Optimal Dynamic Remapping of Data Parallel Computations

DAVID M. NICOL AND PAUL F. REYNOLDS, JR.

Abstract—**A large class of data parallel computations are characterized by a sequence of phases, with phase changes occurring unpredictably. Dynamic remapping of the workload to processors may be required to maintain good performance. The problem considered here arises when the utility of remapping and the future behavior of the workload is uncertain, phases exhibit stable execution requirements during a given phase, but requirements may change radically between phases. For these situations, a workload assignment generated for one phase may hinder performance during the next phase. This problem is treated formally for a probabilistic model of computation with at most two phases. We address the fundamental problem of balancing the expected remapping performance gain against the delay cost, and derive the optimal remapping decision policy. The promise of the approach is shown by application to multiprocessor implementations of an adaptive gridding fluid dynamics program, and to a battlefield simulation program.**

Index Terms—**Data parallel computations, dynamic remapping, load balancing, Markov decision process, parallel processing, performance analysis.**

I. Introduction

An IMPORTANT issue in parallel processing is the assignment, or *mapping*, of workload to processors. Workload should be mapped evenly among processors, and in a way that does not induce too much interprocessor communication. To accomplish this task, a mapping algorithm must take into account the anticipated behavior of the mapped computation. Some computations dynamically change their behavior as they progress. We are interested in computations that exhibit distinctly different *phases*, and in the problem of dynamically changing their mapping in response to phase changes.

Computations of this sort are frequently domain-oriented, time stepped, and data parallel. This means the computation is focused on a data domain often representing a physical domain, it is composed of a sequence of clearly delineated steps, and the processing for each data element within a step can be performed in parallel with the others. Many numerical simulations of physical systems can be characterized this way; we will illustrate this phenomenon with two such model problems, an adaptive gridding fluids computation, and a time-stepped land-battle simulation.

Phases arise in a number of ways. We focus here on situations where phases occur as the computation responds to either run-time input, or in response to emerging solution features in the computation. One of our model problems uses adaptive gridding [3], which dynamically adds and deletes grid points from the domain. The land-battle simulation changes phase as combating units become close enough to fight; the attrition calculations are intensive, so that the run-time behavior of the simulation dramatically changes.

Many data parallel codes are tightly coupled; this suggests the use of a global mapping (or remapping) mechanism, as opposed to a distributed mapping decision mechanism. This is especially reasonable in view of the fact that many numerical codes of the type we consider already have global components to them. In this paper, we view dynamic remapping as the dynamic invocation of a static mapping algorithm. The decision to remap must weigh the overhead costs against the performance benefits. Tradeoffs can sometimes be readily analyzed for a computation whose behavior is predictable, even in the presence of dynamic behavior [22]. However, it is often more reasonable to assume that the computation's behavior is not completely understood, and that there is uncertainty in the length of the computation, the occurrence of the phase change, and in the benefits of remapping.

Dynamic remapping raises a number of issues including detecting a phase change, choosing and implementing a new mapping, and optimally choosing when to remap. Detecting a phase change is highly problem dependent; however, a computational phase change generally occurs in response to a phase change in the physical system being simulated. It is reasonable to assume that the application code developer understands the physical system, and knows what characteristics suggest a phase change. That developer can design and implement a routine which estimates whether a phase change has occurred. The static mapping problem for data parallel computations is addressed by two very general techniques: binary dissection [2] and scatter decomposition [12]. We will introduce and use scatter decomposition in our model problems. This paper treats the remapping decision problem and has two contributions. One is to formally derive the optimal remapping decision policy, the other is to provide proof of concept, by

Manuscript received August 28, 1987; revised March 21, 1989. This work was supported in part by the National Aeronautics and Space Administration under NASA Contracts NAS1-17070 and NAS1-18017 while the first author was in residence at ICASE, Mail Stop 132C, NASA Langley Research Center, Hampton, VA 23665. It was supported in part by DoD/JPL Contract 957721, under subcontract to DFL, Huntington Beach, CA. It was also supported in part by the Virginia Center for Innovative Technology, while both authors were in residence at the University of Virginia, Department of Computer Science, Charlottesville, VA 22903.

D. M. Nicol is with the Department of Computer Science, College of William and Mary, Williamsburg, VA 23185.

P. F. Reynolds, Jr. is with the Institute for Parallel Computation, The University of Virginia, Charlottesville, VA 22901.

IEEE Log Number 8932975.

 © 1990 IEEE

demonstrating remapping's utility on two very different model problems, implemented on a multiprocessor. We will argue that this utility scales with problem size and architecture.

Most of the literature related to remapping takes a different view than ours. The work reported in [8], [11], and [30] essentially presumes that jobs arrive at a central dispatcher that assigns jobs to processors. This model of computation behavior does not describe data parallel computations well. Recent work, including [10], [18], [28], [29], and [31], allows dynamic, decentralized assignment decisions. This flavor of work generally assumes that tasks are independent; this assumption is inappropriate for most data parallel computations because of the dependence between adjacent data regions. Static task and file assignment algorithms are presented in [2], [4], [6], [7], [9], [14], [15], [17], and [24]. The present paper compliments an earlier treatment of dynamic remapping of parallel computations, where we consider computations whose behaviors change gradually [21]. The decision policy we develop here is quite different, as it focuses on detecting the abrupt phase change. Our approach is a variation on aspects of the broad treatment of change detection under uncertainty given in [25]. Our model modifies this analysis by using a different decision cost structure and by assuming a random computation duration.

This paper contains two contributions. One is to formalize the remapping problem as a Markov decision process and then determine the structure of the decision policy that minimizes the expected execution time. The second is to report on the implementation of dynamic remapping on two distinct model problems. The preliminary results obtained from these problems shows that dynamic remapping can substantially improve performance.

The remainder of the paper is organized as follows. Section II describes two model problems. Section III develops the analytic decision model, identifies important functions related to the remapping problem, and derives the optimal decision policy. Because the optimal policy cannot typically be achieved in practice, Section IV proposes a realistic heuristic. Section V reports on parallel implementations of our model problems. Performance using dynamic remapping is shown to be substantially superior to the performance of static mapping. Section VI presents our conclusions, and the Appendix treats analytic issues in detail.

II. Remapping Model Problems

The first model problem is a one-dimensional fluid dynamics code. The code numerically simulates the density $\rho(r, t)$ and velocity $\nu(r, t)$ of a compressible fluid flowing through a tube, as a function of position r and time t. Our implementation employs the ETBFCT code [5] which solves the general continuity equation

$$\frac{\partial \rho(r, t)}{\partial t} = \frac{\partial(\rho(r, t)\nu(r, t))}{\partial r} + \frac{\partial D_1(r, t)}{\partial r} + C_1 \frac{\partial D_2(r, t)}{\partial r} + D_3(r, t)$$

where C_1, D_1, D_2, and D_3 are problem dependent constants or functions. For simplicity our experiments set all of these to zero. The one-dimensional tube model is initially discretized with a coarse grid having one point every h spatial units. Given the density and velocity values of ρ throughout the domain at time t_0 and the density of fluid entering the tube at time $t_0 + \Delta t$, ETBFCT numerically integrates in t to solve for fluid behavior at time $t_0 + \Delta t$. This problem is data parallel, because the value of any two grid points at time $t_0 + \Delta t$ can be computed in parallel. The problem is easily seen to be composed of a sequence of steps—the integrations at each time step.

Variant behavior is caused by an adaptive gridding technique proposed in [3]. Every $5\Delta t$ time units the solution is examined. Additional fine grid points (at the smaller interpoint spacing of $h/6$) are added in regions where the solution is changing rapidly; likewise, existing fine grid points in stable regions are removed. Regions covered with fine grids are integrated more often, with a smaller time step.

Now consider a parallel implementation of ETBFCT on a distributed memory multiprocessor. The density at position r_0 at time $t_0 + \Delta t$ depends functionally on the density values for all grid points between $r_0 - 5h$ and $r_0 + 5h$ at time t_0. Interprocessor communication is required if any of these points reside on different processors. The coarse-grained partitioning illustrated in Fig. 1 is illustrative of a *block decomposition* which assigns each processor one length of domain, all lengths are equal. A processor must transmit and receive five points for every grid that crosses a boundary of a processor's partition. Coarse-grained partitioning minimizes communication; however, fine grids will cause severe load imbalance if most of a fine grid is assigned to one processor's domain. As shown in Fig. 1, load imbalance can be improved with a fine-grained partition, *wrapped* around all processors, at the price of much more communication. A coarse-grained partition is preferred when little or no dynamic regridding is occurring, because the load is naturally distributed. A fine-grained partition is preferred in the presence of active dynamic regridding, to distribute the extra workload introduced by fine grids. Both instances are realizations of so-called *scatter decompositions* discussed at length in [12].

