NASA Contractor Report ~~181756~~

ICASE REPORT NO. 88-58

ICASE

ACCURATE MODELING OF PARALLEL SCIENTIFIC COMPUTATIONS

David M. Nicol

James C. Townsend

Contract Nos. NAS1-18107, NAS1-18605, and AFOSR 88-0117
November 1988

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

1 Introduction

Scientific codes are usually parallelized by partitioning a grid among processors. Irregular grids (see figure 1) must be partitioned carefully in order to balance the workload. Solution methods for time-varying problems often change the grid as the computation progresses; furthermore, the grid structure is not generally manifest until *load-time*, when the code reads the problem from a file. These characteristics make parallelization nontrivial. Our ability to map such problems well depends on our ability to analytically model the computation's execution time and compare different candidate partitions.

This paper outlines a framework for treating these problems. We illustrate the method on a fluid flow computation which periodically redefines its grid, giving rise to a sequence of grid structures. The computation's execution time is modeled analytically as a function of grid placement, partition, and the computation's logical structure. Model constants are estimated by measuring the execution time of critical code fragments. We use the model to construct partition schedules, and show that the model accurately predicts performance—on large problems the model is within 3% of measured performance. The modeling approach is not dependent on numerical particulars of the fluid's problem. Similar approaches on other codes should produce good results as well.

2 Problem Particulars

We illustrate the method using a numerical solution of a wave equation:

$$\frac{\partial u}{\partial t} = -u\frac{\partial u}{\partial x}. \tag{1}$$

$u(x,t)$ represents fluid density at position x and time t; equation (1) is an idealized law governing the change in fluid density as a function of position and time. We numerically solve for u over $[0,1]$ by imposing a three-tiered hierarchy of grids on $[0,1]$. A grid at level i (i = 0,1, or 2) has points spaced uniformly $\Delta h/2^i$ apart. As shown in figure 1, a single grid at level 0 spans $[0,1]$; a multiplicity of grids may exist at levels 1 and 2, provided that grids at a shared level do not overlap, and each level 2 grid completely overlaps a level 1 grid. We will refer to a hierarchical collection of such grids as a *grid structure*, to be distinguished from a single grid at some level. The higher level grids help to resolve rapidly changing features in the solution. These features tend to move as a function of time. Consequently, the size and position of the grids are typically changed periodically as the equation is integrated in t. We will denote the resulting sequence of grid structures by $G_1, \ldots, G_M$. The computation begins by integrating over grid structure G_1; some number of time-steps later G_1 is replaced by G_2, which in turn is later replaced by G_3, and so on.

During a single time-step the solution procedure revolves around a routine, `integrate(G, lvl, t, Δt)`. `G` here is some contiguous grid at level `lvl`; equation (1) is integrated between `t` and `t+Δt`. The

Accurate Modeling of Parallel Scientific Computations

David M. Nicol *
Department of Computer Science
College of William and Mary
Williamsburg, VA 23185

James C. Townsend
NASA Langley Research Center
Hampton, VA 23665

Abstract

Scientific codes are usually parallelized by partitioning a grid among processors. To achieve top performance it is necessary to partition the grid so as to balance workload and minimize communication/synchronization costs. This problem is particularly acute when the grid is irregular, changes over the course of the computation, and is not known until load-time. Critical mapping and remapping decisions rest on our ability to accurately predict performance, given a description of a grid and its partition. This paper discusses one approach to this problem, and illustrates its use on a one-dimensional fluids code. The models we construct are shown empirically to be accurate, and are used to find optimal remapping schedules.

*This research was supported by the National Aeronautics and Space Administration under NASA Contract Nos. NAS1-18107 and NAS1-18605 and by the Air Force Office of Scientific Research under Contract No. AFOSR 88-0117 while the author was in residence at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA 23665.

