NASA Contractor Report 178379

ICASE REPORT NO. 87-39

ICASE

PRINCIPLES FOR PROBLEM AGGREGATION AND ASSIGNMENT IN MEDIUM SCALE MULTIPROCESSORS

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

Principles for Problem Aggregation and Assignment in Medium Scale Multiprocessors

David M. Nicol [*]
The College of William and Mary

Joel H. Saltz [†]
Yale University

Abstract

One of the most important issues in parallel processing is the mapping of workload to processors. This paper considers a large class of problems having a high degree of potential fine grained parallelism, and execution requirements that are either not predictable, or are too costly to predict. The main issues in mapping such problems onto medium scale multiprocessors are those of aggregation and assignment. We study a method of parameterized aggregation that makes few assumptions about the workload. The mapping of aggregate units of work onto processors is uniform, and exploits locality of workload intensity to balance the unknown workload. In general, a finer aggregate granularity leads to a better balance at the price of increased communication/synchronization costs; the aggregation parameters can be adjusted to find a reasonable granularity. The effectiveness of this scheme is demonstrated on three model problems: an adaptive one-dimensional fluid dynamics problem using message passing, a sparse triangular linear system solver on both a shared memory and a message-passing machine, and a two-dimensional time-driven battlefield simulation employing message passing. Using the model problems we study the trade-offs between balanced workload and the communication/synchronization costs. Finally, we use an analytic model to explain why the method balances workload, and minimizes the variance in system behavior.

[*]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-0310, and NSF grant DCR 8106181.

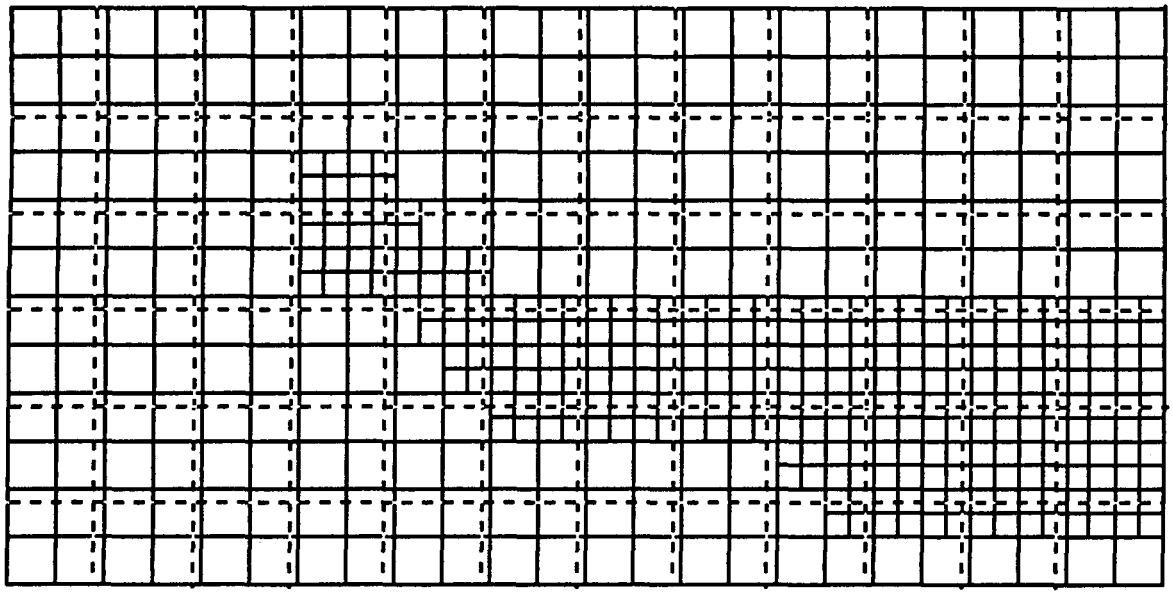


Figure 1: Irregular Grid for Two-Dimensional PDE

1 Introduction

We consider the broad class of computational problems that exhibit a high degree of potential fine-grained parallelism, and whose dynamic workload is either unpredictable or too costly to predict. In these problems, the computation is often easily decomposed into a sequence of *phases*, each phase being composed of a large number of operations which we will call *grains*. This paper treats the problems of aggregating sets of grains into *work units*, and mapping the work units onto a medium scale message-passing or shared memory multiprocessor. The machines used in these investigations were the Encore Multimax, the Flex/32, and the Intel iPSC hypercube.