In the experiments described in Section V, the computation experiences one phase where little regridding occurs, followed by another where a substantial amount occurs. It is not known in advance when a phase change will occur. This is typical of more complex fluid codes that analyze the effects of a shock wave on some body (e.g., a wing). The shock itself forms unpredictably, making the occurrence of a phase change uncertain. However, by observing the computation, one can count fine grid points and *guess* when a phase change occurs, but this method is fallible—occasionally fine grid points appear, and quickly disappear. The remapping policies we develop govern the use of a fallible phase change detection mechanism to decide when to remap from a coarse-grained to a fine-grained partition. One might also argue that we should merely use a static scatter decomposition which is less extreme than either of these. The drawback to such an approach is that it leaves unspecified exactly which scatter decomposition (among the many possible) to use—it simply pushes

Coarse-grained partitioning, block assignment to processors

Fine-grained partitioning, wrapped assignment to processors

Fig. 1. Partitioning and assignment of a fluids code domain.

the problem of balancing the competing concerns of the two phases onto the initial mapping selection. The balance to strike depends on when the phase changes and the relative performance of both phases under each of many static mappings. As these quantities are unknown, we see that simply choosing a good static mapping is a problem equally as hard as the dynamic remapping decision problem. With a two-phase dynamic remapping approach, we can be fairly confident that the block decomposition will work well during quiescent periods, while the fine-grained decomposition will work well in the presence of highly irregular gridding.

Our second model problem is Zipscreen, a time-stepped simulation originally developed by the BDM Corporation to study issues in the parallel simulation of land battles [13]. Zipscreen simulates the movement and interaction of *units* in a two-dimensional domain. Units from opposing sides engage in combat when they are close to each other—the bulk of the computational workload is in combat attrition calculations. Because the units move (possibly under the interactive direction of a simulation user) the computational workload is unpredictable and variant. Zipscreen can be mapped onto a multiprocessor using scatter decomposition in two dimensions (see [19] for details). Here, the domain through which the units move is partitioned and assigned to processors. A processor is responsible for all enemy perception, combat, and movement calculations for units resident in its region(s). Communication is required to move units from one region to another, and to share state data of all units on region boundaries. Like the fluids code, a fine-grained partition has good load balance and bad communication costs; these tendencies are reversed by a coarse-grained partition. Zipscreen is easily seen to fit our assumed model; its steps are simply the time steps in the simulation. It is data parallel because perception, attrition, and movement calculations for all units at a given time step can be performed in parallel.

The utility of remapping Zipscreen arises from the way that such simulations are typically run. In an initial phase, the opposing sides are geographically separate, and begin to move. The running time is dominated by communication and is optimized using a coarse-grained partition. The second phase begins as the opposing sides move into proximity. The running time becomes dominated by attrition calculations, and a fine-grained partition is most effective. It is not known in advance when the phase change will occur. One can measure the computation/communication ratio and guess that the phase has changed when the ratio rises. This method is fallible because our ability to separately measure computation and communication costs is not exact, and minor skirmishes can temporarily raise the ratio prior to full scale engagement. Once again we see that uncertainty in predicting and detecting the phase change complicates the remapping decision problem. Were we to simply choose a static mapping, this same uncertainty complicates the choice of that mapping. However, like the fluids problem, we can be fairly confident that a coarse-grained partition works well prior to full-scale combat, and that a fine-grained partition works well during combat. Provided that the remapping cost is not excessive, a dynamic remapping approach has much to offer.

The model problems share important characteristics. Both exhibit a phase where communication costs dominate, followed by a phase where computation costs dominate. Both can be viewed as a sequence of steps; the fluids code has regridding steps, the simulation has time steps. At some specified frequency of steps a global monitor can estimate whether a phase change has occurred. Such mechanisms are inherently fallible. For both problems a new mapping is defined simply by a new domain partitioning. Given this simple description, the processors can implement the new mapping in parallel. The serious issue is that of deciding when to remap, in the presence of uncertainty. The mathematical model developed in the following section addresses that issue.

III. A Remapping Decision Model

This section develops the remapping decision model. We first describe the decision mechanism, and then the various costs and uncertainties inherent to the model. Next we show how "remaining cost functions" can be employed to describe the expected remaining time to completion, as a function of how long the computation has already been running, what our estimated probability that the phase has already changed is, and the next remapping decision we make. The optimal decision to make at a decision step is then the one whose associated remaining cost function is least. Finally, we show that the optimal decision policy always has a threshold structure.

A. A Decision Model

Many computations, like the model problems, have clearly defined time steps. We assume that the processors synchronize at a barrier synchronization each step. It is natural to decide at the end of each step (or with some less frequent, but fixed periodicity) whether to remap. A decision step in the fluids

code occurs immediately after a gridding decision (every five time steps). Each time step in the land-battle simulation serves as a decision step. We define *cycle* i to be the period of computation between decision steps i and $i+1$. A decision to remap imposes a delay cost, after which one hopes that the time required to execute a cycle, the *cycle time*, will have improved. The problem of deciding when to suffer a short-term cost in order to achieve a long-term gain can be addressed in the context of a Markov decision process [26]. To make the analysis of such a decision process tractable, we assume that at most one phase change will occur during the course of the computation.

The remapping decision mechanism works as follows. At each decision step a monitor consults a "phase change" mechanism which tells the monitor whether, in its estimation, the phase change occurred and so whether performance will improve following a remapping. Treating this advice as statistical information, the monitor uses a Bayesian formulation to update the probability that the phase change did indeed occur before the decision step. The decision whether or not to remap depends on the value of that probability, and how many time steps the computation has already executed. A decision policy specifies a remapping decision for each decision step and phase change probability. The optimal decision policy is one which, when followed, minimizes the expected completion time of the computation.

The formal model uses random variables to describe execution times, remapping costs, and uncertainty in the phase change monitor's response. The mean cycle time depends on whether the phase change has occurred and whether the computation has been remapped. The mean cycle time for each possible combination is defined below.

Mean Cycle Time	Mapping	Phase
e_F	Original	first
e_B	Original	second
e_P	New	first
e_R	New	second

In the model problems, $e_P \geq e_F$ reflects the additional communication cost of fine-grained partitioning when load balancing is not needed; likewise, $e_B \geq e_R$ reflects the bad performance suffered by a coarse partition following a phase change. Throughout our discussions, we implicitly assume these inequalities, so that a mean per-cycle gain by remapping is possible if and only the phase has changed.

There is uncertainty in when the computation will terminate. This is captured by allowing the number of cycles N to be random and independent of phase change. We assume that N is bounded from above by some constant M. To model the uncertainty in when the phase changes, we assume that the cycle index of a phase change is geometrically distributed, with parameter ϕ. Consequently, for every n, ϕ is the probability that the phase change happens at cycle n given that it has not happened before cycle n. The geometric distribution is attractive, in that it both offers analytic tractability and some control over where probability weight is placed. If a problem tends to exhibit a phase change relatively early in a computation, ϕ can be made large; if a phase change occurs late, or not at all, ϕ can be made small. Uncertainty in the phase change detection mechanism's veracity is modeled by assuming that every invocation of the mechanism has a probability α of prematurely reporting a phase change and a probability β of failing to report an existing change. For the model problems, this uncertainty is inherent due to noisy performance measurements and the problems' dynamic behavior.

B. Calculation of Gain Probability

Using these model definitions, one can formally describe how the probability that the phase has already changed behaves as the computation progresses and as we continually consult the monitor. Under the assumption that a per-cycle performance gain can be achieved following a phase change, this probability is equivalently the *gain probability*. Faced with potential high remapping costs and poor performance in the event of premature remapping, we do not want to remap on the basis of a single positive test report. The gain probability p_n is recomputed at each decision step n. The initial value p_0 is taken to be zero. The result of a test at cycle n gives information about the likelihood that the phase has already changed. The result is combined with our prior estimate p_{n-1} using Bayes' Theorem [27]. To apply this theorem, it is necessary to first compute an *a priori* probability $p_a(p)$, which is the pretest probability that the phase has changed at the nth cycle, given that $p_{n-1} = p$:[1]

$$p_a(p) = p + (1-p)\phi.$$

If a positive report is observed, Bayes's Theorem gives

$$p_n = p^c(p) = \frac{p_a(p) \cdot (1-\beta)}{p_a(p) \cdot (1-\beta) + (1 - p_a(p)) \cdot \alpha}.$$

Given a negative report, p_n is similarly defined by $p^{\bar{c}}(p)$:

$$p_n = p^{\bar{c}}(p) = \frac{p_a(p) \cdot \beta}{p_a(p) \cdot \beta + (1 - p_a(p)) \cdot (1-\alpha)}.$$

We require one other related probability. Let $q^c(p)$ be the probability that the detection mechanism reports a phase change at cycle n, given that $p_{n-1} = p$. By conditioning on whether the change has actually changed by cycle n, it is not difficult to see that

$$q^c(p) = p_a(p)(1-\beta) + (1 - p_a(p))\alpha.$$

The probability of a negative report at cycle n given $p_{n-1} = p$ is simply $q^{\bar{c}}(p) = 1 - q^c(p)$.

