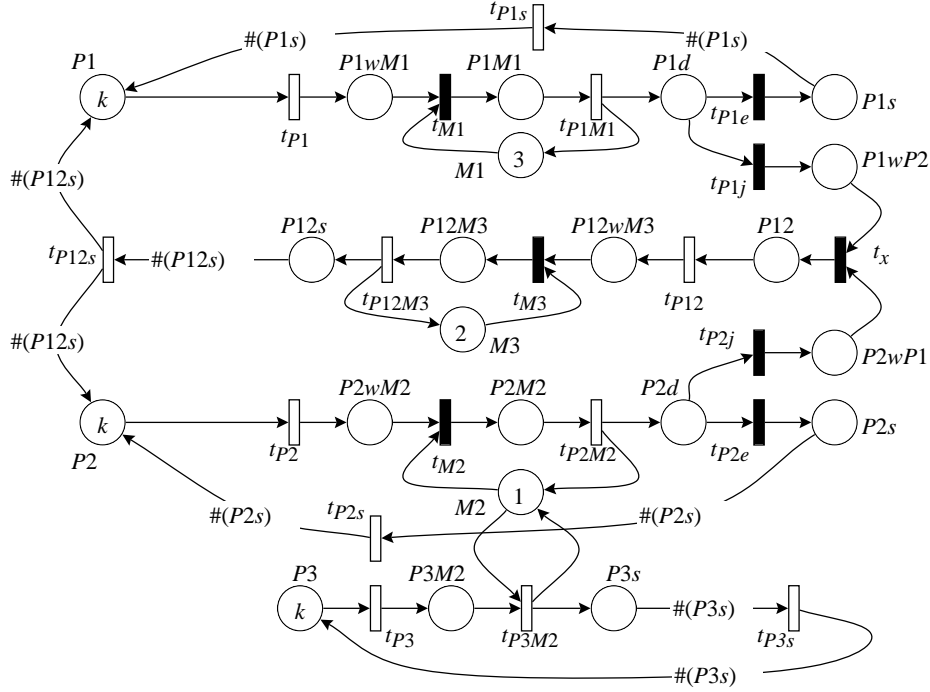


Figure 6: Petri Net Model of a Flexible Manufacturing System. White transitions are timed, black transitions are immediate. Expressions in places denote the number of tokens in the initial marking, labels on places and transitions are used for description, and labels on arcs describe the number of tokens transferred by the firing of the connected transition.

The control classes sampled ranged from 800 to 900 states, around one half of one percent of the total state-space size. Less than a second was required to sample and combine this set of states. We also measured locality among states within a class; slightly over 50% of all edges in the state space are between states in the same class. An analytic explanation for locality is given in Section 7. The code we run is non-deterministic, as its behavior depends on system snapshots measured at intervals induced by the interrupt timers, as well as randomization. We have found that variance due to randomization is surprising small (in cases where no remapping occurs), but that variance due to different system states when remapping occurs have a more pronounced affect on performance. The data we present for any set of parameters is based on five runs, reset with the same random number generator seed. Different means of reporting on those runs will be described when we look at different experiments.

We wished first to determine how well the automated state mapping works compared with the previous hash-function approach. On the best of the hash functions we previously investigated, the new method (with no remapping) is significantly faster (as much as 20%) than the old method. On another of the previously studied hash functions (not a strawman, it seemed plausible that it would be good) the new method is over five times faster on 16 processors. Thus it seems that the randomized class generation method coupled with a cyclic mapping of classes to processors is every