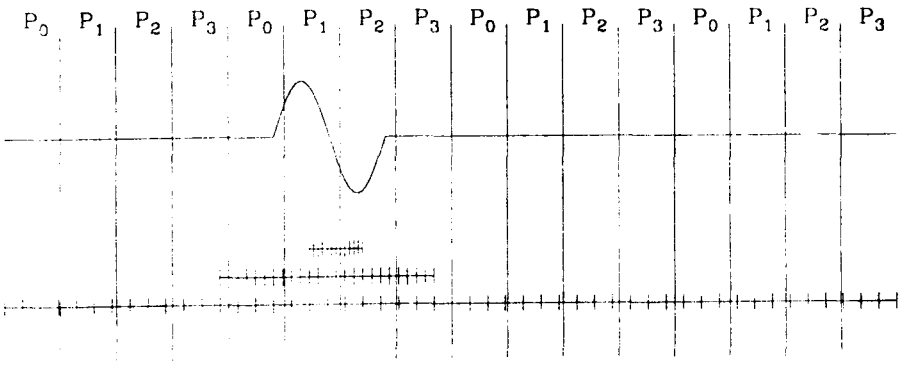


Figure 1: Three-tiered grid hierarchy for one-dimensional problem. The grid is partitioned using a scatter decomposition which assigns four subintervals to each of four processors.

synthesized treatment of all grid levels is carried out recursively by routine **update**, shown as figure 2. To carry the solution ahead one *time-step* from t to $t + \Delta t$ we call **update** with **G** being the entire level 0 grid. Each level 1 grid is integrated twice, each level 2 grid four times. The computational workload is consequently intense in finely gridded regions.

integrate computes $u(x, t + \Delta t)$ as a function of points close to x at time t. Consequently, parallelization through domain partitioning requires inter-processor communication. If $u(x_1, t + \Delta t)$ is computed by processor P_i and depends on a value $u(x_0, t)$ computed by P_j, then P_i and P_j must communicate. In our solution a partition is semi-static, meaning that for each time-step a processor carries out computations in the same regions of the domain, until the partition is explicitly changed. This paradigm is used with distributed memory architectures, and with shared memory architectures having a significant access cost differential between a processor's local memory and the shared memory.[1] The grid structure is distributed throughout the processor's local memories, with the shared memory being used only to exchange data lying at partition boundaries.

[1] While most shared memory architectures fall into this class, the Flex/32 on which we performed our experiments does not. This anomaly is due more to the *delay* in a local access than it is to fast global access.

```
update(G, lvl, t, Δt) {
    integrate(G, lvl, t, Δt);
    if there are any level lvl+1 grids above G
       For all level lvl+1 grids G' above G {
          update(G',lvl+1,t,(Δt)/2);
          update(G',lvl+1,t+(Δt)/2, (Δt)/2);
       }
    }
```

Figure 2: Update function

The requirement that we partition at load-time and remap at run-time prohibits the use of computationally expensive partitioning algorithms. We consequently consider only two classes of partitions, *binary partitions*[2], and *scatter decompositions*[4]. Binary partitioning explicitly balances workload by examining the grid structure and estimating the workload. A first "cut" finds the domain point where the sum of workload to the left equals (approximately) the sum of workload to the right. The resulting intervals are themselves split using the same balancing rule; the algorithm proceeds recursively, creating 2^n intervals with approximately equal workload. Typically, one uses binary partitioning to create exactly as many intervals as there are processors. By contrast, scatter decomposition divides the domain into a large of number small, equi-length subintervals, and assigns an identical number of subintervals to each processor. Scatter decomposition balances workload statistically, and is less sensitive to dynamic changes in the workload. A grid structure can be partitioned using one of a variety of scatter decompositions which differ in the length of the subintervals.

For any given grid structure we have a number of partitioning options. To optimize performance we must be able to predict the computation's execution time, as a function of grid structure and partition. This ability is needed particularly when making remapping decisions in response to a change in the grid structure—the tradeoffs between remapping costs and benefits can only be made if performance is accurately predicted. The section to follow outlines our approach to such modeling.

