NASA Contractor Report 178378

ICASE REPORT NO. 87-52

ICASE

SCHEDULES FOR MAPPING IRREGULAR PARALLEL COMPUTATIONS

David M. Nicol

Joel H. Saltz

Contract No. NAS1-18107

September 1987

INSTITUTE FOR COMPUTER APPLICATIONS IN SCIENCE AND ENGINEERING
NASA Langley Research Center, Hampton, Virginia 23665

Operated by the Universities Space Research Association

National Aeronautics and Space Administration

Langley Research Center
Hampton, Virginia 23665

Schedules for Mapping Irregular Parallel Computations

David M. Nicol *
The College of William and Mary

Joel H. Saltz †
Yale University

Abstract

A large class of scientific computational problems can be characterized as a sequence of steps where a significant amount of computation occurs each step, but the work performed at each step is not necessarily identical. Two good examples of this type of computation are (1) regridding methods which change the problem discretization during the course of the computation, and (2) methods for solving sparse triangular systems of linear equations. Recent work has investigated a means of mapping such computations onto parallel processors; the method defines a family of *static* mappings with differing degrees of importance placed on the conflicting goals of good load balance and low communication/synchronization overhead. The performance tradeoffs are controllable by adjusting the parameters of the mapping method. To achieve good performance it may be necessary to dynamically change these parameters at run-time, but such changes can impose additional costs. If the computation's behavior can be determined prior to its execution, it can be possible to construct an optimal parameter schedule using a low-order-polynomial-time dynamic programming algorithm. We illustrate this on two model problems. Because the dynamic programming algorithms can be too expensive, we study the performance of an expected linear-time scheduling heuristic on one of the model problems and show that it is effective, and nearly optimal. The concepts we discuss here are quite general, and apply to a wide variety of situations.

*This research was supported in part by the National Aeronautics and Space Administration under NASA contract NAS1-18107 while the author was in residence at ICASE, Mail Stop 132C, NASA Langley Research Center, Hampton, VA 23665.

†Supported in part by NASA contract NAS1-18107, the Office of Naval Research under contract No. N00014-86-K-0654, and NSF grant DCR 8106181.

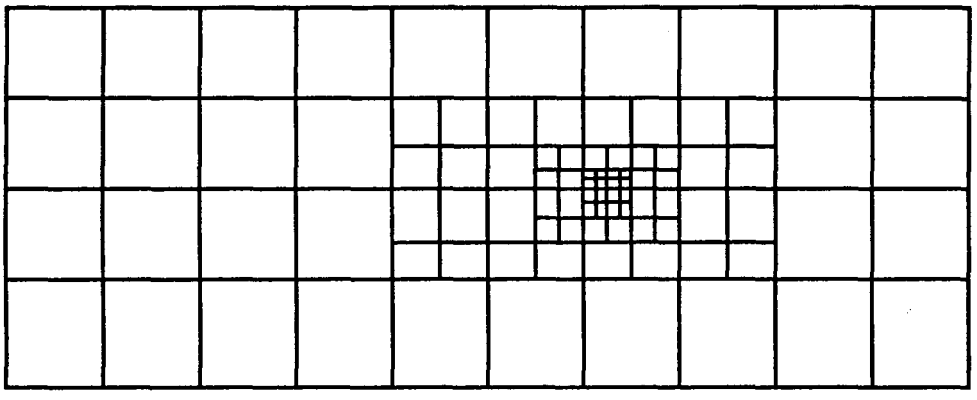


Figure 1: Irregular grid for two-dimensional PDE

1 Introduction

Two of the principle sources of efficiency loss in parallel computation are load imbalance and communication/synchronization overhead. In preselected, very regular problems it is often quite straightforward to find a problem decomposition that provides a favorable load balance while requiring minimal overhead costs. In problems with either explicitly or implicitly varying workloads, different and more general partitioning approaches must be taken.

A class of numerical methods provides good examples of computations with explicit dynamically varying workload. Finite-difference algorithms for solving partial differential equations operate on a grid representing a discretized approximation to a continuous domain. The computational effort required to adequately solve the problem depends in large part on the number of grid points. For the sake of efficiency, time-dependent problems are often solved using dense grids in domain regions where the solution changes rapidly (in time or space), and coarse grids in more stable regions. An irregular grid of this type is illustrated in figure 1; the intersection of lines denote a grid point. In time-dependent problems, explicit integration schemes may use different irregular grids at different time-steps in response to the anticipated behavior of the solution. This paper adopts such an algorithm as a model computation, as it is representative of many in scientific computing—a sequence of steps with potentially variable computational demands. Within a step there are many operations which could be performed in parallel, thereby reducing execution time, and extending the size of problems which can reasonably be formulated for solution. But the irregularity of workload between steps makes it difficult to compute an optimal

scheme for mapping the workload onto a parallel, message-passing computer. In general, we can only hope to discover effective mapping heuristics.

Implicit problem irregularity is possible in problems lacking an obvious time dependence. As an illustration, our second model problem is a solution method for very sparse triangular systems of linear equations; such systems exhibit run-time irregularity due to dynamically changing degrees of available parallelism. Here again the problem irregularity makes it difficult to find a truly optimal workload mapping, we must turn to well-founded but heuristic methods.

Many irregular problems can be statically mapped using a generic approach discussed in [7], [14], [11], [20]. In its fullest generality, this method operates on multi-dimensional domains where the computational workload is tied to points in the domain, and where communication tends to be between points which are close. The method first aggregates geographical portions of the domain into schedulable *work units*, and then statically assigns each work unit to a processor. The assignment is constructed in such a way that any work unit in any processor is geographically "close" to some work unit in every other processor. Applied to our first model problem, this method aggregates grid point solutions within uniformly sized domain regions; The effective mapping parameter is the spatial size of these regions; the number of time-steps or synchronizations required by the computation is completely unaffected by this parameter. Spatially small work units are fine-grained and tend to balance the load, but increase the communication requirements. A detailed study in [18] uses the generic method to statically map our second model problem. Work units in this case are groups of variables representing geographically close grid points in a two-dimensional domain. However, in this case the choice of work unit granularity *defines* a sequence of computational wavefronts that are used as phases. That study assumed that all work units within a wavefront are completed before any work units in the next wavefront can be completed. Again, fine-grained work units tend to balance load well but increase the number of synchronizations, each with an attendant delay cost. Thus for both problems we see a performance trade-off between imbalance and communication/synchronization overhead that is controlled by the choice of work unit granularity.

It is possible to dynamically change work unit granularity between time-steps in the irregular gridding problem; this allows better performance by making the mapping more sensitive to the changing irregularities of the grid. The work unit granularity in the triangular solve method is not so easily changed dynamically. However, for a fixed granularity it is still possible to dynamically choose the synchronization points—while it is *sufficient* to enforce data dependencies by globally synchronizing between wavefronts, it is not *necessary*. By paying careful attention to data dependencies, it can be possible to concurrently evaluate work units from different wavefronts. The advantage of doing so is illustrated by example. Imagine two wavefronts, each with three work units all having unit time execution requirements, and suppose a synchronization requires half a time unit. Using two processors that synchronize between wavefronts, the computation requires four and a half time units. If two work units in different wavefronts can be concurrently evaluated, then

with two synchronizations the computation can be completed in four time units. Judicious selection of synchronization points can consequently improve the load balance with gains large enough to offset additional synchronization costs.