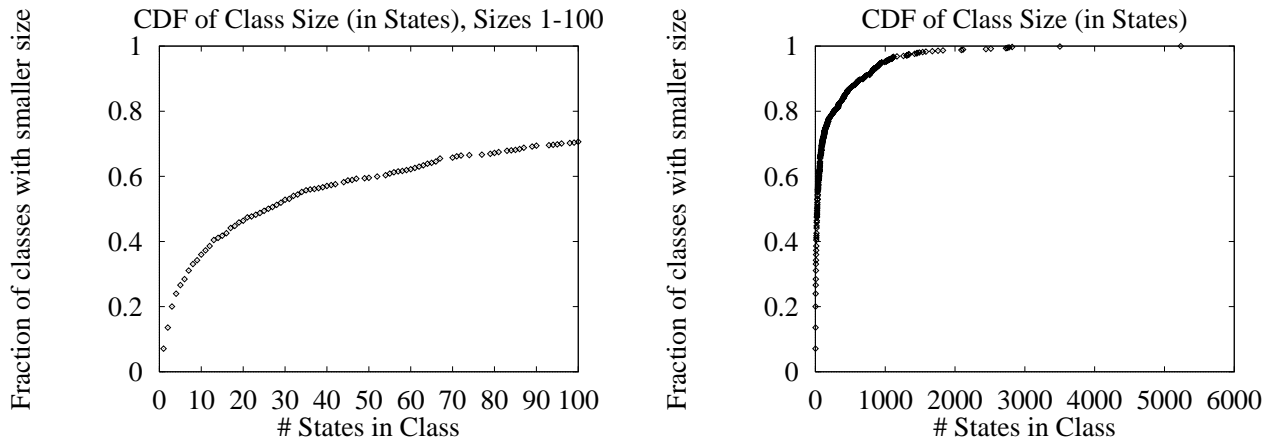


Figure 7: Cumulative distribution function of class size

bit as effective as a smart hash function. But, as the speedup achieved is only 7 (out of 16) there is clearly room for much improvement². We hope to recover that potential using dynamic remapping.

We are interested in the characteristics of classes formed by randomization. Figure 7 plots the cumulative distribution function of class size (in numbers of states). There are 789 non-empty classes overall. For clarity, we present the data in two sections, the first for class sizes 1-100, the second for all sizes. We see significant variation: 70% of classes have sizes smaller than 100 states, 10% of them have sizes greater than 1000.

The dynamic remapping experiments involve two notions of load. “Active” load measures the number of unexplored states, which represent future execution workload. “Memory” load measures the number of all extant states. We will balance active load in attempts to minimize execution time at the possible expense of memory balance. We will balance memory load in attempts to evenly balance memory utilization at the possible expense of computational balance. Figure 8 presents the speedup curves for both load definitions, as a function of remapping “period” (time between remappings), expressed as a percentage of the execution time without remapping. Hence, the rightmost endpoints represent the case with no remapping. In real time units, the remapping periods sampled were 1, 3, 5, 9, 11, 13, 21, 31, and ∞ seconds. The error bars illustrate the highest and lowest speedups measured among all 5 samples; the lines pass through the averages. The span of the error bars illustrate the variability between runs with identical parameters. First we look at performance when balancing is performed on the memory load. We see that

- performance at the highest remapping frequency is worse than less aggressive remapping frequencies. This is due to shorter periods of execution time over which the remapping overhead is amortized, and may also be due to whatever synchronization “skew” exists by synchronization on distributed real-time clocks.

²The speedup of the implementation based on a hash-function is somewhat smaller than reported in our previous work, owing to new optimizations in the serial version that significantly reduced its cost.