All of our experiments use the Flex/32 multiprocessor[5] at the NASA Langley Research Center. The Flex/32 has eighteen processors available for parallel processing, a 4Mb shared memory, 2Mb local memory, and a small instruction cache for each processor.

```
loop {
        barrier;
        CopyInBoundary();
        update(G,0,t,Δt);
        CollateSolution();
        barrier;
        CopyOutBoundary();
        t = t+Δt;
        barrier;
}
```

Figure 3: High-level view of time-step loop

3 Analytic Model

The analytic model exploits the fact that scientific codes spend the bulk of their processing time in loops, and that their synchronization patterns are regular. The model is loosely based on approaches outlined in [6,9]. Our approach is to identify the critical loops, and model each loop's execution time parametrically. Model coefficients are estimated by measuring the loops' execution times. A processor's execution time (between synchronizations) on a given grid structure, under a given partition can be estimated by calculating the appropriate loop model parameters (e.g. number of loop iterations), and applying the performance model. *System* time between two synchronization points is computed as the maximum processor time between those points, plus a synchronization cost.

We now illustrate this approach by application to the fluids problem.

Figure 3 shows the sequence of routines called by a processor during one time-step. `CopyInBoundary()` copies from shared memory to local memory all the partition boundary data needed by the processor. The volume of data copied is directly proportional to the number of subintervals assigned to that processor. `update` we have seen before. It calls a multiprocessor version of `integrate` that is careful to integrate only those points within the processor's purview. `CollateSolution()` readies the processor for another time-step by copying (local to local) the "new" solution (at time $t + \Delta t$) in the space reserved for the "old" solution (at time t). `CopyOutBoundary()` writes to shared memory all partition boundary data needed by other processors. Calls to `barrier` synchronize the processors globally. The juxtaposition of barrier calls at the top and bottom of this sequence allows a control process to perform a minor bit of necessarily serial

book-keeping.

`CopyInBoundary()` and `CopyOutBoundary()` both loop through a processor's set of subintervals and copy nine floating point numbers for each. Our expectation that loop execution time is proportional to the number of subintervals is borne out with measurements. Flex/32 timers have a resolution of 20 msecs. Loop execution times were estimated by first isolating the loop's code fragment in a separate, single-processor calibration program. Within that program we measure the time required to execute a large number of loop instances; our estimate (36 μ-secs per point) is taken by averaging. By *measuring* loop execution times we accurately capture the compiler-dependent run-time overheads of array indexing and loop bounds checking.

`CollateSolution()`'s execution time is found in the same fashion. The per-point cost of a local to local copy is estimated at 27.9 μ-secs. Timings for `barrier` calls are found similarly, with the exception that all processors execute the calibration program. A barrier synchronization involving sixteen processors requires 1.17 msecs.

`integrate()` numerically integrates a single contiguous grid within a subregion. It first sets up some variables used within its loop, and then executes the loop. `integrate()`'s execution time is consequently modeled as a linear function of the number of points being integrated. The slope and intercept of this function were estimated with a least-squares fit over a wide range of measurements; the fit was found to be tight. The loop startup cost (intercept) was determined to be 77 μ-secs, and the per-point cost (slope) was 153.9 μ-secs. The integration method is a variant of the Lax method[8], and requires ten floating point operations per point.

`update()`'s execution time depends on the number and placement of grids within the processor's subdomains. A grid that crosses a partition boundary is viewed by `integrate()` as terminating at that boundary. To estimate `update()`'s execution time under a given partition we need only identify the numbers and lengths of grids passed to `integrate()`, and use its timing model. This process must note the recursive nature of `update()` calls, and weight `integrate()` timings according to the number of times a grid is updated in a single time-step.

Between the top and bottom barriers in figure 3 a control process may change the grid structure and determine whether to remap. This serial cost is 1.4 msecs. The control process supports a decision to remap by rebuilding a number of data structures. The amount of work is proportional to the number of subdomains in the partition (remember that scatter decompositions have more subdomains than processors): 95 μ-secs per subdomain.[2] A decision to remap is flagged in the shared memory, where the new partition is also specified. Upon release from the top barrier a processor checks the remapping flag (a test not shown in the figure). If a new mapping is specified, each processor executes the sequence below:

[2]Our implementation computes this partition at load-time, as will be described in the next section. If the new partition is calculated dynamically, an additional cost should be added here.

```
CopyOutOldSubintervals();
CopyInNewPartition();
barrier;
CopyInNewSubintervals();
barrier;
```

The first copy routine simply dumps into shared memory all of the values it is responsible for under the old partition. The second copy routine reads the processor's new partition, at a cost of 106 μ-secs per assigned subinterval. The third copy routine reads from shared memory the processor's new set of values. The first and third copy routines require 30.1 μ-secs for each point copied.