This alternate selection of synchronization points affects an important mapping parameter: the amount of work assigned to a processor between two synchronizations. The static method discussed in [18] allows one to parametrically adjust the computational granularity. This adjustment of the computational granularity in many cases leads to a partition consisting of a number of work units that is relatively small number compared to the number of available processors. This may, in many cases, lead to a poor balance of load unless care is taken in the assignment of work units to processors. Appropriate selection of synchronization points should facilitate good load balancing in these cases.

Our two model problems are similar in that both can benefit from run-time adjustment of mapping parameters. The first problem has a fixed number of synchronizations and can dynamically change work unit granularity; the second problem fixes granularity and changes the number of synchronizations. Yet despite these apparent differences, the model problems share an important feature. If the computation's behavior is known a priori and the execution, communication, and synchronization costs are known, then it may be possible to increase performance by pre-scheduling values for the mallable mapping parameters. For both problems an optimal parameter schedule can be constructed using a low-order-polynomial-time dynamic programming algorithm. However, these schedules can still be relatively expensive to compute compared to the cost of performing the computation. Consequently, for the second model problem we study the performance of a linear-time scheduling heuristic. We find that the heuristic performs well.

For the sake of clarity, much of our discussion focuses on our two model problems. However, it is important to realize that the principles and concepts we apply are quite general, and apply to a large and significant class of parallel computations. This work is part of the ongoing Crystal/ACRE [19] parallel programming and run-time environment development effort. We aim to develop simple sets of techniques that can be used for the automated mapping and dynamic remapping of a variety of problems onto both loosely coupled systems and tightly coupled systems. Control over performance through mapping parameters is very promising for these purposes; construction of parameter schedules is a necessary and logical extension of this approach.

In §2 we present our first model problem, and explain the mapping method used in this study. We then discuss estimation of execution, communication, and remapping costs, and develop the granularity scheduling algorithm. §3 presents our second model problem, a sparse triangular system solution method, and discusses both a dynamic programming solution method for scheduling synchronization points, and a linear-time heuristic for this problem. Performance data shows that the heuristic is quite effective. Finally, §4 summarizes this paper.

2 Problems with Strictly Defined Phases and Irregular Workloads

We will first consider the solution of a time-dependent partial differential equation. This solution has the characteristic that the computation's phases are defined independently of the problem granularity or mapping. For any given granularity different work units may have different execution requirements, depending on the density of grid points assigned to them. Because the grid changes in time, the workload requirements of any particular work unit can change. It is this type of irregularity that permits increased performance by changing the workload granularity during execution. A wide variety of scientific problems have these characteristics; the type of approach developed for this particular problem is applicable to others.

Consider the one-dimensional wave equation

$$\frac{\partial u}{\partial t} = -v\frac{\partial u}{\partial x}$$

where u represents some density function, and v represents some wave velocity function. A numerical solution to a problem of this type is to discretized the interval of interest with a uniform grid having points Δx distance apart. The solution for u is sought at times Δt, $2\Delta t$, $3\Delta t$, and so on. The value of u at the jth grid point at time $n\Delta t$ is denoted u_j^n. There are a variety of ways to numerically solve this problem. The Lax method of solution for u [16] is to solve the equations

$$u_j^{n+1} = \frac{1}{2}\left(u_{j+1}^n + u_{j-1}^n\right) - \frac{v_j^{n+1}\Delta t}{2\Delta x}\left(u_{j+1}^n - u_{j-1}^n\right). \quad (1)$$

So-called *equations of state* dictate the solution for v_j^{n+1} which is required here; for simplicity we focus only on equation (1). A commonly studied equation of state is the identity $v = u/2$ (Burger's equation), in which case only equation 1 need be solved. Boundary conditions also need to be specified, but that issue does not affect the focus of this paper. Without loss of generality we suppose that u is solved over the unit interval $[0, 1]$.

2.1 Solution Method

The dynamic regridding technique discussed in [2] creates a hierarchy of different grid density levels. Figure 2 illustrates a hierarchy of three grid levels on a one-dimensional domain. All grids are superimposed on the interval $[0, 1]$, but the problem is treated with different grid resolutions at different steps in the algorithm. We assume that the coarsest grid has 2^N grid points, with a spacing of $\Delta x = 2^{-N}$. Each step in the resolution refinement decreases Δx by a factor of two. The additional grid levels are added to regions where the solution exhibits rapid change. We call the coarsest grid level the $0th$ refinement level, the next coarsest level the $1st$ level, and so on. As illustrated by figure 2, any geographical

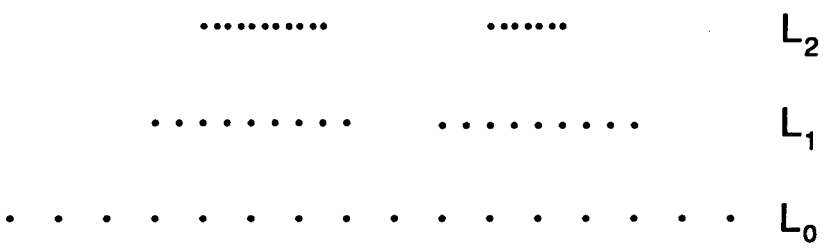


Figure 2: Grid hierarchy for one-dimensional problem

point x which falls within a region discretized by a grid at level i must also fall within a region discretized by a grid at level $i-1$. We will say that the grid at level $i-1$ *covers* the grid at level i. `integrate(G,i,t,Δt)` is a function which solves equation 1 on a grid `G` at level i (thus specifying Δx) and time t, with time step Δt. Communication between grid levels is carried out by a function `update(G,i,t,Δt)` with the same parameters. `update` is most easily described recursively, as shown in figure 3. The action of updating a grid

```
update(G, i, t, Δt) {
    integrate(G, i, t, Δt);
    For all grids G' covered by G {
        interpolate G to create function values on G' at time t;
        update(G', i+1, t,(Δt)/2);
        update(G', i+1, t+(Δt)/2, (Δt)/2);
        Replace G's grid points matching G' by copying;
        }
    }
```

Figure 3: Update function for grid at level i

at level i calls for two updates (with smaller time-steps) of all grids at level $i+1$ which it covers. Figure 2 gives a simple example of a grid hierarchy. Note that grid levels L_1 and L_2 are not contiguous; the collection of updates to contiguous subgrids at level i is semantically considered to be an update of grid L_i. According to the update algorithm, the middle grid (level) in figure 2 is updated twice for every update of the coarse grid, and the finest grid is updated twice for every middle grid update. The complete integration of the coarsest grid from time t to $t+\Delta t$ calls for a sequence of grid updates: $L_0, L_1, L_2, L_2, L_1, L_2, L_2$.

The analysis in [2] shows how to dynamically generate the grid hierarchy in response to the solution behavior. This paper presumes that the dynamic grid hierarchy is known in advance. This is reasonable if the code has been run before so that the solution behavior is predictable; it is also reasonable if the user understands his problem well enough to place the fine grids in regions of interest (there is an advantage to this, as dynamic grid generation is computationally expensive). New grids are typically employed only every few (say 5) coarse grid updates. Because of this, it is feasible for the gridding schedule for the entire computation (at least large portions of it) to be pre-loaded at initialization. All that is needed is a description of the *extent* of various grids at various levels. For example, the hierarchy shown in figure 2 is described by four endpoint pairs (the level zero grid always extends from 0 to 1, and so need not be described). This compact description suffices for 5 coarse grid updates, which is 35 grid updates at the various different levels.