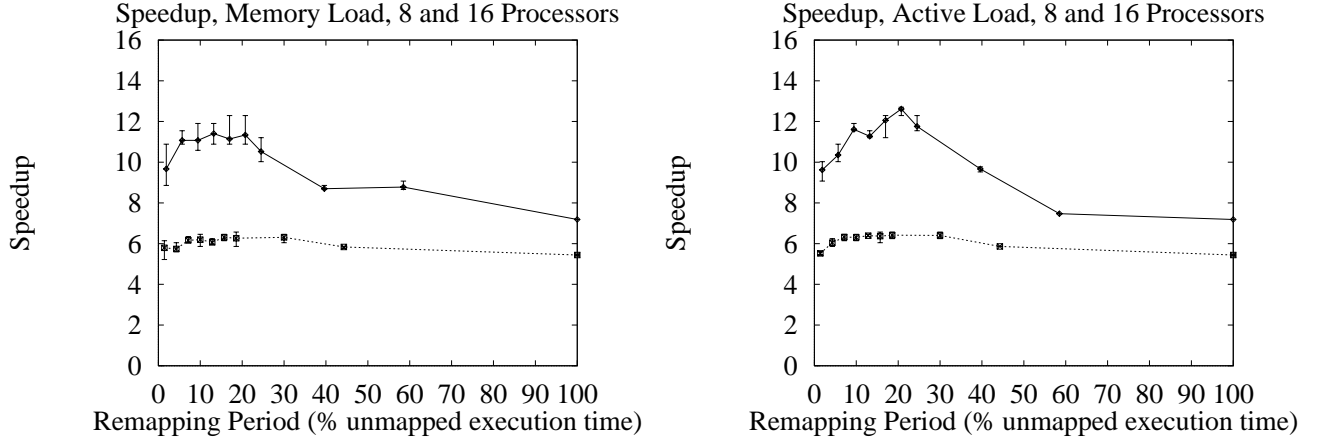


Figure 8: Speedups

- There is a significant range of frequencies over which performance is relatively constant.
- For 16 processors, performance drops off when there are fewer than 3 remappings.
- For 16 processors, there is a significant performance benefit to remapping.
- Performance for 8 processors is relatively insensitive to the remapping period, and there is small benefit to remapping.
- For both 8 and 16 processors, attainable speedup is relatively close to the maximum possible (6.8 and 13.6).

For performance when balancing active load we draw most of the same conclusions, although the range of frequencies over which performance is relatively constant is smaller. Also, interestingly, we see that there is little to distinguish between balancing memory or active load; the latter is slightly faster.

Another view of the benefit of remapping is obtained by considering how computational demands vary as the computation progresses. A first approximation to a processor’s computational workload is the number of states it explores between remappings. “Utilization” is then the ratio of average number of explored states on a processor between remappings to the maximum number explored in that period. This is admittedly inexact, as it does not account for time a processor spends searching for and inserting remotely generated states into its data structures. Figure 9 illustrates how utilization varies for the two load definitions, for the original hashing method (on the best hash function), and for our class-based method with remapping disabled, with sampling every 3 seconds. Here we clearly see that remapping keeps utilizations high, as long as there is ample workload. Utilizations tail off as the final pieces of the state-space graph are found. By contrast, utilizations in the static methods degrade gradually, as processors exhaust their unexplored state lists and must passively await receipt of an unexplored state from another processor before generating further

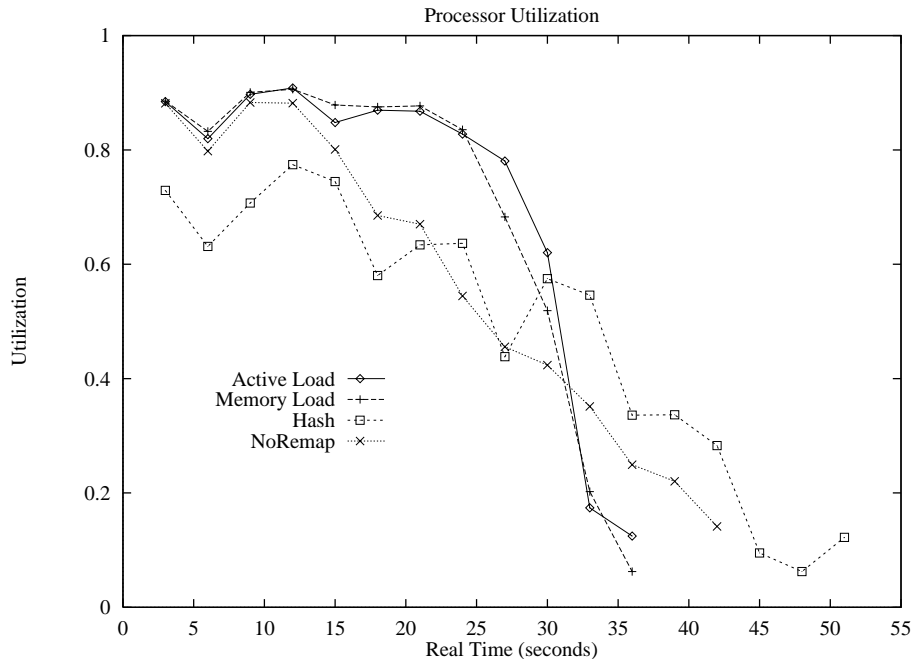


Figure 9: Processor utilizations, 16 processors, two remapping methods and the original hashing method.

states. It is interesting to note that the class-based static method tracks the dynamic methods early in the computation, but then falls away quickly.