C. A Stochastic Dynamic Programming Problem

It is now possible to formulate the remapping decision problem in terms of stochastic dynamic programming. At cycle n the *state* of the remapping decision process is the pair $\langle p_n, n \rangle$.

[1] If one has more detailed knowledge of the phase-change distribution, this expression is easily modified: $p_a(p, n) = p + (1-p)h(n)$, where $h(n)$ is the distribution's hazard rate function. Derivation of the optimal decision policy depends on the geometric assumption.

At every cycle, the decision incurs a cost dependent on the decision (retain, or remap) and the current state. We seek a decision policy that minimizes the expected sum of costs incurred by the decisions followed under that policy. In our application, the costs model cycle times and remapping overhead. The optimal decision policy depends on these costs, the remaining length of the computation, and our confidence that remapping gain is possible. Given gain probability p at cycle n, let $V(\langle p, n\rangle)$ denote the expected remaining execution time of the computation if we use the optimal decision policy. If we choose to retain the old mapping at cycle n, the next cycle's expected execution time is $pe_B + (1-p)e_F$. Now let $E_v(\langle p, n\rangle)$ be the optimal expected remaining execution time after cycle $n+1$, given a retain decision in state $\langle p, n\rangle$. $E_v(\langle p, n\rangle)$ describes optimal decision policy costs after cycle $n+1$, and so is stated in terms of $V(\langle \cdot, n+1\rangle)$. Given $p_n = p$ the test mechanism reports a change at $n+1$ with probability $q^c(p)$, in which case $p_{n+1} = p^c(p)$. Likewise, p_{n+1} will become $p^{\bar{c}}(p)$ with probability $q^{\bar{c}}(p)$. Thus,

$$E_v(\langle p, n\rangle) = q^c(p)V(\langle p^c(p), n+1\rangle) + q^{\bar{c}}(p)V(\langle p^{\bar{c}}(p), n+1\rangle).$$

From $\langle p, n\rangle$ the expected optimal remaining execution time following a retain decision is

$$C_t(\langle p, n\rangle) = pe_B + (1-p)e_F + E_v(\langle p, n\rangle).$$

We call C_t the *retain cost function.*

We similarly define $C_m(\langle p, n\rangle)$, the *remap cost function*, to be the optimal expected remaining execution time achieved by remapping now. Remapping incurs an overhead cost D. If the phase has changed, every remaining cycle will have mean time e_R; there are a mean number $\hat{N}_n - n + 1$ of cycles remaining, where $\hat{N}_n = E[N | N \geq n]$. Let x be the first cycle after which the phase change has occurred. Then the remap cost function at $\langle p, n\rangle$ conditioned on $X \leq n$ is

$$C_m(\langle p, n\rangle | X \leq n) = D + (\hat{N}_n - n + 1)e_R.$$

If remapping is chosen prematurely, we assume that this is determinable after suffering the cost D, that the computation uses the old mapping for cycle n, and that the decision process is free to choose remapping at a later cycle.[2] Discovering that remapping is premature sets the gain probability to zero. The remap cost function at $\langle p, n\rangle$ conditioned on $X > n$ is thus

$$C_m(\langle p, n\rangle | X > n) = D + e_F + E_v(\langle 0, n\rangle).$$

Combining these expressions gives the unconditional remap cost function

$$C_m(\langle p, n\rangle) = D + p(\hat{N}_n - n + 1)e_R + (1-p)(e_F + E_v(\langle 0, n\rangle)).$$

[2] Our results are not substantially affected if we do not assume this checking ability. The form of the remap cost function changes, but the single threshold decision policy is still derivable. The model problems are actually better suited to the modified formulation, but other problems may incorporate a checking ability.

This formulation assumes that once a new mapping is implemented the decision process stops and no further remapping is considered. Finally, we will find it convenient to define $C_t(\langle p, n\rangle) = 0$ and $C_m(\langle p, n\rangle) = D$ for $n > N$. We summarize our decision model definitions in Table I.

The optimal decision from state $\langle p, n\rangle$ is given in terms of $C_m(\langle p, n\rangle)$ and $C_t(\langle p, n\rangle)$. The principle of optimality states that

$$V(\langle p, n\rangle) = \min \begin{cases} C_m(\langle p, n\rangle) \\ C_t(\langle p, n\rangle) \end{cases},$$

and that the optimal decision from $\langle p, n\rangle$ is the one whose cost function minimizes the right-hand side of this equation. We next show that the optimal decision policy is nicely characterized.

D. Optimal Decision Policy Thresholds

We are now in a position to determine the optimal decision policy. It turns out that it is a threshold policy: for every cycle n there exists $\pi_n \in [0, 1]$ such that the optimal decision from $\langle p, n\rangle$ is to remap if and only if $p > \pi_n$. While this is a very intuitive result, its derivation is not obvious. The following lemma provides the fundamental reasons for the optimal policy structure. Its proof is given in the Appendix.

Lemma 1:

- For all n, $C_m(\langle p, n\rangle)$ is a linear function of p;
- For all n, $C_t(\langle p, n\rangle)$ is a piecewise linear concave function of p;
- There exists $n_0 \in [0, \infty]$ such that for all $n \geq n_0$, $C_t(\langle p, n\rangle) \leq C_m(\langle p, n\rangle)$ for all $p \in [0, 1]$;
- If $n < n_0$, $C_m(\langle 1, n\rangle) \leq C_t(\langle 1, n\rangle)$. □

Consider the implications of Lemma 1. $C_t(\langle p, n\rangle) < C_m(\langle p, n\rangle)$ whenever $n \geq n_0$, implying that we should retain for any value of p_n. In this case, $\pi_n = 1$. For $n < n_0$ we know that $C_m(\langle 1, n\rangle) \leq C_t(\langle 1, n\rangle)$. As a consequence of $C_m(\langle \cdot, n\rangle)$'s concavity it is geometrically impossible for $C_t(\langle \cdot, n\rangle)$'s functional curve to intersect $C_m(\langle \cdot, n\rangle)$'s functional curve more than once, as illustrated by Fig. 2. If $C_t(\langle \cdot, n\rangle)$ and $C_m(\langle \cdot, n\rangle)$ intersect at $\pi_n < 1$, then $C_t(\langle p, n\rangle)$ is less than $C_m(\langle p, n\rangle)$ for $p \in [0, \pi_n]$, and $C_m(\langle p, n\rangle)$ is less than $C_t(\langle p, n\rangle)$ for $p \in [\pi_n, 1]$. It follows that the optimal decision from state $\langle p, n\rangle$ is to retain if $p \leq \pi_n$, and to remap if $p > \pi_n$. We summarize this result:

Theorem 1: For every cycle n, there exists a π_n such that the optimal decision from $\langle p, n\rangle$ is to remap if and only if $p > \pi_n$. □

IV. A Remapping Heuristic

Given quantified model parameters we can in theory compute the thresholds identified by Theorem 1; in practice, there are significant obstacles. A computational problem arises from the fact that the optimal cost functions $V(\langle \cdot, n\rangle)$ are all piecewise linear. As we have discussed elsewhere [20], the number of linear segments describing $V(\langle p, n\rangle)$ is approximately twice that of $V(\langle p, n+1\rangle)$. An exact solution generates an exponentially large number of line segments. This problem

TABLE I
MODEL DEFINITIONS

Notation	Definition
n	Decision Step Number
N	Random number of decision points
M	Upper Bound on N
N_n	N given $N \geq n$
$\hat{N}_n$	$E[N_n]$
e_F	Pre-Gain Execution Time, Original Mapping
e_B	Post-Gain Execution Time, Original Mapping
e_P	Pre-Gain Execution Time, New Mapping
e_R	Post-Gain Execution Time, New Mapping
D	Delay to Calculate and Implement New Mapping
α	Gain Test False Alarm Error
β	Gain Test Missed Gain Error
ϕ	Phase Change Gain Rate
$p_a(p)$	A Priori Probability of Gain At Next Decision Step
$p^c(p)$	A Posteriori Probability of Gain After Positive Gain Observation
$p^{\bar{c}}(p)$	A Posteriori Probability of Gain After Negative Gain Observation
$q^c(p)$	Probability of Observing Gain Next Observation
$q^{\bar{c}}(p)$	Probability of Not Observing Gain Next Observation