It is possible to implement this algorithm very simply if a maximum grid level L is known. All that is needed is a single grid, with spatial resolution $\Delta x = 2^{-(N+L)}$ (recall that the solution is sought over $[0,1]$). When grid level i is updated, only certain grid points $L-i$ points apart are involved. The copying of values from high order grids to low order grids at common geographical points is therefore done implicitly. Likewise, much of the initialization of a level $i+1$ grid from a level i grid is implicitly done. This method does suffer from inefficient use of space; however, the space it does require is the same as that required in the worst case of any other grid management scheme. Because this scheme simplifies the task of estimating execution and remapping costs, we will tacitly assume its use.

2.2 Wrapping to Map Work Units

The aggregation/assignment method discussed in [7], [14], and [20] divides a domain into kP equal-sized *work units*, P being the number of available processors, and k being some positive integer. We assume that $P = 2^p$, although the mapping method does not require this. The work units are assigned in a regular manner to the processors so that every work unit in every processor is "close" to some piece in any other processor. Locality of workload intensity in space then gives us a certain degree of load balance, without explicit consideration of the load being balanced. Figure 4 illustrates one possible aggregation and assignment of a one-dimensional domain grid hierarchy. A processor receiving the subregion $[j/2^d, (j+1)/2^d)$ is responsible for updating all grid points at all grid levels in the domain region $[j/2^d, (j+1)/2^d)$. These subregions define work units; because of the grid irregularity different work units will have different computational requirements. The work unit corresponding to $[j/2^d, (j+1)/2^d)$ is given the index j; with P processors available, work unit j is assigned to processor $j \bmod P$. Called *wrapping*, this type of scheme is easily extended to higher dimensions: a k-dimensional domain is aggregated into work units which are indexed by k-coordinate vectors, and the processors are indexed by k-coordinate vectors. Work units which differ in exactly one coordinate position are logically adjacent

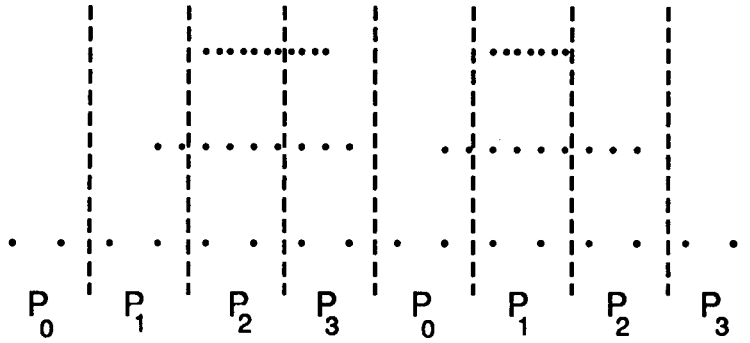


Figure 4: Aggregation and assignment of one-dimensional domain

in the domain. The *ith* processor index coordinate falls within $[0, p_i - 1]$. Then work unit $(d_0, d_1, \ldots, d_{k-1})$ is assigned to processor $(d_0 \bmod p_0, d_1 \bmod p_1, \ldots, d_{k-1} \bmod p_{k-1})$. We will assume that a work unit length can be no smaller than 2^{-N}, the inverse of the number of coarse grid points. Smaller lengths can be implemented, but lead to anomalous situations requiring special treatment.

Empirical data in [14] and [20], analytic modeling in [14], and common sense suggest that as the size of the aggregated work units decrease (and hence the number increase), the quality of the load balance increases. But communication costs tend to increase as the number of work units increases. For example, it is clear from equation (1) that a parallel solution of the model problem requires communication between processors P_i and P_{i+1} whenever grid point u_j is assigned to P_i and u_{j+1} is assigned to P_{i+1} [1]. As the number of work units increases, the number of such communications increases. Consequently, the work unit size controls an inherent performance trade-off between load imbalance and communication overhead.

For each step we could plot performance vs. granularity, and determine the optimal degree of granularity for that step. The optimal granularity can therefore change between steps, with a possibly profound impact on performance. For example, experiments reported in [12] consider the performance when granularity is dynamically changed; extreme per-step sensitivity to granularity is observed. Also, measurements taken on a battlefield simulation suggest that different granularities are called for at different simulation time-steps [11]. It is consequently important to be able to change work unit granularity in response to changing workload behavior. To do this optimally we must first estimate the step execution times under varying granularities, and must estimate the costs of changing granularity. We may

[1] Throughout this paper we assume that indexing arithmetic is modulo the number of indexed objects.

then formulate an optimization problem which chooses a granularity for each step. These topics are addressed in the subsection to follow.

The technique of aggregating the domain into equal sized pieces is not as sensitive to precise load distribution as are other mapping techniques such as binary dissection [1]. However, our approach has several important advantages. First, inter-processor communication is guaranteed to be very local—this is not true with other mapping methods. [2] This is especially important as high communication costs can easily defeat an otherwise perfectly balanced load. Secondly, our method is less sensitive to changes in the load distribution and is less sensitive to miss-estimation of execution costs. Finally, our method yields a significantly smaller space of possible mappings than a method like binary dissection. While this can be viewed as a disadvantage for static problems, it is actually an advantage for dynamic problems, provided that the space includes effective mappings. Fewer possibilities need to be considered as the optimal sequence of mappings is sought, leading to an efficient scheduling algorithm. Indeed, the principle reason we are able to derive a polynomial-time granularity scheduling algorithm is that the number of possible mappings at a given scheduling step is polynomial.

2.3 Granularity Scheduling

The first step in constructing a granularity schedule is to estimate the run-time costs under differing granularities, and to estimate the costs of changing granularity. The regularity of the regridding model problem makes it possible to estimate its execution, communication, and remapping costs. The basic idea behind the estimation procedures is that knowledge of grid placement and algorithm behavior allows one to calculate the computational and communication workload at any given grid update. This type of approach has already been fruitfully applied to a simplier numerical problem in [17] and [15].

As seen in §2.1, our first model problem can be viewed as a sequence of grid level updates. If we can estimate the costs of a single grid level update, we may then apply the estimation procedure to each grid level in the sequence. The exact details of such an estimation procedure depend very much on the code and architecture. However, it is not unreasonable to assume that we can estimate or measure the execution time of a subgrid integration as a function of subgrid length and density, or to estimate or measure communication and synchronization costs as a function of communication volume. Foreknowledge of grid placement and sequencing then permits estimation of the time required to update a grid level, *as a function of the work unit granularity.* We do not intend to trivialize here the complexity of parametrically characterizing a program's execution behavior. A detailed treatment of this problem will be found in a forth-coming technical report [3]. For our purposes here it suffices to argue that the knowledge of grid point

[2] For common architectures such as grids and hypercubes, binary dissection will give local communication in one-dimensional domains, but need not in higher dimensional domains.

[3] D. Nicol, J. Saltz, J. Townsend, *Estimating run-time costs in parallel computations for the purposes of*

```
for j:=0 to p-1 {
   perform perfect shuffle on Z;
     if(bit j of my_adrs = 0) then
       exchange Z[m/2] - Z[m-1] with neighbor in jth dimension;
     else
       exchange Z[0] - Z[m/2 -1] with neighbor in jth dimension;
}
```

Figure 5: Remapping communication algorithm

placement and program behavior allows us to either model or measure the dependency of performance on granularity.