A clearer difference between balancing active and memory load is seen when we examine variability in memory usage. Since balancing memory load gives us greater control over memory balance we expect its memory utilization to be better. Memory balance varies as the computation progresses. While the terminal memory balance is most important in some contexts, in others we are concerned with dynamics. For instance, when the problem is very large relative to the number of processors, one or more processors can reach their memory limitations and so degrade performance until remapping establishes a balance that frees them to continue processing. Even before this, excessive memory use can force a processor’s virtual memory system to thrash. The better that balance can be maintained while processing, the smaller the risk of pushing a processor against its memory limits.

Our implementation saves the number of states in a processor at every remapping timer interrupt. For each such instant we will describe the global memory state with the unitless ratio $(s_{max} - s_{min})/s_{avg}$, where s_{max} is the maximum number of states in any processor, s_{min} is the minimum, and s_{avg} is the average. For each time instant we plot the average of this statistic taken over 5 runs. Figure 10 displays this data, comparing balancing memory and active loads for remapping intervals of 1, 3, and 5 seconds. Data for both 8 and 16 processor runs is given. These plots show a clear distinction between the two notions of load. With a remapping interval of one second the two methods execute at the same rate, yet the memory balance is very noticeably different.

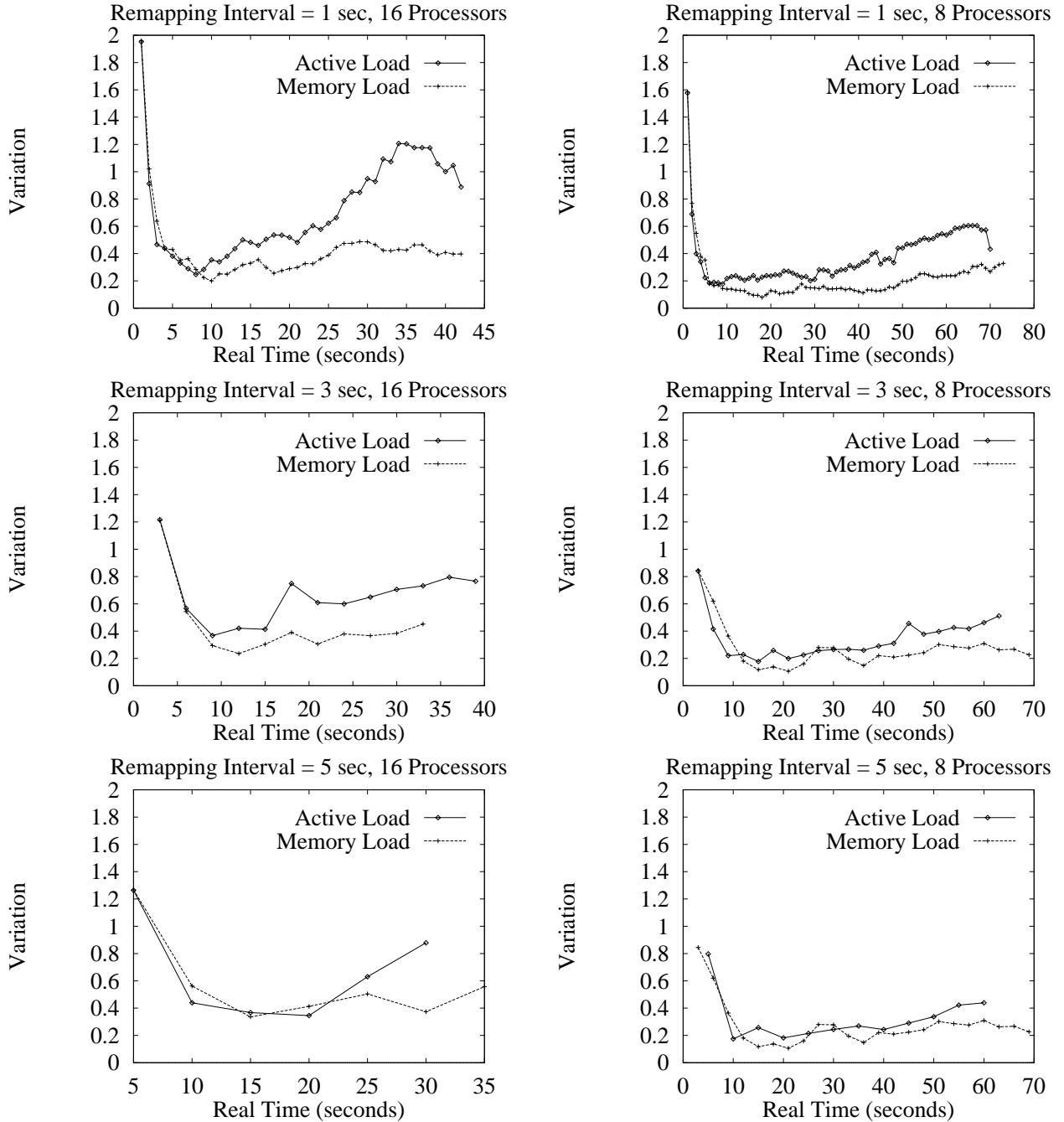


Figure 10: Variation in memory utilization, as a function of time. Variation is measured as average ratio of the difference between maximum and minimum memory usage to average memory usage.