The overall execution time is built up from individual processor timings. The *system* execution time between two given barrier synchronizations is the maximum time taken by any processor between those barriers, plus the barrier costs. Given the processor timing models and knowledge of synchronization behavior, we are able to predict the time required by the system to perform a time-step. This quantity is multiplied by the number of time-steps the defining grid structure and partition are used; additional costs are included wherever a remapping is applied. The following section describes how we use the model to schedule problem remappings, and to evaluate remapping heuristics.

4 Remapping Schedules

We now apply the analytic model to the problem of deciding when and how to remap. In an experimental environment, one run of a production code often looks much like another, differing only by a small perturbation of some parameter. It is sometimes reasonable then to assume that an entire run's sequence of grid structures is known at load-time. Under these circumstances we can determine a schedule of partitions which minimize the computation's execution time. Under other circumstances the sequence of grid structures is created dynamically in response to the solution behavior. This precludes a priori scheduling of remappings, forcing one to use remapping heuristics. However, we can compare a posteriori the heuristic and optimal performance by observing and recording the sequence of grid structures, and then constructing an optimal *a posteriori* schedule of remappings.

Let $G_1, G_2, \ldots, G_M$ denote the sequence of grid structures used in the course of solving a problem. Let s_i denote the number of time-steps during which G_i is employed. G_1 can be partitioned by binary dissection or by one of a number of scatter decompositions. If we use binary partioning on G_1, and G_2 is quite similar, we may decide to retain G_1's partition for use on G_2. We may alternatively construct a binary partition tailored to G_2, or use a scatter decomposition. In general, we may partition G_j using a binary partition tailored to G_j, we may use a binary partition tailored to some G_i, $i < j$, or use some scatter decomposition.

We let n_i be the number of partitions we will consider for G_i, and denote them by $p(i,j)$ for $j = 1, \ldots, n_i$. Finally, we let $T(i,j)$ denote the time required to execute s_i time-steps using partition $p(i,j)$ on G_i, and let $C_i(j_1, j_2)$ be the cost of remapping from partition $p(i,j_1)$ to partition $p(i+1,j_2)$. Note that this cost is zero if $p(i,j_1)$ is identical to $p(i+1,j_2)$. Also note that $C_i(j_1,j_2)$ depends on whether $p(i+1,j_2)$ is computed at run-time, or is computed off-line and merely implemented at run-time. When we compute schedules a priori we may assume that the partition is computed off-line. When computing a posteriori schedules, $C_i(i+1,j_2)$ should include the cost of computing $p(i+1,j_2)$ at run-time.

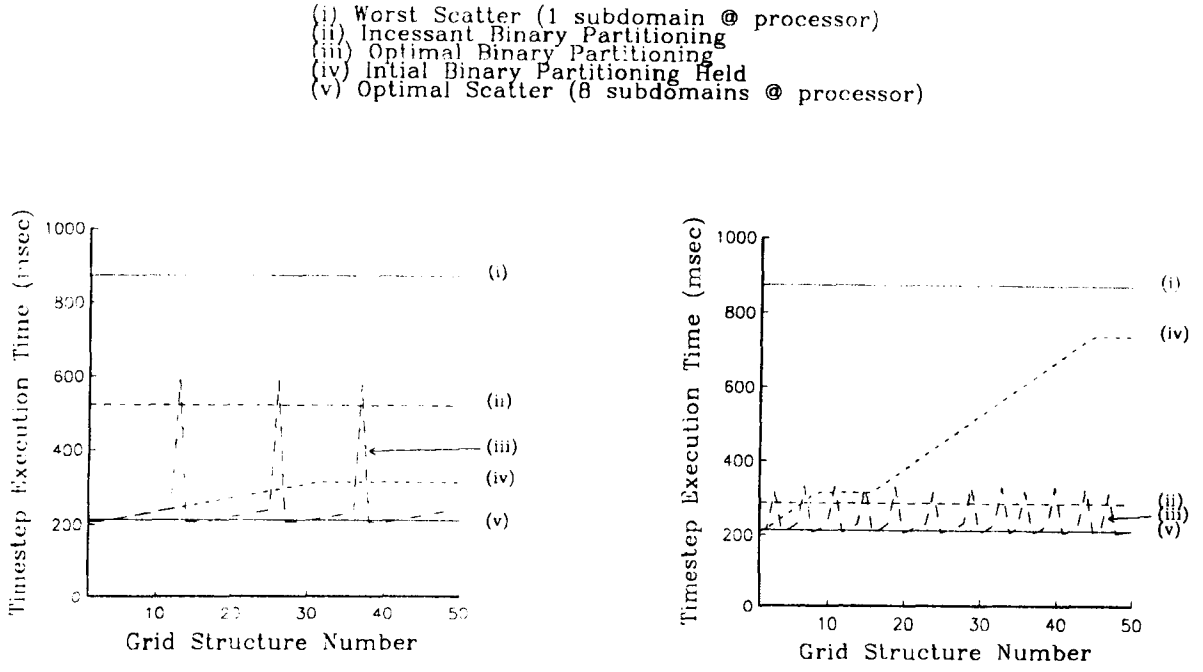
We use dynamic programming to determine a remapping schedule that minimizes the overall execution time. The optimal cost function $A(i,j)$ denotes the minimal remaining execution time, starting from G_i under partition $p(i,j)$. The principle of optimality states that