Within a phase, the aggregated granularity of work units can often be specified parametrically, without specific regard for the execution requirements of the resulting work units. This is particularly easy when the computation is tied to a physical domain, like the numerical solution of a partial differential equation (PDE) using an explicit method of integration. A grain in this case can be the set of computations required to update a single grid point solution. We aggregate these grains by tessellating the domain with subregions of regular and equal size; Figure 1 shows how a two-dimensional irregular grid (intersection of solid lines define grid points) for a PDE is aggregated into rectangles (by the dotted lines). The work unit granularity is specified by the parameters defining a subregion's shape and size. The execution requirement of such a subregion is the collection of all workload related to the physical region it covers, e.g. the workload of a region in

Figure 1 consists of all updates to grid points contained in that region. It shall be argued below that the principles discussed in this paper extend to a wide variety of numerical and non-numerical problems.

Given a collection of work units we can attempt to balance the unknown workload during a phase by placing an equal number of work units on each processor. The likelihood of achieving a reasonable balance increases as the work unit size becomes smaller. But in medium scale multiprocessor environments (particularly message-passing environments), the assignment of fine grained computations in ways that balance load may incur high communication costs due both to higher communication volume, and a higher number of communication startups. We therefore observe a trade-off between load imbalance and communication overhead, but can control that trade-off with the aggregate granularity parameters.

In many situations, aggregation may be performed in a way that leads to an inverse relationship between work unit size and *the number* of phases required to complete a problem. If we perform synchronizations between phases, we note a tradeoff between load imbalance and synchronization costs.

We demonstrate simple parametric aggregation techniques that are applicable for a variety of physically-based problems. We then describe a straightforward means of indexing the work units so that their computational requirements exhibit a degree of locality, i.e., their computational requirements become more highly correlated as the difference in the indices of the work units becomes smaller. Under this mapping the communication that must take place between work units is predominantly between nearby work units.

The indexed work units may be assigned to processors in a number of ways. The method we shall explore in this paper is that of *wrapping*. In its simplest version one numbers the processors from 0 to $P-1$, and labels work units with indices from 0 to $n-1$. Work unit j is assigned to processor $j \bmod P$. One can also generalize wrapping to apply to a (possibly logically defined) multidimensional processor array. We assume that processors are assigned indices $(i_1, i_2, .., i_d)$ where $0 \leq i_k \leq P_k - 1$, and work units are assigned indices $(j_1, j_2, .., j_d)$. Work unit $(j_1, ..., j_d)$ is assigned to processor ($j_1 \bmod P_1, ..., j_k \bmod P_d$). Figure 2 illustrates a wrapped assignment of a 6×12 work unit array onto a 3×4 array of processors.

This work is part of the ongoing Crystal/ACRE [26] parallel programming 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 very tightly coupled systems, and onto loosely coupled systems. Parameterized aggregation and assignment is very promising for our purposes, in that it offers an automated mapping (and/or remapping) system a reasonably sized space of easily determined mappings to choose from; as we will see, some of these mappings are better suited for tightly coupled systems, and others for loosely coupled systems. The load balancing properties of the kind of aggregation and mapping studied here have been briefly discussed in [5]. We extend that work in a number of ways; we explore in some detail the use of aggregation