It is notable that in all plots the variation starts high and is corrected, but then tends to grow. Balancing memory load the growth is due to the built-in constraint against spending too much time remapping. Balancing active load it is due to the fact that some imbalance simply cannot be corrected—any class without unexplored states is ineligible for remapping. The data also suggests that the methods differ less as the remapping frequency decreases.

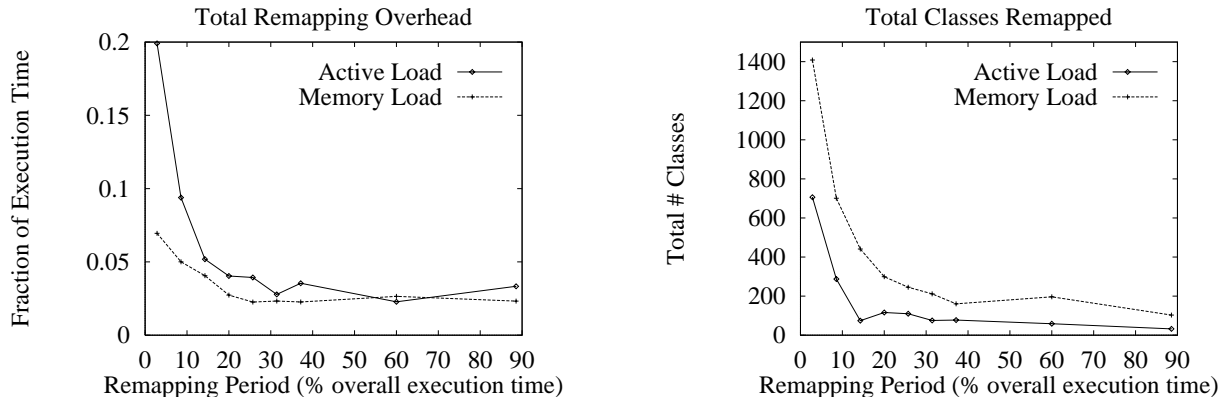


Figure 11: Fraction of execution time spent remapping, and total number of classes remapped. The remapping period is expressed as a percentage of the overall execution time.

A final view of remapping behavior is obtained by looking at statistics gathered at remapping epochs. We measure the total amount of time spent synchronizing for, computing, and conducting remapping; we also measure the total number of classes remapped (counting each move a class makes individually).

Figure 11 plots this data for the two load definitions, as a function of the remapping period, expressed as the percentage of total execution time. We see that under the highest frequency of remapping, balancing unexplored states exacts a very high overhead, 20% of the total execution time (our check against spending more than 10% time in remapping overhead is both inexact, and applied only to transfer costs). We also see very clearly that (surprisingly) balancing active load tends to have higher overhead, while balancing memory load moves far more classes! Two factors may explain these observations. First, this data says nothing about the sizes of the classes being moved (and we did not measure this). It may be that classes moved when balancing unexplored states tend to be much larger. A second factor is the cost of computing which states will be moved. The heart of the mapping algorithm searches through classes looking for the largest movable ones that satisfy certain constraints. Since the set of classes with unexplored states is smaller, it may be that much more searching is required. Both factors are mere conjectures.

7 Analysis

In this section we provide a simple analysis that explains locality between states in a common class, and that looks at the scalability of remapping in the global fashion we have used.