$$A(i,j) = \begin{cases} T(i,j) & \text{if } i = M \\ T(i,j) + \min\limits_{1 \le k \le n_{i+1}} \{C_i(j,k) + A(i+1,k)\} & \text{if } i \neq M \end{cases} \quad (2)$$

It is straightforward to solve these equations by unraveling the recursion. The optimal schedule of partitions (and hence remapping schedule) is found in the usual way by backtracking. The j_1 minimizing $T(1,j)$ defines the first partition (i.e. $p(1,j_1)$ is the first partition). $T(1,j_1)$ is the minimal execution time possible, given the choice of partitions. The j_2 minimizing $\min_j\{C_1(j_1,j) + A(2,j)\}$ defines G_2's partition. If $p(2,j_2)$ is different from $p(1,j_1)$ then we remap to $p(2,j_2)$ when the grid structure is changed to G_2. In like fashion the j_3 minimizing $\min_j\{C_2(j_2,j) + A(3,j)\}$ defines G_3's partition and the corresponding remapping decision, and so on.

Let $P_{\max}$ be the maximum number of partitions considered for use on any grid structure. The complexity of computing the schedule is $O(MP_{\max}^2)$, once the $T(i,j)$ values are known. The complexity of estimating $T(i,j)$ is $O(n \cdot Size(G_i))$, where n is the number of processors.

Binary partition is a frequently cited technique for balancing irregular workload in a domain (e.g. [1,3]). Given its appealing properties, one is led to ask whether the partition scheduling approach outlined here is of any use—why should we even consider scatter decompositions? Why not simply tailor a binary partition for every grid structure? The answer is that scatter decompositions are better *static* partitions for dynamic workloads than are binary partitions. By using a binary partition one is *forced* to remap and suffer the attendant costs. Figure 4 illustrates a scenario where the optimal static scatter decomposition clearly outperforms any sort of dynamic remapping. Upon a large coarse grid (16384 points) we initially placed a level 1 grid with 2048 points, and a level 2 grid with 1024 points near the left end of the domain. For figure 4(a) this initial structure was then drifted to the right by six coarse-grid points, every 6 time-steps. We plot the average time-step execution time of (i) the worst scatter decomposition policy (one subdomain per processor), (ii) the policy of remapping using binary partitioning whenever a grid is changed, (iii) the optimal binary partitioning policy, (iv) the policy of using the initial binary partition throughout the entire



(a) Regridding every 6 steps

(b) Regridding every 24 steps

Figure 4: Comparison of binary partitioning and scatter decompositions.

computation, and (v) the optimal scatter decomposition policy (eight subdomains per processor). The same plots are made in figure 4(b) for the case where the grid structures are drifted 24 coarse-grid points every 24 time-steps. It is important to note that these measurements include the cost of computing binary partitions at run-time. The grids which suffer this cost are clearly identified by spikes in the performance curve. Clearly the movement of grid structures forces frequent remapping, if we use binary partitioning. On the other hand, the overhead of remapping drives up its cost beyond that of a good scatter decomposition, but the worst scatter decomposition is quite bad. The remapping scheduling algorithm is able to discern whether, when, and which scatter decomposition to use.