There is a significant communication cost associated with changing granularity between grid level updates: the just-updated grid level (say j) is redistributed among processors in accordance with the new work unit size and the wrapping assignment. Partial results from as yet uncompleted updates to lower grid levels must be maintained; consequently all 2^{N+j} possible grid points at level j are redistributed during the remapping. To decrease work unit size (and hence increase granularity) every existing work unit is broken into smaller pieces which are then distributed. To increase work unit size the existing work units are routed to new destinations, where they are coalesced into larger work units. The volume of communication during a granularity change will depend on the level, j, of the grid being communicated. The following algorithm changes the number of work units per processor from w_{old} to w_{new}. For simplicity we suppose that both of these values are powers of two. If $w_{old} < w_{new}$, then a processor first divides each of its work units into w_{new}/w_{old} new work units, and computes the new processor location for each. It organizes the new work units as an array Z with w_{new} elements, each element holding a work unit of the new size. The array is initially sorted by processor destination; work units destined for the same processor are sorted by geographical position. If $w_{old} > w_{new}$, then the new processor destination for each old work unit is computed, and Z is similarly organized. In either case Z has $\mathtt{m} = \max\{w_{old}, w_{new}\}$ entries.

On a parallel machine possessing hypercube inter-connections processor P_k's identity, `my_adrs`, is taken to be the value k expressed in binary. P_k has a direct communication link with any processor whose address differs from P_k's in exactly one bit. If i differs from k in the jth bit, we say that P_k and P_i are neighbors in dimension j. The parallel algorithm in figure 5 implements *personalized all-to-all* communication [10],[9], and moves the new work units to their proper locations. This algorithm is optimal on a hypercube where only one communication port can be active at a time. Upon termination, the Z array of

resource allocation, ICASE report in preparation.

each processor holds the processor's new work units, sorted by geographical position. If $w_{old} > w_{new}$, then consecutive groups of w_{old}/w_{new} elements of Z are coalesced into work units with size w_{new}.

Here again, knowledge of the exchange algorithm's behavior and volume-dependent communication costs allow us model or measure the cost of changing granularity. It is interesting to note that in the algorithm above, the cost of changing granularity depends only on the level of the grid being exchanged; it does *not* depend on the work unit granularities. This fact will be exploited in the granularity scheduling algorithm.

Our model problem can be viewed as a sequence of E grid level updates, on grids $G_1, \ldots, G_E$. Recall that an update to G_i is interpreted here as the collection of updates to all contiguous grids at a common level. Although it is not *logically* necessary, we suppose that all processors synchronize at the end of a grid level update. The processors do tend to be synchronized by the boundary exchange; we will later sketch a means of relaxing this assumption. Assuming that the cost of a synchronization is constant, we need not explicitly consider this cost in the scheduling problem.

We have shown that differing work unit granularities can be used to map the grids onto a parallel processor; there are $N - p$ possible choices of granularity, which we suppose are indexed $1, 2, \ldots, N - p$. The optimal schedule of granularity changes can be determined in $O(E(N - p))$ time using dynamic programming. A small number of additional definitions facilitate derivation of this result.

Let $L(i)$ denote the level of the grid transmitted in a remapping prior to an update of grid G_i, and let F_{ij} denote the update finishing time of an update to grid G_i using the jth granularity. F_{ij} is a time required to solve equation 1 in parallel on all contiguous subgrids on grid G_i; the recursion of algorithm `update` has been unrolled in the definition of the grids $\{G_k\}$. The cost of remapping prior to updating grid G_i is $M_{L(i)}$, where we emphasize the sole dependence of this cost on the level of the grid being exchanged. Finally, let A_{ij} be the *minimum execution time* required to bring the computation from the beginning through the G_i update; this last update occurs using granularity j. Then the principle of optimality states that

$$A_{ij} = \min \begin{cases} \min\limits_{k \neq j} \{A_{(i-1)k}\} + M_{L(i)} + F_{ij} \\ A_{(i-1)j} + F_{ij} \end{cases} . \tag{2}$$

This expression simply says that before updating G_i we should either retain the present granularity, or remap from some other granularity, identified as the one minimizing $\{A_{(i-1)k}\}$. The value of A_{ij} is thus seen to be the minimum of a *remapping cost function* and a *retaining cost function*. This system of equations can be solved by noting that $A_{1j} = F_{1j}$ for all j; it is then possible to find A_{2j} for all j. When solving for A_{2j} we will record which cost function determines its value. This procedure is continued for all $A_{3j}, A_{4j}, \ldots, A_{Ej}$; then the optimal granularity schedule is readily determined. The j minimizing A_{Ej} is found, say at $j = k$. The value A_{Ek} is the minimum possible execution time under the granularity scheduling scheme, and to achieve that time G_E must be updated with granularity k. If

the value for A_{Ek} is determined in equation (2) by the remapping cost function, then before updating G_E we should have remapped from the granularity defining the remapping cost function's value. Otherwise we reach the G_E update using the same granularity as was used on G_{E-1}. The procedure for determining the remapping decision prior to the G_{E-1} update is the same—the cost function defining $A_{(E-1)k}$ defines the remapping action that should have been taken just prior to the G_{E-1} update. The optimal remapping schedule is determined by backtracking through the array of solutions to equation (2).

The time required to solve this system of equations is easily shown to be $O(E(N-p))$. Suppose that the solutions $A_{(i-1)j}$ are known for all j, and that the minimal value among these is achieved when $j = k$. To solve for any A_{ij} we need only compare $A_{(i-1)j}$ with $A_{(i-1)k} + M_{L(i)}$, which is done in constant time. As we solve for each j we can retain the minimum value of A_{ij} seen so far, again a constant time cost. The $O(E(N-p))$ complexity follows from the observation that there $E(N-p)$ function values requiring solution . The backtracking phase to determine the granularity schedule is simply $O(E)$, which is dominated by the solution phase complexity.

The overall cost of computing a schedule must include the cost of estimating the update finishing times. While this cost is quite low, it can easily dominate the cost of computing a schedule and potentially even exceed the cost of running the gridding computation. Granularity scheduling makes sense if a single grid structure will be used repeatedly (as they often are), or if the necessary information is already available, as it might be in a compiler.

A number of extensions to this approach are possible. In the face of high communication costs it may be advantageous to "contract" the computation onto a smaller number of processors when the workload drops, and "expand" again when it rises. This can be directly incorporated into the dynamic programming solution by allowing a remapping to a different sized set of processors. This type of extension has immediate application to the parallelization of multi-grid algorithms [4]. A second obvious extension is to higher dimensional domains. A third extension allows differing integration schemes on differing grid levels. A fourth extension is to reduce the number of states in the dynamic programming solution by collapsing states between which a remapping cannot occur. A fifth extension is to relax the assumption of global synchronizations between grid level updates, and directly model the data flow synchronization that is more likely to be found in the code. These considerations, along with empirical data on the performance of this method are taken up in an upcoming report [4] .

[4]D. Nicol, J. Saltz, J. Townsend, *Estimating run-time costs in parallel computations for the purposes of resource allocation*, ICASE report in preparation.