When a state is classified by searching for it in the control class, some random number C of its components will determine how the state compares with states in the control set. C need not be the depth of the tree; for example the first component tested may indicate that the state is less than the tree root, and the same component may indicate that it is greater than the tree's left child. So, if we choose a state s from some class at random, C describes the number of its components

that were involved in classifying the state. Now consider any child of s . The child differs from s in some random number of components D . If that set of components is entirely disjoint from the set of components used to classify the parent, then the child and parent are assured to be in the same class. Note that the reverse is not true, that is, states might differ in some (or even all) of the C components and still be in the same class, as previously observed for class 1 in Figure 2.

The set of components used to classify s and the set of components in which parent and child differ are independent, since the components used in the lexicographical ordering for comparison are randomized. If the state vector has n components, the probability p_s that the child is in the same class as the parent is at least

$$\begin{aligned} p_s &\geq E \left[\prod_{i=0}^{D-1} \left(\frac{n-C-i}{n} \right) \right] \\ &> E \left[\left(\frac{n-C}{n} \right)^D \right]. \end{aligned}$$

These expressions come from thinking of the sequence of selecting D components that distinguish parent and child, without replacement from the vector of n components, each selection needing to avoid the C components that classify the parent. Further bounding is possible by recognizing that $\frac{n-c}{n}$ is convex in c , so that by closure $\left(\frac{n-c}{n}\right)^d$ is convex in c for any $d \geq 1$. We may write

$$\begin{aligned} p_s &> E \left[\left(\frac{n-C}{n} \right)^D \right] \\ &= \sum_{d=1}^n \Pr\{D = d\} E \left[\left(\frac{n-C}{n} \right)^d \right]. \end{aligned}$$

Jensen's Inequality [22] states that if $g(x)$ is convex, then for any random variable where $E[X]$ exists, $E[g(x)] \geq g(E[X])$; thus for every d , $E \left[\left(\frac{n-C}{n} \right)^d \right] \geq \left(\frac{n-E[C]}{n} \right)^d$. Therefore

$$\begin{aligned} p_s &> \sum_{d=1}^n \Pr\{D = d\} \left(\frac{n-E[C]}{n} \right)^d \\ &= E \left[\left(\frac{n-E[C]}{n} \right)^D \right] \\ &\geq \left(\frac{n-E[C]}{n} \right)^{E[D]} \end{aligned}$$

where the last step follows from another application of Jensen's Inequality. This last expression gives us a way of considering how problem characteristics n , $E[C]$, and $E[D]$ affect locality. For the specific problem used as illustration in this paper, $n = 22$ and we instrumented the program to estimate $E[C] = 4.2$ and $E[D] = 5.1$. This yields $p_s > 0.34$, as compared with the measured $p_s \approx 0.51$. The bound evidently is not tight, but nevertheless is useful in describing the mechanics of locality and bounding locality from below.

We now turn to a simple analysis of the costs of computing a global view of load. Like many before us we will model the cost of communicating one message as a constant startup cost, plus a length-dependent transmission cost. The reductions we employ involve $\log P$ stages of message-passing, (P is the number of processors) each stage separated by computation that performs the vector-length reduction locally. The cost of computing the global reduction is modeled as $(\log P)(\alpha + (\beta + \gamma)P)$, where α is the startup cost, β the per-integer transmission cost, and γ the per-integer computation cost. We will view these constants in units of machine cycles. α includes all MPI overhead and messaging protocol logic; a reasonable order of magnitude is $\alpha = 10^4$. On modern high-bandwidth networks β is small, say, $\beta = 1$. Computation costs reflected by γ include copying into and out of memory, and comparison. $\gamma = 10$ is reasonable.