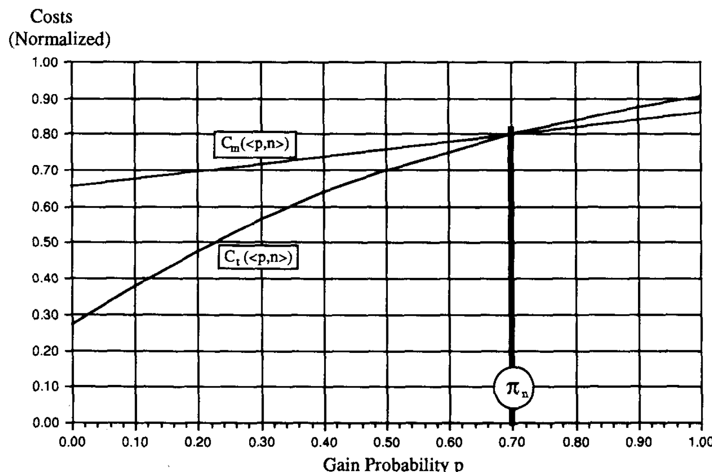

Fig. 2. Intersection of decision cost curves.

is not insurmountable; in [20], we outline an efficient approximation method with user-selected bounded error. If the model parameters are estimable, then this approximation can be used. A more difficult problem is that some of the model parameters are *not* known in advance, particularly those concerning post-phase-change behavior.

Fig. 3 illustrates the behavior of the optimal decision thresholds as a function of cycle number, for two representative sets of parameter values (and constant N). It is interesting to note that the threshold stays relatively constant until the end of the computation is near. The region where $\pi = 1$ is a region where, if we remap, the gain cannot amortize the cost because there is so little computation left. The relatively constant behavior suggests a simple decision heuristic: choose some fixed τ, and remap whenever $p_n > \tau$. Clearly, if τ is chosen to be approximately equal to π_0 (approximately the value in the initial constant region of π_n's), then this is a good policy over that region. But, of course, we do not know the value of π_0. We next examine the likely quality of this heuristic in the event that τ is not close to π_0.

Consider the behavior of p_n immediately following a phase change. The phase change mechanism correctly reports a change with probability $1 - \beta$, and incorrectly reports no change with probability β. If $\beta < 0.5$ the tendency will be for the mechanism to correctly report a change. The question is: how many cycles are required before p_n exceeds τ, thereby triggering a remapping? Fig. 4 plots the expected number of such cycles (established by simulation), as a function of τ, when $\alpha = \beta = 0.25$, and when $\alpha = \beta = 0.05$. In both cases, we take $\phi = 0.001$. Here we see a surprising degree of insensitivity to τ, provided that τ is not too large. This shows that if we can expect a reasonably large number of cycles to follow a phase change, and if we can avoid remapping prematurely, the difference between the optimal response to a change and our heuristic's response is small. On the other hand, premature remapping occurs if τ is small. Consider a situation where a phase change does not occur for at least 1000 cycles, and where α, β, and ϕ have the values given above. Fig. 5 plots the probability that p_n will exceed τ sometime in those 1000 iterations, thereby triggering a pre-

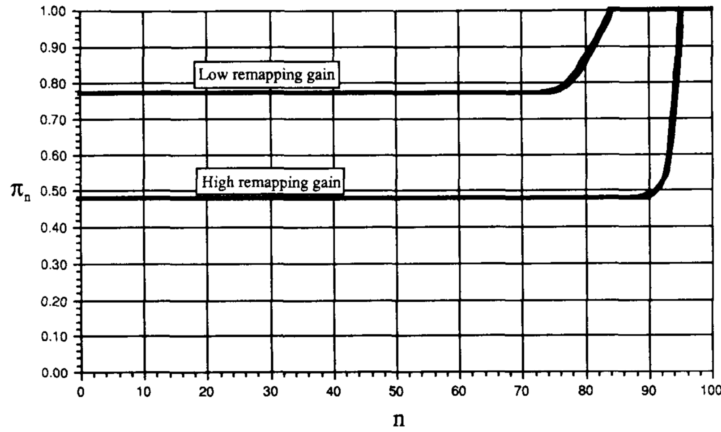

Fig. 3. Behavior of optimal thresholds.

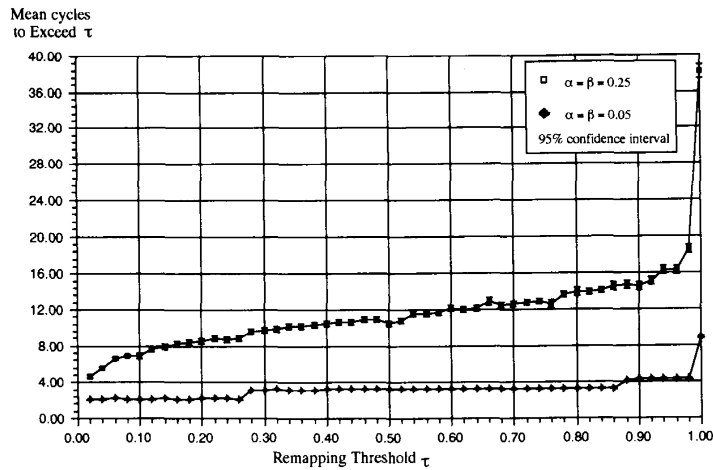

Fig. 4. Response time to phase change.

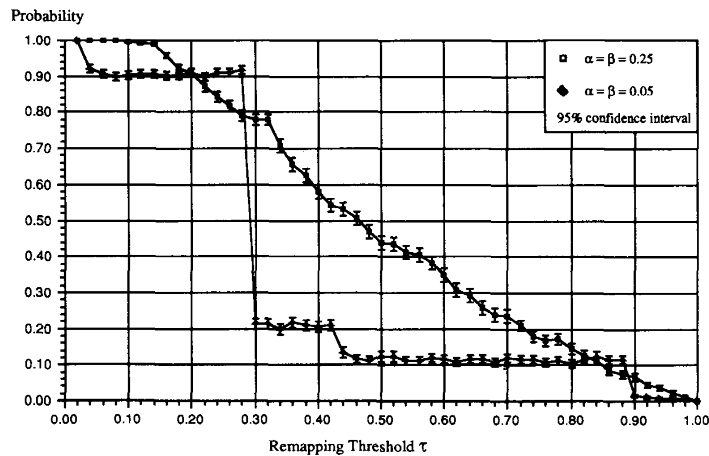

Fig. 5. Probability of remapping error in 1000 cycles.

mature remapping. When τ is low there is a high probability of a premature remapping. As τ grows, this probability diminishes significantly. For $0.7 \leq \tau \leq 0.9$ we can expect both fast response to an extant change, and reasonable protection from premature remapping. It is important to remember that there is a risk of remapping prematurely under the optimal policy as well.

The fixed threshold policy heuristic breaks down near the end of the computation; the optimal policy is to never remap, while the heuristic will. In situations where the remaining number of iterations is known, it makes sense to modify the fixed-threshold heuristic so that remapping is not chosen when the end is near. A simple mechanism for this exists if we can estimate the costs e_B, e_R, D, and the remaining number of cycles T. A lower bound on the remaining execution time given a remapping is $T \cdot e_R + D$; an upper bound on the remaining execution time without a remapping is $T \cdot e_B$. We should not remap if T is so small that this lower bound exceeds the upper, i.e., if $D > T(e_B - e_R)$.

Multiple phases are another issue of concern. Derivation of an optimal policy under these conditions seems daunting, and in any case, such a policy would not be practical. However, the data in Figs. 4 and 5 imply that if phases contain sufficiently many cycles, then the heuristic will work well. The key issues are protection from premature remapping, responsiveness to a phase change, and enough postchange cycles before the next phase so that the cost of remapping can be amortized.

We have tested the fixed threshold heuristic extensively, using simulations of the analytic model [20]. Those results show if error probabilities α and β are reasonably small, then choosing $\tau \approx 0.7$ yields nearly optimal performance. The heuristic deviates strongly from optimal performance only when its use prompts a remapping when termination is eminent. The simulations also show that the heuristic's performance is relatively insensitive to miss-estimation of ϕ, α, and β. Given our understanding of what the real issues here are, these results come as no surprise. However, the true test of the heuristic is its application to real multiprocessor problems, a topic taken up in the following section.