3 Tradeoffs between Load Imbalance and Pattern of Synchronizations

In the first model problem we are able to dynamically adjust granularity, and hence the trade-offs between load imbalance and communication overhead. Synchronizations, however costly, are not variable. In the second model problem this situation is reversed: we cannot easily change granularity dynamically, but we may have considerable freedom in choosing where we synchronize. As has already been illustrated, this may allow us to place synchronizations in such a way that we achieve a good balance of load. But we suffer additional costs by increasing the number of synchronizations beyond that which is minimally needed to enforce data dependencies. Instead of scheduling granularity changes, we turn now to the problem of scheduling synchronizations. As we shall see, this problem is easily treated with dynamic programming. Our approach here is directly applicable to the general class of recursion equations possessing reasonably regular local data dependencies. For clarity we shall focus on the model problem of solving triangular systems of linear equations. Pre-scheduling synchronizations is particularly important in situations where the same matrix *structure* is repeatedly used, even though the matrix *elements* change. This is exactly the case encountered in certain preconditioned conjugate gradient algorithms [18], [8], [21], and is the framework within which our methods are tested.

It is not difficult to partition the variables expressed by a triangular matrix into a sequence of sets; all variables within a given set can be solved for concurrently. We will discuss only the algorithm and results for solution of a lower triangular matrix, the situation applying in the upper triangular case is essentially identical.

The data dependencies between variables can be described by a directed acyclic graph (DAG) in the usual way. If the solution to variable a depends on the solution to variable b, then b's vertex roots a directed edge to a's vertex. Computational wavefronts are efficiently determined by sorting the DAG topologically. One stage of the sort consists of removing all vertices that are not pointed to by edges, and then removing dangling edges. All vertices removed during a given stage constitute a wavefront; the wavefronts are numbered by order of generation.

Wavefronts form the basis of a general parallel solution method discussed in [18] that is well-suited for a shared-memory machine such as the Encore Multimax[6]. There are no data dependencies between any two variables in the same wavefront; all variables within a wavefront may thus be solved concurrently, provided that all of their data dependencies have been satisfied. Solving for a variable consists of forming a linear combination of solutions to other variables; the coefficients are specified by the triangular matrix, and the other solutions are available in the shared memory. Every wavefront constitutes a phase, and the problem's data dependencies can be enforced by inter-phase synchronization. As discussed in [18] this method can be generalized: "closely coupled" variables are first systematically aggregated into work units where all variables in a work unit can be solved

for (in some specified order) by a single processor without interaction with other processors. The dependencies between work units induce a *work unit dependency* DAG and associated wavefronts.

3.1 Problem Partitioning

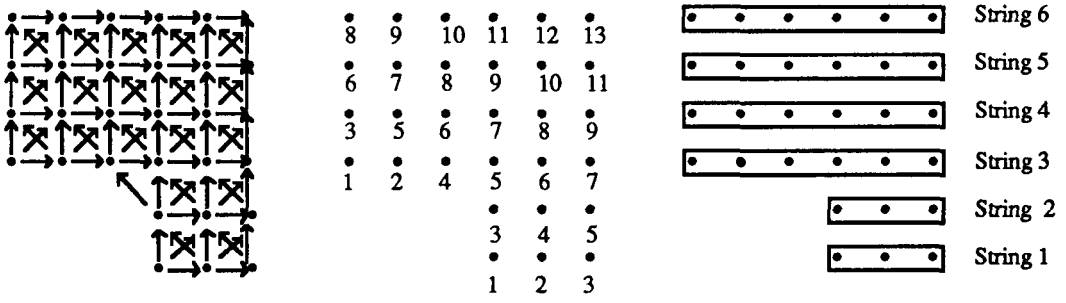
Wavefronts identify work that can be done in parallel; assignment of that work to processors is aided by the concept of a *string*. Loosely speaking, a string is a path through the work unit dependency DAG, and so is roughly orthogonal to wavefronts. Methods described in [18] assign workload by wrapping strings onto processors. In this paper we use strings primarily as a guide to the independence of work units in the work unit dependency DAG. A method for partitioning the DAG into strings is discussed below.

Let D be a DAG, and let G be a set of vertices in D without predecessors. Choose some $s_0 \in G$ as a string *start* vertex. For our model problem D is the work unit dependency DAG, and G holds a single vertex. Given a partial string $s_0, \ldots, s_i$ we remove from D all edges rooted in s_i. If this removal creates one or more vertices without incoming edges we choose one as s_{i+1} and place the rest in G. Otherwise, we terminate this string and begin another with a vertex in G. Strings are enumerated by order of creation. If a string is extended by a vertex v, then all vertices depending directly on v are marked. When choosing s_{i+1} from among a set of possibilities, preference is given to vertices previously marked by this process. Depending on the problem underlying the DAG, sophisticated priority schemes for string extension can control string construction in desirable ways. Such topics are outside the scheduling concerns of this paper, but are studied in [18].

The partitioning of a DAG into strings induces another DAG whose vertices represent strings. This graph is called a *string DAG* where vertex S_i roots an edge into S_j if string S_j has at least one variable that depends on some variable in S_i. To better illustrate these ideas, figure 6(a) depicts a DAG which could be obtained from a zero fill incomplete factorization of a matrix arising from the discretization of an elliptic partial differential equation using a nine point star template. Figure 6(b) enumerates the wavefronts in the computation, and figure 6(c) illustrates a string decomposition of the DAG.

3.2 Load Balancing

In linear systems arising from the solution of partial differential equations, it is common for the string DAG to form a strictly ordered chain[18]. A natural way of load balancing is to wrap entire strings onto processors, just as domain regions were wrapped onto processors in the first model problem. Synchronization between wavefronts enforces data dependencies. However, this greedy technique of scheduling an entire wavefront sometimes fails to balance the load. The failure to balance is essentially an end-effect; e.g., the phase has $P+1$ work units with equal computational demands, but only P processors are available. We can correct this problem by noting that any given work unit in a wavefront does not depend



(a) Work Unit Dependency DAG (b) Computational Wavefronts (c) String Decomposition

Figure 6: Dependency DAG, Wavefronts, and Strings

on all work units in the previous wavefront. It is possible to concurrently compute work units in consecutive wavefronts provided that we carefully observe data dependencies. The identification of strings helps in this task.

Designate each work unit by (w, i) where w is the work unit's wavefront number, and i is its string number. We can define a strict ordering on these two tuples in the following manner: $(a, c) > (b, d)$ when $a > b$ or $a = b$ and $c > d$. It follows by construction that for any a and b, all work units greater than (a, b) and less than $(a + 1, b)$ can be solved for concurrently. This fact allows us to schedule work units from different wavefronts in the same phase, and allows us a considerable degree of freedom in choosing synchronization points.