The cost of computing a global snapshot increases as a function of $P \log P$, we are interested in determining for what magnitude of P this cost is excessive. We do this by comparing the length-independent and length-dependent costs. For a given fraction f we can compute the value of P for which the ratio of length-independent and length-dependent costs is f :

$$\begin{aligned} f &= \frac{\alpha \log P}{(\log P)(\alpha + (\beta + \gamma)P)} \\ &= \frac{1}{1 + P \left(\frac{\beta + \gamma}{\alpha} \right)}. \end{aligned}$$

Solved for P we obtain

$$P = \left(\frac{1}{f} - 1 \right) \left(\frac{\alpha}{\beta + \gamma} \right). \quad (1)$$

To assess the relative cost of computing a global viewpoint, we ask ourselves what degree of dependency on length we can tolerate. This consideration should include the amortization of the remapping cost over computational periods between remaps. For instance, if an order of magnitude difference is acceptable we would use $f = 0.1$. Using the estimates of α , β , and γ given earlier, on the order of 10^4 processors could be tolerated with this approach. Considering today's technology, there is clearly room for two orders of magnitude reduction in α and *still* have computation of a global viewpoint make sense. We conclude from this analysis that remapping based on a global view should scale to machines available in the foreseeable future, and when tens of thousands of processors are routinely used we can reassess the merits of the approach.

8 Conclusions

The generation of large discrete state-spaces is a computationally intensive activity with extreme memory demands, highly irregular behavior, and poor locality of reference. As the generation of large state-spaces causes virtual memory thrashing on single processor systems, we are lead to consider exploiting the larger memory space available in parallel or distributed systems.

State-space generation is essentially a breadth-first generation and traversal of the reachability graph that is implicit in the model semantics. The critical problem is determining whether a newly

generated state has been generated before. In a serial implementation this question is answered by organizing known states in a search tree, and looking for the new state in that tree. As this is a centralized activity, any parallel or distributed solution must find an alternative approach. The method we pursue is to assign states to processors. After a state has been generated, it is sent to its assigned location, where a local search tree determines whether the state already exists. Our solution entails automated classification of states into classes, and dynamic mapping and remapping of classes to processors.

Our state classification system is based on randomization: states visited in a set of initial random walks are organized in a search tree, using a randomized lexicographical comparison function. All states that exit the search at the same point are in the same class. We provide analysis of the method that lower bounds the probability that a child is in the same class as its parent, the bound is expressed in terms of easily understood model characteristics.

Our remapping scheme uses a real-time clock to coordinate all processors periodically. When processes coordinate, they use a global vector OR reduction to distribute a snapshot of the global load state. Each processor uses the snap-shot to determine whether to send load, receive load, or do nothing. Further vector reductions establish the number and identity of the classes involved in the balancing. The processors involved in the exchange implement it, and all processors modify the global class-to-processor mapping table. We analyze the cost of a global snapshot to argue that it scales to system sizes much larger than those we anticipate would be commonly used for years to come.

We report on data collected on solution of a system model discussed earlier in the literature. We considered two variations that differ in the notion of “load”. One method balances extant classes without explicit concern for balancing future state generation workload. The other method balances states that have yet to be explored, without explicit concern for balancing existing states. We find that both methods execute significantly faster and achieve better memory balance than static methods. However, while we find that the method that balances memory does indeed achieve better memory balance, it runs almost as fast as the method that balances future workload. For both methods, remapping too often (e.g. every second) degrades performance due to excessive overhead; backing off somewhat amortizes remapping overhead but stays responsive to load variation. Our data suggests that remapping frequently and suffering greater remapping overhead is to be preferred to letting the computation get widely out of balance and then attempt to correct it. To the extent that we can generalize from our experiments, it would seem that we should remap as frequently as possible, relative to the speed of the communication medium. We expect that a longer remapping period is needed on an ethernet-connected network of workstations, for instance, to avoid undue communication overhead.

The fundamental message is that for distributed or parallel discrete state space generation, remapping is essential. We have demonstrated techniques that prove its feasibility and demonstrate its potential. Key elements to our success were automated and efficient state classification,

and low-overhead methods for computing and using global snapshots of load state to compute remappings. Access constraints to the SP-2 forced us to focus on a relatively small model instance when conducting our parametric study. In fact, pilot runs on larger models demonstrate even more significant performance gains due to remapping, in one case we observed a factor of 5 difference when using 32 processors. The point is clear, remapping is essential, and is feasible.

Acknowledgements

The code we worked with has been modified several times, by several people. The original distributed state-space generation code was authored largely by Josh Gluckman. Jeremy Gottlieb ported this code to use MPI on the IBM SP-2. Barry Lawson implemented most of the class-based data structures and book-keeping. To these tireless workers we offer our thanks.