V. Multiprocessor Experiments

We implemented our two model problems on the Flex/32 [16] multiprocessor at the NASA Langley Research Center. The Flex has 20 processors, each having 1 Mbyte of local memory. All processors have bus access to a 2 Mbyte common memory. The fluids code was written using a message passing paradigm. Our intent was to accurately simulate the costs of a true message passing machine. We developed a message-passing layer which presents to the user an interface similar to those presented by operating systems on message-passing machines. Our code suffers the usual and substantial costs of packing and unpacking message data, of copying messages between the user space to the system space, of blocking on unreceived but anticipated messages. These are the major costs of message-passing—time on the wire is comparatively smaller, especially on second generation message-passing machines such as the Intel iPSC/2.

The land-battle simulation was written using a shared memory paradigm where all processors work out of their local memories, and use the global memory to simply exchange data. This technique is commonly used on shared memory architectures whose global memory access cost is significantly larger than that of local memory. This paradigm is especially important for large-scale parallel architectures using a hierarchical memory—the access cost increases as the number of processors increase.

We will see that dynamic remapping works well on these two different problems, using two different paradigms. The technique therefore shows promise. However, the experiments performed are not extensive, and only two problems were implemented. One should consequently view the results as preliminary.

A. Fluids Code Experiments

512 coarse grid points were used, spread across 16 processors. Each computation consisted of 400 basic time steps, leading to 80 decision steps. A computation tended to require a few minutes of wall clock running time (the best speedup for this problem size was only about 7; speedup improves by increasing the size of the grid, but increasing the size of the grid increases the wall clock running time, a scarce resource). The phase change is forced by changing the boundary conditions at the point where the fluid enters the domain. Fluid is initially introduced at a constant rate. To effect a phase change, the boundary conditions are changed to reflect the introduction of a sine wave, which will develop into a shock. To vary the timing of the phase change we vary the point in the computation where the wave is introduced.

Our experiments compared the performance of the heuristic remapping policy (with $\tau = 0.7$, and no end of computation detection mechanism) to the performance of 1) a coarse-grained partitioning into 16 pieces, 2) a fine-grained partitioning into 64 pieces, and 3) an optimal policy created by solving for the $\{\pi_n\}$ using estimated problem parameters. We find that over a spectrum of scenarios the remapping policy strongly outperforms either static mapping. The heuristic and the optimal policy are seen to have nearly identical performance.

Our experiments use a coarse grid spacing of $h = 0.25$, and a basic time step of $\Delta t = 0.025$. The fluid velocity at a point is taken to be the fluid density at the point. The fluid density is initially constant at 5.0 throughout the tube. Entering wave densities are centered at 5.0, have amplitude 0.25, have period 8.0, and last for 1.6 units of time. ϕ is taken to be $1/400$, α and β are both taken to be 0.1. The value for ϕ is chosen to reflect basic nonknowledge of the wave's occurrence; the values of α and β heuristically reflect our experience that the phase detection mechanism is good, but not perfect. We measured the remapping cost and found it to be quite low; approximately the time required to perform one full integration step (which is one-fifth the cycle time).

Fig. 6 plots the relative performance of the static mappings compared to the remapping heuristic. The horizontal axis plots the fraction of the computation which elapses before the wave is introduced, thereby causing a phase change. The vertical

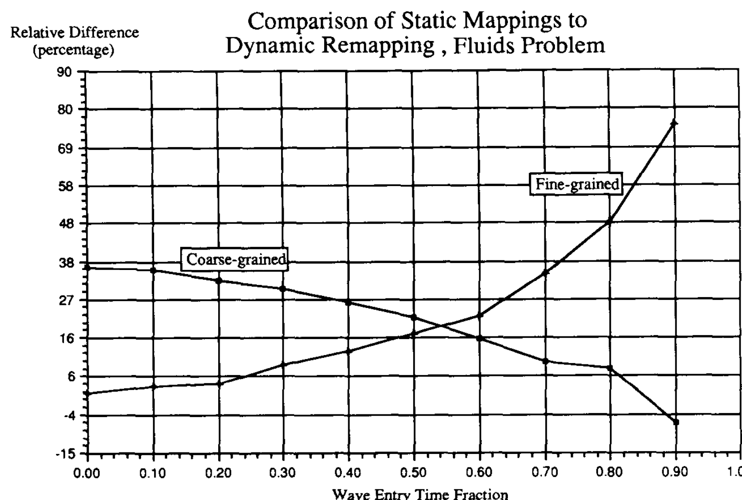

Fig. 6. Relative difference (in percentage) between two static mappings and dynamic remapping of fluids code. Vertical axis is $100 \times (T_s - T_d)/T_d$ where T_s is static mapping execution time and T_d is dynamic remapping execution time. Horizontal axis is the fraction of the computation that elapses before the phase change.

axis gives $(T_s - T_d)/T_d$, where T_s is the running time under a static mapping, and T_d is the running time under dynamic remapping. For example, a plotted value of 0.5 for some statically mapped problem means that the static mapping's running time is 50 percent worse than that of dynamic remapping for that same problem.

The time of the phase change strongly affects static mapping performance. The fine-grained partition suffers from communication overhead so long as the wave has not yet entered. Once the wave enters, the coarse-grained partition suffers from load imbalance. Especially poor performance occurs using a fine-grained partition throughout a wave-free computation, or using a coarse-grained partition when the wave enters almost immediately.

The performance of the optimal policy is virtually indistinguishable from the heuristic's, and so was not plotted. Except for the rightmost sample point, the optimal policy was less than 0.1 percent faster than the heuristic (and never slower). The heuristic's near optimality supports the same observation made with the simulation study reported in [20].

B. Land-Battle Simulation

We simulate scenarios on a 128×128 hex board, with 64 units to a side. The simulations are implemented using eight processors. The common memory dedicates space for the full state of each unit, and space for interprocessor control bits. A processor communicates part or all of a unit's state to other processors by modifying the common memory representation, and setting appropriate control bits.

The initial vertical coordinate of a unit is chosen randomly from a discretized normal distribution with mean 64 and standard deviation 16. The time of the phase change is governed by the initial horizontal placement of units. The control parameter is S: all units on one side lie in column $64 - S$, all units on the other side lie in column $64 + S$. The units from opposing sides will move toward each other, and the phase changes when the two sides are close enough to cause significant attrition calculations. Therefore, small values of S cause the phase to change early in the computation, large values cause it to change later. We varied S between 1 and 16.

Each unit's speed is chosen uniformly at random in such a way that a unit requires between 3.25 and 6.5 time steps to traverse a hex. Each unit can move in one of three directions: straightforward towards the opposing side, forward and angling upwards, and forward and angling downwards. The simulation is run for 60 time steps. A unit moves forward so long as it perceives no more than three enemy units on hexes adjacent to it. ϕ is taken to be $1/60$, α and β are taken to be 0.1, τ is 0.7, and we do not use an end of computation protection mechanism. Again these parameter values are selected heuristically to reflect observed general tendencies.

The two alternate mappings were chosen so as to optimize performance before the phase change, and after. The prechange mapping partitions the domain into eight 64×32 hex regions; the postchange mapping partitions the domain into 2^{14} 1×1 regions. The measured remapping cost is again low; we found it to be roughly equivalent to the time required by one time step during the combat phase.

A phase change occurs when the two sides are close enough to cause significant attrition computations. We can control the point in time where this occurs by adjusting S. As we did for the fluids computation, we plot in Fig. 7 the relative performance of static mapping compared to dynamic mapping. The horizontal axis plots $2S$, the number of hexes between the two sides at step 0. Large S corresponds to a long first phase, small S corresponds to a long second phase. The fixed-threshold heuristic again performed nearly optimally. In this case, the parameters required to solve the optimality equations are particularly difficult to determine. The "optimal" policy we used in this comparison is simply the unrealistic one of using, for every time step, the mapping that minimizes the execution time of that step. While we cannot implement that policy without remapping costs, we can and did measure the execution time of each cycle under different mappings. The

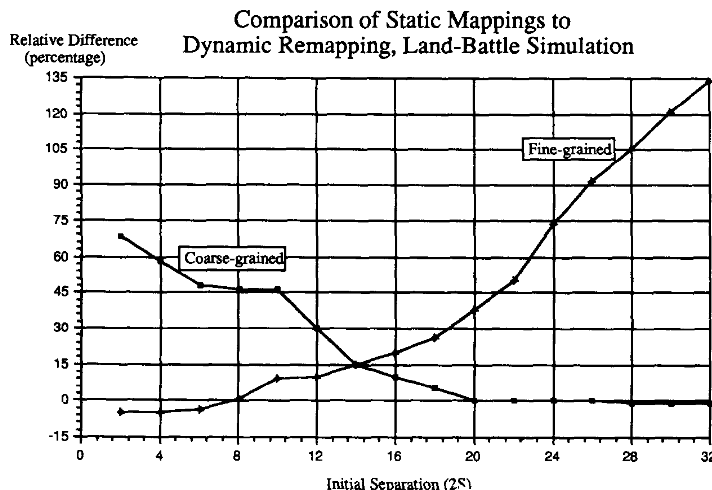

Fig. 7. Relative difference (in percentage) between two static mappings and dynamic remapping of land-battle simulation. Vertical axis is $100 \times (T_s - T_d)/T_d$ where T_s is static mapping execution time and T_d is dynamic remapping execution time. Horizontal axis is the initial number of hexes separating the two sides.

lower bound on the optimal execution is obtained by summing the minimized execution times at each step.