The computational phases can be selected using dynamic programming. A small number of definitions facilitate this result. Let the W work units be renumbered $1, \ldots, W$ with respect to the ordering above, and let the execution time of work unit n be denoted w_n. For any work unit n, let $P(n)$ denote the furthest "previous" work unit that we allow to be concurrently evaluated; if n's tuple is (a, b) then $P(n)$'s tuple is greater than or equal to $(a-1, b+1)$ (it may not be equal to $(a-1, b+1)$ because string $b+1$ may have no work unit on wavefront $a-1$). Similarly, let $F(n)$ denote the furthest "future" work unit that we allow to be concurrently evaluated with n; $F(n)$ is less than or equal to $(a+1, b-1)$. Let S be the delay cost of synchronizing. Finally, let $wmax(i, j)$ be the time required to evaluate work units i through j in parallel, provided that $P(j) \leq i \leq j$. $wmax(i, j)$ obviously depends on the assignment strategy used to distribute the workload, and is equal to the time required by the most heavily load processor. If we place no constraints on

the assignment method or variable solution costs, the problem of determining an optimal mapping of i to j on P machines is a well-known NP-complete scheduling problem. Lower complexities are achieved if we view the processors as a chain, and constrain the assignment by requiring that for every k, work units k and $k+1$ are either assigned to same processor, or two adjacent processors. In this case the best assignment is determined in polynomial time [3]. The necessity of quickly determining a solution leads us to adopt the simple wrapping scheme previously described. In this case work unit k is assigned to processor $(k-i) \bmod P$, and $wmax(i,j)$ is given by

$$wmax(i,j) = \max_{0 \le p \le P-1} \{ \sum_{k=0}^{\lfloor (j-i-p)/P \rfloor} w_{i+p+kP} \}.$$

Define $A(n)$ to be the minimum time possible to bring the computation through the solution of n, given that n is the largest indexed work unit in its phase. The principle of optimality then states that

$$A(n) = \min_{P(n) \le k \le n} \{A(k) + wmax(k,n) + S\}. \tag{3}$$

For any given n, the k which defines $A(n)$ by minimizing the right hand side of equation (3) also defines the workload to be done in the phase containing n's solution.[5] Once we solve these equations the synchronization schedule is found by backtracking through the solutions: the k_1 defining $A(W)$ defines the last phase, the k_2 defining $A(k_1)$ defines the phase prior to the last, and so on.

The function $wmax(k,n)$ can be determined in $O(PTW)$ time where T is the number of strings. In many of the problems we have studied T is approximately equal to a small constant times $\sqrt{W}$. Once all values for $wmax(k,n)$ are determined the dynamic programming solution can be computed in $O(TW)$ time. Like the first model problem, the complexity of this scheduling algorithm is dominated by the time required to pre-compute the run-time costs under differing scenarios. The fine granularity of the units of work to be scheduled make this cost estimate prohibitive in many cases, although the solution method could be attractive in the common situation where the same triangular system structure is repeatedly used to solve many different sets of equations. The use of a relatively expensive dynamic programming algorithm for such fine grained work scheduling is somewhat inappropriate; in an actual multiprocessor, the computation costs and the synchronization costs can only be roughly characterized. There is therefore reason to believe that it may be possible to find an inexpensive policy that leads to performance that is comparable to the dynamic programming method.

The policy we propose bears a very close relationship to the SAR (Stop at Rise) policy proposed in [13] for choosing remapping points. The guiding principle for our model

[5] The equation for $A(W)$ need not contain the S term, although inclusion of S does not affect the optimal schedule.

problem is to define a phase by minimizing the ratio of the average time a processor is idle due to load imbalance and synchronization to the average time it spends doing useful computation. A more general exposition of this principle will be found in a forth-coming technical report [6].

We continue to use the linear variable ordering, and let w_i denote the estimated computation time required by the ith work unit. This computation time is easily estimated, as the cost of each variable solution is directly related to the number of nonzero columns in the variable's matrix row. Like the dynamic programming solution, we assume here that work units within a phase are assigned to processors by wrapping.

The ratio of a processor's average idle time in a phase computing work units i through $i+n$ to its average computation time is

$$W_i(n) = \frac{S + wmax(i, i+n) - (1/P)\sum_{j=i}^{i+n} w_j}{(1/P)\sum_{j=i}^{i+n} w_j}.$$

A simple version of our heuristic schedules work in a greedy manner, choosing the phase size n that minimizes $W_i(n)$. To schedule the first phase, the n_1 minimizing $W_1(n)$ is found, and the first phase solves work units 1 through n_1. To find the second phase, the n_2 minimizing $W_{n_1}(n)$ is found, and so on. Our experimental results indicate that this simple version of the heuristic can be very short sighted, in that the scheduling decision entirely neglects the decision's effect on the performance of subsequent phases. Under some circumstances a phase is ended in a manner that leaves the subsequent phase with little work to perform.

This myopic difficulty is alleviated by attempting to ensure that the amount of work available to the following phase is relatively large compared to the synchronization cost. This is accomplished by adding a term expressing the ratio of the synchronization cost to the maximum possible average processor execution time in the next phase. Recalling that $F(n)$ is the largest index of a variable that can be concurrently evaluated with variable n, we redefine $W_i(n)$ as

$$W_i(n) = \frac{S + wmax(i, i+n) - (1/P)\sum_{j=i}^{i+n} w_j}{(1/P)\sum_{j=i}^{i+n} w_j} + \frac{S}{(1/P)\sum_{j=n+1}^{n+F(n)} w_j}.$$

In the event that $F(i+n) > P(W)$, a phase ending at n leaves at most one phase left. In this case there is no advantage in forcing substantial work onto the last phase; we consequently drop the additional term.

Note that the expressions are written in the form above for conceptual clarity, obvious algebraic simplifications yield simpler expressions with the same minima.

The computational savings achieved by this algorithm over the dynamic programming solution occur chiefly because only select values of $wmax(i, j)$ need to be computed. There

[6] J. Saltz, D. Nicol, *Performance Studies on Mapping Methods for Irregular Parallel Computations*, ICASE report in preparation.

is a $O(W+P)$ startup cost to compute all processor load prefixes $\sum_{j=0}^{k} w_{p+jP}$ for $0 \leq p \leq P-1$ and $k = 0, \ldots, W/P$. Following this, any value for $wmax(i,j)$ can be computed in $O(P)$ time. The time required by the heuristic is $O(P)$ times the number of times the $W(n)$ statistic is calculated. We will show that the expected number of such calculations is $O(W)$, making the heuristic's expected complexity $O(PW)$.

Suppose that work unit i begins phase j. $W_i(n)$ is calculated for all $1 \leq n \leq (F(i)-i)$; let $X_j = F(i)-i$, and note that X_j may vary as a function of i. If there are Z phases chosen by the heuristic, then the $W(n)$ statistic is computed $\sum_{j=0}^{Z} X_j$ times. Now let Y_j be the number of work units that the heuristic places in the jth phase; $Y_j = n_0$ if the jth phase begins with work unit i and among all $n \in \{1, 2, \ldots, F(i)-i\}$, $W_i(n_0)$ is minimal. We now consider X_j, Y_j, and Z to be random variables, hopelessly correlated, with arbitrary distributions. The expected linear-time complexity of our heuristic rests squarely on the following assumption: there is a constant c such that for every j, $E[Y_j] \geq E[X_j]/c$. For example, if every work unit among the X_j possible has an equal chance of minimizing $W_i(n)$, then $c = 2$. In practice we expect c to be smaller than two, because Y_j tends to be close to X_j.

The following equation must always hold:

$$Y_1 + Y_2 + \ldots + Y_Z = W.$$

By the linearity of the expectation operator we thus have

$$E[Y_1] + E[Y_2] + \ldots + E[Y_Z] = W.$$

But for every j, $E[Y_j] \geq E[X_j]/c$; this leads to the inequality

$$cW \geq E[\sum_{j=1}^{Z} X_j]$$

where we recognize the right hand side as being the expected number of times a $W(n)$ statistic is computed. This number is $O(W)$, making the heuristic's complexity $O(PW)$ which is linear in the problem size for a fixed architecture.