We incorporated the scheduling algorithm into the fluids code. At load time the preprocessor reads in the full schedule of grid definitions, computes all $T(i,j)$ and $C_i(j_1,j_2)$, and solves the optimal cost equations to produce a partition schedule which is passed to the fluids code. In general it is possible to reduce the additional load-time delay by using the parallel processors themselves to compute the timing estimates and solve the optimality equations. Alternatively, the schedule can be calculated on a (cheap) front-end processor

prior to loading the code on an time-accounted super-computer, such as the current generation of Crays, Cybers, or ETA-10s.

5 Model Validation

We validated the performance model on a set of grid structures which approximate those used in practice. On large problems we find the predicted values to be within 3% of measured performance; on small problems the model is within 8% of measured performance.

To test the model's robustness we generated grid structures which vary in size and density. Size is simply the number of coarse grid points; density refers to the frequency with which higher level grids appear. A grid structure having size N and density ρ is generated randomly, as follows. The coarse grid with N points is immediately defined, any level 1 grid is dictated to have $N/4$ points, and any level 2 grid is dictated to have $N/16$ points. Next we scan across the coarse grid, at each point performing a random trial which decides whether to begin a level one grid there. The trial compares ρ/N to a (0,1) uniform random number— a level one grid is placed if the random number is smaller. In this case the scan point is advanced past the end of the newly defined grid, and the random trials resume. Level two grids are randomly placed above existing level one grids in entirely the same fashion, save that the probability threshold is increased to $\rho/(N/4)$. The grid structure so constructed is taken to be G_1. We specify that each grid structure be used for ten time-steps. G_2 is constructed from G_1 by "drifting" every higher level grid ten coarse grid positions to the right. In general, G_i is constructed by drifting G_{i-1} eight coarse grid positions.

We compared predicted and measured execution times on problems indexed by size, and grid density. *Large* problems use 16382 coarse grid points, *small* problems use 2048. *Sparse* problems are generated with density factor $\rho = 5$, *dense* problems are created using $\rho = 25$. We randomly generated five different instances of each problem type, and tested each under five, sixteen-processor partition schedules. The *Optimal* partition schedule is obtained from the scheduling algorithm. On these test problems the optimal schedule tended to remap frequently, using binary partitions (the optimal static scatter decomposition was very close to this in performance). This is in stark constrast to the problem illustrated in figure 4; where the cost of computing the binary partition was computed at run-time. This comparison serves to show that the major cost of remapping for this problem is not the data movement, but the partition calculation. The other four partitions, labeled *SD*, are static scatter decompositions indexed by the number of subintervals they assign to each processor. The table below gives the maximum, and average percentage (absolute) deviation of the analytic model from measured performance on all problem types, under the five selected partitions. The greater accuracy seen on the large problems is due to the fact that the large problems spend proportionally more time in the loops we carefully modeled. The problems represented by this data span a

	Large, Sparse		*Large, Dense*		*Small, Sparse*		*Small, Dense*	
Partition	avg	max	avg	max	avg	max	avg	max
Optimal	2.1	2.2	2.5	2.6	5.3	8.2	1.8	2.8
SD 1	0.1	0.1	0.8	1.3	3.7	4.8	0.3	0.5
SD 8	1.6	2	1.7	1.8	5.8	6.8	2.7	3.4
SD 16	0.4	0.4	1.1	1.1	7.1	8.0	5.0	5.2
SD 32	0.7	0.7	0.5	0.6	7.5	8.0	6.7	7.3

Table 1: Absolute (percentage) deviation of analytic model from measured performance.

wide spectrum of behaviors. The large problem running times are an order of magnitude larger than those of the small problems. For a single problem the worst partition's running time is usually at least twice as large as the best partition's. Both computation bound and communication bound partitions are represented. The data in table 1 gives us a high degree of confidence in the model, in its performance projections, and in the conclusions we may draw from its use.