Like the fluids computation, dynamic remapping is much better than one of the static mappings when the phase changes either immediately, or near the end of the computation. Naturally, we could dispense with dynamic remapping altogether if we knew that the phase would change at one end or the other of the computation. The important point is that we risk poor performance if we choose either of these static mappings without *a prior* knowledge of the phase change. One might argue that we need only examine S in these experiments to acquire this *a priori* knowledge. In general, one cannot depend on such a simple analysis, because more realistic simulations allow interactive users to unpredictably control movement and engagement. Another option is to choose a static mapping that is suboptimal in both phases, but which does not fail miserably in either. The chief difficulty is that we may not be able to construct in advance a mapping that achieves an acceptable compromise. By contrast, dynamic remapping exploits our *a priori* understanding of extreme coarse and fine-grained partitions, even if we cannot *a priori* predict performance measures.

C. Discussion of Results

In view of the limited extent of these experiments, and the computing community's general suspicion that overheads should render dynamic remapping useless, it is helpful to reflect on the reasons remapping seems to work and whether it will continue to work on larger problems and machines.

A point in the favor of remapping's robustness is the fact that the test mechanisms used by the model problems do not accurately quantify postremapping performance. For both problems, the same mechanisms were used without modification on a number of instances whose computation and communication intensities differ from those presented. Remapping consistently yielded good performance gains. For the problems considered here, the fact that there *is* gain is enough to achieve that gain. While this may seen counterintuitive at first, it makes sense in light of the key problem parameters: $G = e_R - e_B$ (the per-cycle performance gain from remapping), D (the remapping cost), and T (the expected number of cycles for the second phase). Fig. 8 illustrates an envelope relating $G \cdot T$ and D. At the envelope's edge, we have scenarios where the remapping costs are just barely amortized by remapping gains. As one gets deeper into the envelope, remapping becomes relatively more effective, and there is much more room to deviate from the optimal policy and still get significant performance gains. The decision policy works well inside the envelope—it is not necessary to know precisely where the problem is inside the envelope, only that the problem *is* inside the envelope, so that responsible remapping is safe, and effective.

The statistical updating of p_n also plays an important role in achieving good performance. Our experience with both codes shows that false alarms from the testing mechanism occur often enough to make them a potential problem. If we were to remap *whenever* the mechanism said a phase had changed, we would often be premature. The Bayesian updating of p_n combined with a reasonably high decision threshold provides good protection against premature remapping.

Another important performance factor is that the length of time between a phase change and program termination was generally long enough to amortize the cost of remapping. The heuristic should work equally well on computations with more than one phase, provided that the average time between phase changes is long enough and the remapping gain is large enough to amortize the overhead costs. If phase durations are too short, dynamic remapping will degrade performance.

The performance differential between dynamic and static mapping relies on the ability of a "wrong" static mapping to seriously degrade overall performance. This need not always be the case, e.g., a code where the computation phase contributes far more to the overall running time than the communication phase. Dynamic remapping can still be expected to

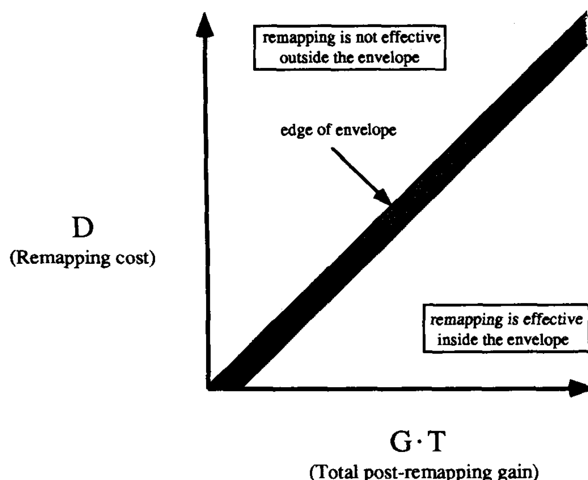

Fig. 8. Position within envelope determines utility of remapping.

perform well, it may just be that it does not perform notably better than a static mapping optimized for the computation phase.

The relatively low cost of remapping is a strong contributing factor to good performance—it places the problem deep inside of the envelope described by Fig. 8. We were at first surprised by this low cost; our expectations were that remapping would be quite expensive. Upon reflection though, one sees that this need not be the case. Indeed, on the computations most suitable for parallel computation it is not the case. The fundamental cost of remapping is the communication of data states. Communication (even in message passing scenarios) is essentially just copying information. Parallel computation is best suited for data parallel problems where the cost of computing a new data state is much larger than the cost of communicating that state, e.g., at a partition boundary. As the relative cost of computing a new data state increases, remapping becomes comparatively less expensive, and more important, in order to avoid load imbalance.

One might disregard our empirical experiments on the grounds that the problems considered are small, and that remapping costs will increase with larger problems and architectures. However, it is not difficult to see that the relative cost of remapping scales up. To facilitate this result we make the following definitions. Let P be the number of processors, and let L be the number of domain points. Let c_x be the time required to update the state of one point, during one cycle; let c_r be the cost of remapping one point from one processor to another; let b_r be the number of cycles between two remappings. The cost of executing b_r cycles on a serial machine is taken to be $L \cdot c_x \cdot b_r$. The parallel machine cost is affected by load imbalance, and by the remapping cost. To model load imbalance and communication inefficiencies, we define the *parallel efficiency* function $e(L, P)$. The time required to perform one step on L points, using P processors is taken to be $L \cdot c_x/(e(L, P) \cdot P)$. A value of $e(L, P) = 1$ implies perfect load balance and no degradation due to communication; overheads are modeled by decreasing $e(L, P)$. We make the reasonable assumption that $e(L, P)$ does not decrease as L increases when P is fixed (indeed, for a fixed number of processors one expects it to increase [23]). On architectures with sufficiently rich communication topologies, one can overlap the remapping communication. We model this by defining the fraction h to be such that the time required to remap is $h \cdot L \cdot c_r$; $h = 1$ implies complete serialization of the remapping communication, $h = 1/P$ implies complete parallelization. Using these notions, the speedup measured between two remappings is

$$\text{Speedup} = \frac{b_r \cdot L \cdot c_x}{\dfrac{b_r \cdot L \cdot c_x}{e(L, P) \cdot P} + h \cdot L \cdot c_r} = \frac{b_r \cdot c_x}{\dfrac{b_r \cdot c_x}{e(L, P) \cdot P} + h \cdot c_r}.$$

Here we see that for fixed P, the speedup depends on L only in the term $e(L, P)$. Since $e(L, P)$ does not decrease as L increases, we see that as L grows, the speedup does not decrease due to increasing remapping costs, even if those costs are completely serial. Consequently, remapping costs scale in the problem size, when the number of processors is fixed. The behavior as both L and P increase is also of interest. A reasonable choice of h is $h = (\log P)/P$; for example, this is achieved by implementing remapping using a crystal routing [12] that moves all L points in parallel $\log P$ times. In this case, speedup can be written as

$$\text{Speedup} = \frac{P}{\dfrac{1}{e(L, P)} + \dfrac{c_r \log P}{b_r c_x}}.$$

Speedup will increase as L and P are simultaneously increased provided that $e(L, P)$ does not decrease too rapidly. Note that $e(L, P)$ depends entirely on the problem and architecture. Consequently, dynamic remapping does not hinder a natural

tendency of a computation to get better speedups as it is scaled up.