3.3 Experiments

We evaluated the performance of our scheduling algorithms on a shared-memory architecture. The Encore Multimax is a bus based shared memory machine that utilizes 10 MHz NS32032 processors and NS32081 floating point coprocessors. All tests reported were performed on a configuration with 18 processors and 16 Mbytes memory at times when the only active processes were due to the authors and to the operating system. On the Encore the user has no direct control over processor allocation. Tests were performed by spawning a fixed number of processes and keeping the processes in existence for the length of each computation. The processes spawned are scheduled by the operating system; throughout

the following discussions we make the tacit assumption that there is a processor available at all times to execute each process. In order to reduce the effect of system overhead on our timings, tests were performed using no more than 14 processes; this left four processors available to handle the intermittent resource demands presented by processes generated by the operating system.

In Krylov space algorithms such as the preconditioned conjugate gradient algorithm, very sparse triangular systems with identical matrices are repeatedly solved. The triangular matrices are formed through a process of incomplete factorization[5]. Triangular matrices with varying degrees of sparsity may be created by this process. The degree of sparsity is controlled by determining by one of a variety of mechanisms which matrix elements are allowed to fill in or become non-zero. There is frequently a numerical advantage in allowing a moderate number of matrix elements to fill in. The triangular matrices thus created are still quite sparse but have only a limited number of variables in a wavefront. Methods for improving the performance of these relatively difficult to parallelize problems are consequently of practical interest.

In the example to be discussed below, a matrix M was formed describing a problem discretized using a 9 point star template on a 63 by 63 point domain. M was then factored using an incomplete factorization algorithm which allowed a moderate degree of fill (level 2 fill). The typical row of the resulting triangular matrix contains seven non zero entries. Experiments reported in [18] suggest that on the rather fine grained Encore Multimax machine used here, the work units for this problem should be as small as possible—the substitution of a single variable. Using this granularity there are 311 computational wavefronts. The triangular system was solved using 14 processors. Every measured speedup number we present was obtained by averaging the timings from 25 runs; the variance between runs was extremely small. The serial timing used in the speedup calculation is taken from a separate, specifically sequential triangular solve code.

The cost of a synchronization on the Multimax is roughly 4 times the cost of a floating point add and multiply. We tested the sensitivity of our heuristic to inaccurate estimations of this cost by varying the values of S used in computing $W(n)$ between 0.01 and 100 times the cost of a floating point add and multiply. Table 1 lists the experimentally observed speedups achieved by the different schedules produced as S varies. For the purposes of comparison table 1 also gives speedup estimates that would result from the absence of any inefficiencies other than load imbalance. These symbolically estimated optimal speedups are obtained by considering only floating point calculations, assuming that an add and a multiply each have unit cost. The same processor assignments are used in calculating the symbolically estimated speedups as are used in the computed problems. Table 1 also depicts the number of computational phases created by the scheduling heuristic.

We see that the $W(n)$ policy is rather insensitive to the precise estimate supplied for synchronization cost. With decreasing estimated synchronization costs, we note a moderate increase in measured speedup, a more substantial improvement in symbolically estimated speedups, and a slight increase in the number of phases required to solve the problem. It

Synchronization Cost	*Est. Optimal Speedup*	*Number of Phases*	*Experimental Speedup*
Wavefront Scheduling	8.01	311	6.67
100	8.05	312	6.68
50	9.26	321	7.29
10	11.52	336	8.22
1	11.53	336	8.26
0.1	11.53	336	8.26
0.01	11.53	336	8.26

Table 1: Experimental Results for 63 × 63 Problem Using 9-Point Star Stencil

is notable that the schedules obtained using synchronization costs of 1, 0.1 and 0.01 were identical, only one set of multiprocessor timings were obtained for these cases. In addition to the results obtained using $W(n)$, we also ran the problem configured so that all points in a wavefront would constitute a phase. The results obtained were, as expected, very similar to the results obtained using $W(n)$ with high synchronization costs.

To provide an estimate of the relative performance of the dynamic programming scheduling policy, we solved the same problem described above again using 14 processors, using this more expensive scheduling method. In table 2 we compare the experimental speedups observed when using the $W(n)$ and dynamic programming policies. We also compare the predicted speedups obtained through the use of the two policies. These predicted speedups are computed symbolically and, unlike the estimated optimal speedups presented above, take into account the estimated synchronization costs. It is notable that in table 2 the differences in both estimated and actual performance between the $W(n)$ heuristic and the dynamic programming solution are minimal. It is worth noting that the relative superiority of the dynamic programming solution depends on the availability of accurate cost estimates. For a miss-estimated value of S equal to 100, both the estimated optimal speedup in the absence of synchronization costs and the experimental speedup are slightly higher for the $W(n)$ than for the dynamic programming policy. [7] The schedules produced using dynamic programming were identical for synchronization costs of 10, 1, 0.1, 0.01; only one set of multiprocessor timings were obtained these cases. Table 2 clearly shows that the additional complexity of the dynamic programming approach is not warranted—

[7] The estimated optimal speedup using dynamic programming for scheduling is 8.01 while the estimated optimal speedup is 8.05 when the $W(n)$ policy is used.

Synchronization Cost	*Predicted W(n) Speedup*	*Predicted D.P. Speedup*	*Experimental W(n) Speedup*	*Experimental D.P. Speedup*
100	0.872	0.874	6.68	6.50
50	1.576	1.579	7.29	7.38
10	5.077	5.089	8.22	8.26
1	10.229	10.240	8.26	8.26
0.1	11.384	11.393	8.26	8.26
0.01	11.528	11.537	8.26	8.26

Table 2: Experimental Results for 63×63 Problem Using 9-Point Star Stencil

the $W(n)$ heuristic achieves very nearly optimal performance (relative to the wrapping mapping paradigm).

We can artificially increase the cost of synchronization by increasing the number of barriers that processors pass through at a synchronization. Increasing the synchronization cost allows us to to predict performance on machines with less favorable computation to synchronization ratios than the Encore Multimax. Table 3 compares the performance of wavefront scheduling and $W(n)$ scheduling as the number of barriers at a synchronization varies between 1 and 4. Since the $W(n)$ heuristic is insensitive to most values of S and we do not wish to burden the user with estimations of synchronization costs, all experiments reported in table 3 used $S = 0.1$. Table 3 clearly shows that our heuristic improves performance by accepting more synchronization costs in order to improve load balance. Naturally, it becomes harder to do so in the face of high synchronization costs, but in the range of costs studied $W(n)$ effectively managed the load imbalance/synchronization cost tradeoff. We see then that using a scheduling heuristic gives a non-trivial performance improvement over the static method in [18].

The relative performance of the $W(n)$ policy and the wavefront scheduling policy were examined for other matrices of practical interest; in each case the $W(n)$ policy outperformed by varying degrees, the policy that schedules work by wavefronts. This effect was most noticeable in cases in which there are relatively few units of work in a phase.