6 Summary and Conclusions

The highly structured nature of many scientific codes make them well-suited for deterministic performance modeling. Most codes spend a large fraction of their time in a few loops—we can capture a parallel code's overall performance by carefully modeling these loops' behavior. This type of modeling is critically needed to solve the problem of mapping a dynamic, irregular computation onto a parallel computer. We have illustrated these points using a fluids code. We showed how the model can be used to optimally schedule problem remappings, and showed that the model is accurate.

Models of the type we propose are useful in a number of ways. For example, *adaptive gridding* schemes generate new grid structures as a function of the solution behavior. A very real problem is the dynamic decision to remap or not in response to an unpredictable change in grid structure (see [7] for a treatment of this problem). Our ability to accurately model performance allows us to evaluate the costs and benefits of remapping, at run-time. The analytic model can also be used in performance studies of remapping decision policies. We can measure the performance under a policy in a run, then construct a posteriori an optimal remapping schedule for that run. The optimal schedule gives a baseline against which the remapping decision policy can be judged.

The method we used to model the fluids problem depends only on the fact that the code spends most of its time in a few easily identified loops, that we can parametrically model those loops' performance, and that

the code synchronises in a regular fashion. A large number of scientific codes have these same characteristics, and are therefore promising candidates for modeling.

References

[1] S. B. Baden. *Run-Time Partitioning of Scientific Continuum Calculations Running on Multiprocessors.* Technical Report LBL-23635, Lawerence Berkley Laboratory, June 1987.

[2] M.J. Berger and S. H. Bokhari. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Trans. on Computers*, C-36(5):570–580, May 1987.

[3] G. Cybenko and T.G. Allen. Parallel algorithms for classification and clustering. In *Proceedings of the 31st Annual Int'l Tech. Symposium on Optical and Optoelectronic Applied Science and Engineering*, San Diego, CA, August 1987.

[4] G. Fox, M. Johnson, G. Lyszenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Computers.* Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[5] N. Matelan. The Flex/32 multicomputer. In *Proceedings of the 12th International Symposium on Computer Architecture*, pages 209–213, Computer Society Press, June 1985.

[6] D. M. Nicol and F. H. Willard. Problem size, parallel architecture, and optimal speedup. *Journal of Parallel and Distributed Computing*, 5:404–420, August 1988.

[7] D.M. Nicol and J.H. Saltz. Dynamic remapping of parallel computations with varying resource demands. *IEEE Trans. on Computers*, 37(9):1073–1087, September 1988.

[8] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vettering. *Numerical Recipes.* Cambridge University Press, New York, 1986.

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

NASA
National Aeronautics and Space Administration

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA CR-181756 ICASE Report No. 88-58		

4. Title and Subtitle	5. Report Date
ACCURATE MODELING OF PARALLEL SCIENTIFIC COMPUTATIONS	November 1988
	6. Performing Organization Code

7. Author(s)	8. Performing Organization Report No.
David M. Nicol and James C. Townsend	88-58
	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, NAS1-18605 AFOSR 88-0117
	13. Type of Report and Period Covered
12. Sponsoring Agency Name and Address	Contractor Report
National Aeronautics and Space Administration Langley Research Center Hampton, VA 23665-5225	14. Sponsoring Agency Code

15. Supplementary Notes

Langley Technical Monitor:
Richard W. Barnwell

Final Report

Submitted to the 1989 Sigmetrics Conference

16. Abstract

Scientific codes are usually parallelized by partitioning a grid among processors. To achieve top performance it is necessary to partition the grid so as to balance workload and minimize communication/synchronization costs. This problem is particularly acute when the grid is irregular, changes over the course of the computation, and is not known until load-time. Critical mapping and remapping decisions rest on our ability to accurately predict performance, given a description of a grid and its partition. This paper discusses one approach to this problem, and illustrates its use on a one-dimensional fluids code. The models we construct are shown empirically to be accurate, and are used to find optimal remapping schedules.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
parallel processing, dynamic remapping, analytic modeling	61 - Computer Programming and Software 66 - Systems Analysis Unclassified - unlimited

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of pages	22. Price
Unclassified	Unclassified	13	A02

NASA FORM 1626 OCT 86