The data reported here are not an exhaustive performance study of dynamic remapping, although extensive testing of a simulation model [20] supports our conclusions. Rather, the data we present should be taken as confirmation of the observations we have made, and of the utility of dynamic remapping. Much more work is needed before the user of a general parallel code can intelligently determine whether his code benefits from remapping, and what form that remapping should take.

VI. Conclusions

An effective mapping of workload to processors in a parallel processing system must make certain assumptions about the computation's running behavior. The behavior of many data parallel computations is characterized as a sequence of phases, where behavior within a phase is fairly stable, but the behavior between two phases can be quite different. A mapping can become ineffective when a phase change occurs, so that dynamically *remapping* the computation may be required to maintain good performance. The decision to remap must take into account the performance gains and costs involved, and must deal with uncertainty in remapping gains, the computation's future behavior, and the computation's termination time. We have modeled this decision problem with a Markov decision process, and determined the structure of the optimal decision policy. However, precalculation of this policy requires estimation of unknown parameter values. We therefore studied the performance of a simple threshold heuristic that does not assume knowledge of remapping's cost and gains. A study on two diverse multiprocessor codes shows that this heuristic works well, and shows that precise estimates of these parameters are not needed. The key issue for the remapping decision problem is thus seen to be the relatively accurate assessment of when remapping will lead to performance gains.

There are certainly limitations to the approach we have taken here. To use our policy it is necessary to have enough fore-knowledge of the computation to be able to write code that dynamically analyzes behavior and looks for remapping gain. Our model problems allowed fairly simple test mechanisms; more complex problems may require more complex analysis, which will tend to be application dependent. Nevertheless, there will always be codes that people agonize over in order to get the best possible performance; when those codes have unpredictable phase-like behavior our approach can significantly improve performance.

Appendix

In this Appendix, we prove Lemma 1 from Section IV.

Proof of Lemma 1: We use the notation $g(\langle p, n\rangle | N = m)$ to denote the value of function g at state $\langle p, n\rangle$ given that $N = m$. We will also say that a function g is plcc if it is piecewise linear, continuous, and concave.

Lemma 1's first claim follows immediately from $C_m(\langle p, n\rangle)$'s definition. The second claim is that for every n, $C_m(\langle p, n\rangle)$ is a plcc function of p. We employ the following lemma reported in [25] and stated in terms of our notation.

Lemma A-1: Suppose that $N = m$. If $V(\langle p, n+1\rangle | N = m)$ is a plcc function of p, then $E_v(\langle p, n\rangle | N = m)$ is a plcc function of p. □

First condition on $N = m$ for some m. We will inductively prove that $V(\langle p, n\rangle | N = m)$ and $C_t(\langle p, n\rangle | M = m)$ are plcc functions of p. For $n > m$ we have $V(\langle p, n\rangle) = C_t(\langle p, n\rangle) = 0$, which is trivially plcc. For the induction base let $n = m$. Then

$$C_t(\langle p, m\rangle | N = m) = pe_B + (1-p)e_F + E_v(\langle p, m\rangle | N = m)$$
$$= pe_B + (1-p)e_F,$$

showing that $C_t(\langle p, m\rangle | N = m)$ is plcc. $C_m(\langle p, m\rangle | N = m)$ is also plcc since it is linear. The class of plcc functions is closed under the pointwise minimum operation; $V(\langle p, m\rangle | N = m)$ must also be plcc, establishing the induction base.

For the induction hypothesis we suppose that both $C_t(\langle p, n+1\rangle | N = m)$ and $V(\langle p, n+1\rangle | N = m)$ are plcc functions of p for some $n \leq m - 1$. Lemma A-1, and the closure of plcc functions under addition and pointwise minimum again ensure that $C_t(\langle p, n\rangle)$ and $V(\langle p, n\rangle)$ are plcc functions of p, completing the induction.

To complete the proof, we note that the class of plcc functions is also closed under scalar multiplication, and observe that $V(\langle p, n\rangle) = E[V(\langle p, n\rangle | N = m)]$ and $C_t(\langle p, n\rangle) = E[C_t(\langle p, n\rangle | N = m)]$ are finite weighted sums of known plcc functions.

To help establish Lemma 1's third and fourth claims, we analyze the values of $C_m(\langle p, n\rangle)$ and $C_t(\langle p, n\rangle)$ at $p = 1$. Key results are given by Lemma A-2.

Lemma A-2: Either

i) $V(\langle 1, n\rangle) = C_m(\langle 1, n\rangle)$ for all n for which Prob $\{N = n\} \neq 0$; or

ii) $V(\langle 1, n\rangle) = C_t(\langle 1, n\rangle)$ for all n for which Prob $\{N = n\} \neq 0$; or

iii) There exists an n_0 (possibly ∞) such that for all $n < n_0$, $V(\langle 1, n\rangle) = C_m(\langle 1, n\rangle)$, and for all $n \geq n_0$ for which Prob $\{N = n\} \neq 0$, $V(\langle 1, n\rangle) = C_t(\langle 1, n\rangle)$.

Proof: Condition on $N = m \leq M$. Let K be the largest integer such that $(e_B - e_R) \cdot K \leq D$. Simple algebra (omitted here) establishes the inductive proof that for all n such that $m - K < n \leq m$,

$$V(\langle 1, n\rangle | N = m) = C_t(\langle 1, n\rangle | N = m) = (m - n + 1)e_B;$$

and that for $0 \leq n \leq m - K$,

$$V(\langle 1, n\rangle | N = m) = C_m(\langle 1, n\rangle | N = m)$$
$$= D + (m - n + 1)e_R$$

and

$$C_t(\langle 1, n\rangle | N = m) = e_B + D + (m - n)e_R.$$

Define $d(n | N = m) = C_m(\langle 1, n\rangle | N = m) - C_t(\langle 1, n\rangle | N = m)$, and $d(n) = C_m(\langle 1, n\rangle) - C_t(\langle 1, n\rangle)$. Then

$$d(n|N=m) = \begin{cases} D & \text{for } n > m \\ D - (e_B - e_R)\cdot(m-n+1) & \text{for } m-K < n \le m \\ (e_R - e_B) & \text{for } n \le m-K. \end{cases}$$

By the definition of K, $d(n|N=m)$ is an increasing function of n. Consequently $d(n)$ increases in n, since

$$d(n+1) - d(n) = \sum_{m=1}^{M} \text{Prob}\{N=m\} \cdot (d(n+1|N=m) - d(n|N=m)) \ge 0.$$

Case i) occurs if $d(n)$ is negative for all n, case ii) occurs if $d(n)$ is positive for all n, and case iii) occurs if $d(n)$ changes sign at $n = n_0$. □

To establish Lemma 1's third claim we will show that $C_t(\langle p, n\rangle)$ is linear in p whenever $n \ge n_0$. Since $C_m(\langle \cdot, n\rangle)$ is always linear, and $C_t(\langle 0, n\rangle) \le C_m(\langle 0, n\rangle)$ for all n, and $C_t(\langle 1, n\rangle) \le C_m(\langle 1, n\rangle)$ when $n \ge n_0$, it follows directly that C_t and C_m cannot intersect, so that $C_t(\langle p, n\rangle) \le C_m(\langle p, n\rangle)$ for all $p \in [0, 1]$.

Lemma A-3: If $n \ge n_0$, then $C_t(\langle p, n\rangle)$ is linear in p, and $V(\langle p, n\rangle) = C_t(\langle p, n\rangle)$ for all $p \in [0, 1]$.

Proof: We proceed by induction. Note first that

$$V(\langle p, M\rangle) = \text{Prob}\{N=M\}\cdot \min \begin{cases} D + pe_R + (1-p)e_F \\ pe_B + (1-p)e_F. \end{cases}$$

Presuming that $n_0 < M$, we have $C_t(\langle 1, M\rangle) \le C_m(\langle 1, M\rangle)$ and $C_t(\langle 0, M\rangle) \le C_m(\langle 0, M\rangle)$, so that

$$V(\langle p, M\rangle) = \text{Prob}\{N=M\} C_t(\langle p, M\rangle) = \text{Prob}\{N=M\}(pe_B + (1-p)e_F)$$

which is linear in p. The induction base is thus satisfied.