References

- [1] G.E. Blelloch. Scans as primitive parallel operations. *IEEE Trans. on Computers*, 38(11):1526–1538, November 1989.
- [2] G. Ciardo, J. Gluckman, and D. Nicol. Distributed state-space generation of discrete-state stochastic models. *INFORMS Journal on Computing*, 1997. To appear.
- [3] G. Ciardo and K. Trivedi. A decomposition approach for stochastic reward net models. *Performance Evaluation*, 18(1):37–59, 1993.
- [4] G. Ciardo, K. Trivedi, and J. Muppala. SPNP: Stochastic Petri net package. In *Proceedings of the 3rd Int. Workshop on Petri Nets and Performance Models*, pages 142–151, Kyoto, Japan, December 1989. IEEE Press.
- [5] G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *Journal of Parallel and Distributed Computing*, 7(2):279–301, October 1989.
- [6] D.L. Eager, E.D. Lazowska, and J. Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Trans. on Software Engineering*, SE-12(5):662–675, 1986.
- [7] D. Gerogiannis and S. Orphanoudakis. Load balancing requirements in parallel implementations of image feature extraction tasks. *IEEE Trans. on Parallel and Distributed Systems*, 4(9):994–1013, 1993.
- [8] Gropp, Lusk, and Skjellum. *Using MPI*. MIT Press, Cambridge, Mass, 1994.
- [9] E. Horowitz and S. Sahni. *Fundamentals of Data Structures*. Computer Science Press, Potomac, Maryland, 1978.

- [10] L. Kale. Comparing the performance of two dynamic load distribution methods. In *Proceedings of the 1988 Int. Conference on Parallel Processing*, pages 8–12, 1988.
- [11] R. Karp and Y. Zhang. A randomized parallel branch-and-bound procedure. In *Proceedings of the 20th Annual ACM Symp. on Theory of Computing*, pages 290–300, 1988.
- [12] V. Kumar, A. Grama, and N. Vempaty. Scalable load balancing techniques for parallel computers. *Journal of Parallel and Distributed Computing*, 22:60–79, 1994.
- [13] F. Lin and R. Keller. The gradient model load balancing method. *IEEE Trans. on Software Engineering*, SE-13(1), January 1987.
- [14] R. Lüling and B. Monien. Load balancing for distributed branch-and-bound. In *Proceedings of 6th Int. Parallel Processing Symposium*, pages 543–548, 1992.
- [15] D. Nicol. Communication efficient global load balancing. In *Proceedings of the 1992 Scalable High Performance Computing Conference*. IEEE Press, April 1992.
- [16] D. Nicol. Non-committal barrier synchronization. *Parallel Computing*, 21:529–549, 1995.
- [17] D. Nicol, R. Simha, and D. Towsley. Static assignment of stochastic tasks using majorization. *IEEE Trans. on Computers*, 45(6):730–740, 1996.
- [18] 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.
- [19] D.M. Nicol and J.H. Saltz. Dynamic remapping of parallel computations with varying resource demands. *IEEE Trans. on Computers*, 37(9):1073–1087, September 1988.
- [20] D.M. Nicol, J.H. Saltz, and J. Townsend. Delay point schedules for irregular parallel computations. *Int'l Journal of Parallel Programming*, 18(1):69–90, February 1989.
- [21] N. Rao and V. Kumar. Parallel depth-first-search, part I: Implementation. *Int. Journal of Parallel Programming*, 16(6):479–499, 1987.
- [22] H.S. Ross. *Stochastic Processes*. Wiley, New York, 1983.
- [23] J. A. Stankovic. An application of bayesian decision theory to decentralized control of job scheduling. *IEEE Trans. on Computers*, C-34(2):117–130, February 1985.
- [24] M. Willebeek-LeMair and A. Reeves. Strategies for dynamic load balancing on highly parallel computers. *IEEE Trans. on Parallel and Distributed Systems*, 4(9):979–993, 1993.
- [25] C.-Z. Xu and F. Lau. Analysis of the generalized dimension exchange method for dynamic load balancing. *Journal of Parallel and Distributed Computing*, 16(4):385–393, 1992.

- [26] C.-Z. Xu and F. Lau. Iterative dynamic load balancing in multicomputers. *Journal of Operational Research Society*, 45(7):786–796, July 1994.