4 Summary

Recent research has investigated a very regular and parameterized approach to mapping dynamically changing problems onto parallel architectures. The use of *mapping parameters*

Number of Barriers	*Experimental Speedup*		*% Time in Synchronization*	
	W(n)	*Wavefront*	*W(n)*	*Wavefront*
1	8.28	6.69	23.6	18.8
2	6.72	5.74	39.1	30.1
3	5.74	4.93	46.3	34.8
4	4.94	4.44	52.4	43.8

Table 3: Results For Increasing Synchronization Costs

is a key concept in this approach, as it gives a simple handle on controlling load imbalance/communication cost trade-offs. Observation of problem behavior on parallel machines suggests that by dynamically adjusting mapping parameters we can substantially improve performance. We have explored this idea in the context of a dynamic one-dimensional partial differential equation solution method for message-passing machines, and the solution of sparse triangular systems of linear equations on shared-memory machines. We make the important assumption that the computational activity is known in advance, but note that this is often true for the codes we consider. We then formulated dynamic programming solutions for parameter schedules, and showed that such schedules are determined with low complexity. Even so, it can be advantageous to use faster scheduling heuristics; we present an expected-linear-time scheduling heuristic for the triangular systems solution method and give empirical evidence that its performance is virtually identical to that of the dynamic programming solution. The concepts and techniques we describe have application in a wide variety of problems.

Acknowledgements: Thanks are due to Dan Reed who suggested applying granularity scheduling to an irregular grid problem, to Doug Baxter who supplied the matrices used in the triangular solves and to Merrell Patrick who also encouraged this research. As always, we thank Bob Voigt for his support.

References

[1] BERGER, M., AND BOKHARI, S. H. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Trans. on Computers C-36*, 5 (May 1987), 570–580.

[2] BERGER, M. J., AND OLIGER, J. Adaptive mesh refinement for hyperbolic partial differential equations. *J. Comp. Phys. 53* (1984), 484–512.

[3] BOKHARI, S. H. *Partitioning Problems in Parallel, Pipelined, and Distributed Computing.* Tech. Rep. 85-54, ICASE, November 1985.

[4] BRANDT, A. Multigrid technique: 1984 guide. In *GMD Studien 85, Gesellschaft fur Mathematik und Datenverabeitung* (St. Augustin, 1984).

[5] ELMAN, H. *Iterative Methods for Large Sparse Nonsmmetric Systems of Linear Equations.* Department of Computer Science YALEU/DCS/TR-229, Yale University, April 1982.

[6] *Multimax Technical Survey.* Tech. Rep. 726-01759 Rev A, Encore Computer Corporation, 1986.

[7] FOX, G., AND OTTO, S. W. *Concurrent Computation and the Theory of Complex Systems.* Tech. Rep. CALT-68-1343, Caltech Concurrent Computation Program, 1986.

[8] GREENBAUM, A. *Solving Sparse Triangular Linear Systems Using Fortran with Parallel Extensions on the NYU Ultracomputer Prototype.* Report 99, NYU Ultracomputer Note, April 1986.

[9] HO, C., AND JOHNSSON, S. Algorithms for matrix transposition on boolean n-cube configured ensemble architectures. In *Proceedings of the 1987 International Conference on Parallel Processing* (St. Charles, Illinois, August 1987), pp. 621–629.

[10] JOHNSSON, S. *Data Permutations and Basic Linear Algebra Computations on Ensemble Architectures.* Tech. Rep. TR-367, Yale University Department of Computer Science, February 1985.

[11] NICOL, D. Mapping a battlefield simulation onto parallel message-passing architectures. In *Proceedings of the 1988 SCS Conference on Distributed Simulation* (February 1988). To appear.

[12] NICOL, D., AND REYNOLDS, JR., P. *Optimal Dynamic Remapping of Parallel Computations.* Tech. Rep. 87-49, ICASE, July 1987.

[13] NICOL, D., AND SALTZ, J. *Dynamic Remapping of Parallel Computations with Varying Resource Demands.* Tech. Rep. 86-45, ICASE, July 1986. To appear in *IEEE Transactions on Computers.*

[14] NICOL, D., AND SALTZ, J. *Principles for Problem Aggregation and Assignment in Medium Scale Multiprocessors.* Tech. Rep. 87-39, ICASE, July 1987.

[15] NICOL, D. M., AND WILLARD, F. H. Problem size, parallel architecture, and optimal speedup. *Journal of Parallel and Distributed Computing.* To appear.

[16] PRESS, W., FLANNERY, B., TEUKOLSKY, S., AND VETTERING, W. *Numerical Recipes.* Cambridge University Press, New York, 1986.

[17] REED, D. A., ADAMS, L. M., AND PATRICK, M. L. Stencils and problem partitionings: their influence on the performance of multiple processor systems. *IEEE Trans. on Computers C-36*, 7 (July 1987), 845–858.

[18] SALTZ, J. *Automated Problem Scheduling and Reduction of Communication Delay Effects.* Report 87-22, ICASE, May 1987.

[19] SALTZ, J., AND CHEN, M. Automated problem mapping: the crystal runtime system. In *The Proceedings of the Hypercube Microprocessors Conf., Knoxville, TN* (September 1986).

[20] WON, Y., AND SAHNI, S. Maze routing on a hypercube multiprocessor computer. In *Proceedings of the 1987 International Conference on Parallel Processing* (St. Charles, Illinois, August 1987), pp. 630–637.

[21] Y. SAAD, M. S. *Parallel Implementations of Preconditioned Conjugate Gradient Methods.* Department of Computer Science YALEU/DCS/TR-425, Yale University, October 1985.

Report Documentation Page

National Aeronautics and Space Administration

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA CR-178378 ICASE Report No. 87-52		

4. Title and Subtitle	5. Report Date
SCHEDULES FOR MAPPING IRREGULAR PARALLEL COMPUTATIONS	September 1987
	6. Performing Organization Code

7. Author(s)	8. Performing Organization Report No.
David M. Nicol and Joel H. Saltz	87-52
	10. Work Unit No. 505-90-21-01

9. Performing Organization Name and Address	11. Contract or Grant No.
Institute for Computer Applications in Science and Engineering Mail Stop 132C, NASA Langley Research Center Hampton, VA 23665-5225	NAS1-18107
	13. Type of Report and Period Covered: Contractor Report

12. Sponsoring Agency Name and Address	14. Sponsoring Agency Code
National Aeronautics and Space Administration Langley Research Center Hampton, VA 23665-5225	

15. Supplementary Notes

Langley Technical Monitor: Richard W. Barnwell

Submitted to 1988 SIGMETRICS Conference

16. Abstract A large class of scientific computational problems can be characterized as a sequence of steps where a significant amount of computation occurs each step, but the work performed at each step is not necessarily identical. Two good examples of this type of computation are (1) regridding methods which change the problem discretization during the course of the computation, and (2) methods for solving sparse triangular systems of linear equations. Recent work has investigated a means of mapping such computations onto parallel processors; the method defines a family of static mappings with differing degrees of importance placed on the conflicting goals of good load balance and low communication/synchronization overhead. The performance tradeoffs are controllable by adjusting the parameters of the mapping method. To achieve good performance it may be necessary to dynamically change these parameters at run-time, but such changes can impose addititonal costs. If the computation's behavior can be determined prior to its execution, it can be possible to construct an optimal parameter schedule using a low-order-polynomial-time dynamic programming algorithm. We illustrate this on two problems. Because the dynamic programming algorithms can be too expensive, we study the performance of an expected linear-time scheduling heuristic on one of the model problems and show that it is effective, and nearly optimal. The concepts we discuss here are quite general, and apply to a wide variety of situations.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
parallel processing, scheduling, multiprocessors, performance evaluation	61 - Computer Programming and Software 64 - Numerical Analysis Unclassified - unlimited

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of pages	22. Price
Unclassified	Unclassified	25	A02