For the induction hypothesis, we suppose there is an $n > n_0$ such that $V(\langle p, n+1\rangle) = C_t(\langle p, n+1\rangle)$ for all $p \in [0, 1]$, and that $C_t(\langle p, n+1\rangle)$ is linear in p. We know that

$$C_t(\langle p, n\rangle) = \text{Prob}\{N \ge n\}(p\cdot e_B + (1-p)e_F) + q^c(p) \cdot V(\langle p^c(p), n+1\rangle) + q^{\bar{c}}(p)\cdot V(\langle p^{\bar{c}}(p), n+1\rangle),$$

and the induction hypothesis states that

$$V(\langle p, n+1\rangle) = A\cdot p + B$$

for some A and B. Since $p^c(p) = p_a(p)(1-\beta)/q^c(p)$ and $p^{\bar{c}}(p) = p_a(p)\beta/q^{\bar{c}}(p)$, it follows that

$$C_t(\langle p, n\rangle) = pe_B + (1-p)e_F + Ap_a(p) + B$$

which is linear in p. Since $C_m(\langle p, n\rangle)$ and $C_t(\langle p, n\rangle)$ cannot intersect it follows directly that $C_m(\langle p, n\rangle)$ exceeds $C_t(\langle p, n\rangle)$ for all $p \in [0, 1]$. Thus, $V(\langle p, n\rangle) = C_t(\langle p, n\rangle)$, completing the induction. □

References

[1] I. Babuska, Ed., *Adaptive Computational Methods for Partial Differential Equations*. Philadelphia, PA: SIAM, 1983.

[2] M. J. Berger and S. Bokhari, "A partitioning strategy for nonuniform problems on multiprocessors," *IEEE Trans. Comput.*, vol. C-36, no. 5, pp. 570–580, May 1987.

[3] M. J. Berger and J. Oliger, "Adaptive mesh refinement for hyperbolic partial differential equations," *J. Computat. Phys.*, vol. 53, pp. 484–512, 1984.

[4] S. Bokhari, "Partitioning problems in parallel, pipelined, and distributed computing," *IEEE Trans. Comput.*, vol. 37, no. 1, pp. 48–57, Jan. 1988.

[5] D. L. Book, Ed., *Finite-Difference Techniques for Vectorized Fluid Dynamics Calculations*. New York: Springer-Verlag, 1981.

[6] W. W. Chu, L. J. Holloway, M. Lan, and K. Efe, "Task allocation in distributed data processing," *IEEE Comput. Mag.*, vol. 13, no. 11, pp. 57–69, Nov. 1980.

[7] W. W. Chu and K. K. Leung, "Module replication and assignment for real-time distributed processing systems," *Proc. IEEE*, May 1987.

[8] Y. Chow and W. Kowhler, "Models for dynamic load balancing in a heterogeneous multiple processor system," *IEEE Trans. Comput.*, vol. C-28, no. 5, pp. 354–361, May 1979.

[9] T. C. Chou and J. A. Abraham, "Load balancing in distributed systems," *IEEE Trans. Software Eng.*, vol. SE-8, no. 4, pp. 401–412, July 1982.

[10] D. L. Eager, E. D. Lazowska, and J. Zahorjan, "Adaptive load sharing in homogeneous distributed systems," *IEEE Trans. Software Eng.*, vol. SE-12, no. 5, pp. 662–675, May 1986.

[11] G. J. Foschini, "On heavy traffic diffusion analysis and dynamic routing in packet switched networks," in *Computer Performance*, K. M. Chandy and M. Reiser Eds. New York: North-Holland, 1977.

[12] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker, *Solving Problems on Concurrent Computers*. Englewood Cliffs, NJ: Prentice-Hall, 1988.

[13] J. B. Gilmer, "Documentation, state-space reconciliation version of the zipscreen prototype simulation," Tech. Rep., BDM Corp., 1986.

[14] D. Gusfield, "Parametric combinatorial computing and a problem of program module distribution," *J. ACM*, vol. 30, no. 3, pp. 551–563, July 1983.

[15] P. R. Ma, E. Y. S. Lee, and M. Tsuchiya, "A task allocation model for distributed computing systems," *IEEE Trans. Comput.*, vol. C-31, no. 1, pp. 41–47, Jan. 1982.

[16] N. Matelan, "The Flex/32 Multicomputer," in *Proc. 12th Int. Symp. Comput. Architecture*. Los Alamitos, CA: Computer Society Press, June 1985, pp. 209–213.

[17] D. I. Moldovan and J. A. B. Fortes, "Partitioning and mapping algorithms into fixed size systolic arrays," *IEEE Trans. Comput.*, vol. C-35, no. 1, pp. 1–12, Jan. 1986.

[18] L. M. Ni, C. Xu, and T. B. Gendreau, "A distributed drafting algorithm for load balancing," *IEEE Trans. Software Eng.*, vol. SE-11, no. 10, pp. 1153–1161, Oct. 1985.

[19] D. M. Nicol, "Mapping a battlefield simulation onto parallel message-passing architectures," in *Proc. 1988 SCS Conf. Distrib. Simulation*, San Diego, CA, 1988, pp. 141–146.

[20] D. M. Nicol and P. F. Reynolds, Jr., "Dynamic remapping decisions in multi-phase parallel computations," ICASE Rep. 86-48, Sept. 1986.

[21] D. M. Nicol and J. H. Saltz, "Dynamic remapping of parallel computations with varying resource demands," *IEEE Trans. Comput.*, vol. 37, no. 9, pp. 1073–1087, Sept. 1988.

[22] D. M. Nicol and J. Townsend, "Accurate modeling of parallel scientific computations," in *Proc. 1989 SIGMETRICS Conf.*, May 1989, Berkeley, CA, pp. 165–170.

[23] D. M. Nicol and F. Willard, "Problem size, parallel architecture, and optimal speedup," *J. Parallel Distrib. Comput.*, vol. 5, pp. 404–420, Aug. 1988.

[24] C. C. Price and U. W. Pooch, "Search techniques for a nonlinear multiprocessor scheduling problem," *Naval Res. Logistics Quarterly*, vol. 29, no. 2, pp. 213–233, June 1982.

[25] A. Rapoport, W. E. Stein, and G. J. Burkheimer, *Response Models for Detection of Change*. Boston, MA: Reidel, 1979.

[26] S. Ross, *Applied Probability Models with Optimization Applications*. San Francisco, CA: Holden-Day, 1970.

[27] S. A. Schmitt, *An Elementary Introduction to Bayesian Statistics*. Reading, MA: Addison-Wesley, 1969.

[28] J. A. Stankovic, "An application of Bayesian decision theory to decentralized control of job scheduling," *IEEE Trans. Comput.*, vol. C-34, no. 2, pp. 117–130, Feb. 1985.

[29] J. A. Stankovic, K. Ramamritham, and S. Cheng, "Evaluation of a flexible task scheduling algorithm for distributed hard real-time systems," *IEEE Trans. Comput.*, vol. C-34, no. 12, pp. 1130–1143, Dec. 1985.

[30] D. Towsley, "Queueing network models with state-dependent routing," *J. ACM*, vol. 27, no. 2, pp. 323–337, Apr. 1980.
[31] A. N. Tantawi and D. Towsley, "Optimal static load balancing," *J. ACM*, vol. 32, no. 2, pp. 445–465, Apr. 1985.

David M. Nicol received the B.A. degree (magna cum laude, Phi Beta Kappa) in mathematics from Carleton College, Northfield, MN, in 1979, and the M.S. and Ph.D. degrees from the University of Virginia, Charlottesville, in 1983 and 1985.

From 1979 to 1982 he was a Programmer/Analyst with the Information Sciences Division of the Control Data Corporation, where he designed and implemented embedded parallelized real-time image processing systems. From 1985 to 1987 he was a Staff Scientist at the Institute for Computer Applications in Science and Engineering (ICASE). He is currently an Assistant Professor at the College of William and Mary, Williamsburg, VA. His main research interests are in the design and analysis of static and dynamic methods for mapping scientific/engineering problems onto parallel architectures, and in the design and analysis of methods for parallelizing discrete-event simulations.

Dr. Nicol is a member of the IEEE Computer Society, the Association for Computing Machinery, and the Operations Research Society of America.

Paul F. Reynolds, Jr., received the Ph.D. degree from the University of Texas at Austin, in 1979.

He is an Associate Professor of Computer Science and founder and former director of the Institute for Parallel Computation at the University of Virginia. He has been a member of the faculty at University of Virginia since 1980. He has published in the area of parallel computation, specifically in parallel simulation, and parallel language and algorithm design. He has served on a number of national committees and advisory groups including an oversight group for the National Testbed. He has been a consultant to numerous corporations and government agencies in the systems and simulation areas, and he has been a Research Associate at NASA, Langley, Hampton, VA, since 1985. On sabbatical, he was a visiting faculty member at the University of Victoria, Victoria, B.C., during the summer of 1989 and visiting NASA Langley and working quietly at home during the 1989/1990 academic year.

Dr. Reynolds is a member of the IEEE Computer Society and the Association for Computing Machinery.