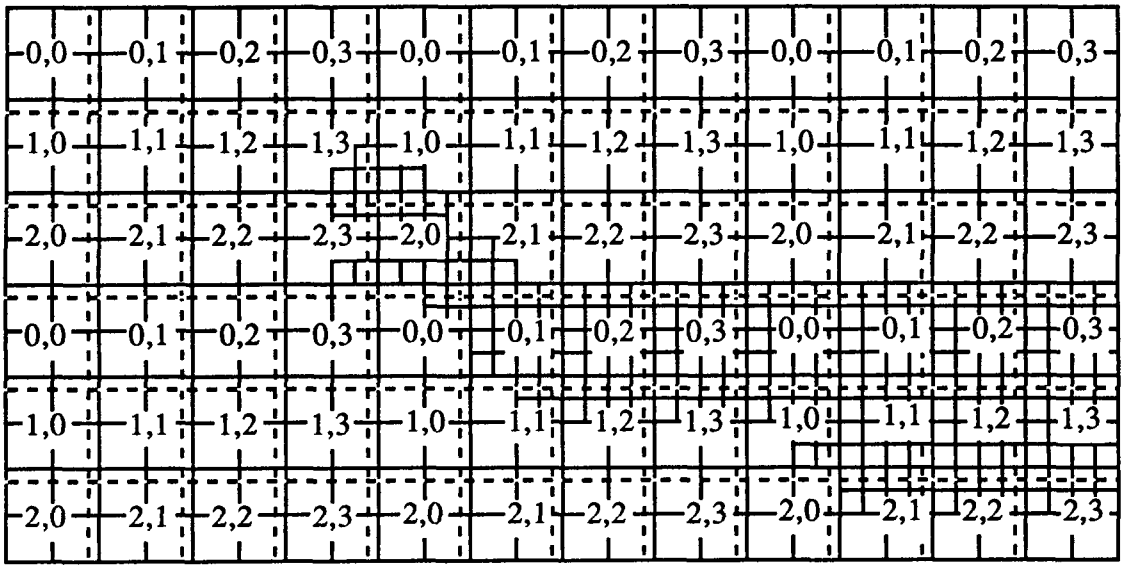


Figure 2: Wrapped Assignment of PDE Domain

in dynamic problems, extend methods of parameterized aggregation to problems without a physically obvious time direction, and develop a statistical model to describe the performance implications of parameterized aggregation and of wrapped mappings. In a more general context, wrapped assignments have been used for a wide variety of problems in parallel processing, a few examples include [11], [23], [6], [27].

We examine three model problems to illustrate how work unit granularity can be specified parametrically, and how the resulting work units are subsequently mapped by wrapping. We study the problems' measured performance to illustrate how the granularity parameters affect the load imbalance/overhead tradeoffs. One problem is a one-dimensional adaptive fluid dynamics computation. The computation uses an explicit numerical method to solve a time-dependent partial differential equation with respect to time. Each phase or time step of the computation advances "time" by some Δt; within a phase, a number of independent calculations are required at each of the discretized domain's grid points. Parameterized aggregation is accomplished by dividing the domain into a number of subintervals having equal length (subinterval length being the parameter). At any given time step, a work unit consists of all computations associated with grid points within a subinterval. The workload distribution changes in time as the adaptive regridding adds and deletes grid points from the domain. The regions where new grid points will appear are generally unpredictable, so that the future workload behavior is unpredictable. This problem computation is implemented on the Flex/32 multicomputer [14] using a message-passing paradigm.

The second model problem is the solution of a sparse triangular linear system of equations arising from the incomplete factorization of a matrix obtained from the discretization of a partial differential equation. In this problem, there is no physically meaningful choice of a "time" axis; in the course of solving the problem a time axis will be chosen. In this problem, the way in which work units are chosen will effect the number of phases required to complete the problem.

In the process of performing the preconditioned conjugate gradient algorithm [3], [10], numerous solutions of sparse triangular systems must be obtained. The aggregation parameters for this problem are obtained from an analysis of a directed acyclic graph specified by the lower triangular matrix to be solved. These parameters turn out to be intimately related to the discretized physical domain of the PDE; they implicitly specify the subregions of domain that are to be treated as work units. A work unit is thus defined by the computations related to a region, and a phase is the collection of work units that can be concurrently evaluated. Unlike the fluid dynamics problem, this problem's behavior as a function of phase could in theory be completely analyzed before actually solving the system. The analysis required, along with the computation of the mapping or schedule to be used during each phase can be quite expensive. It can be preferable not to perform the analysis, and specify a simple wrapped mapping which makes the appropriate tradeoff between load balance and communication and/or synchronization overheads. This computation is implemented using a shared memory paradigm on the Encore multicomputer [4], and using a message-passing paradigm on the Intel iPSC hypercube multiprocessor [21].

The third model problem is a time-driven battlefield simulation based on CORBAN [8]. Each phase or time step in the simulation consists of performing all simulation work required in the next Δt amount of simulation time. At a time step we view the simulation as a collection of independent computational activities distributed over a two-dimensional geographical domain. Like the fluids problem, the aggregation parameters specify the size of regions that cover the domain. At a given step, a work unit is defined by all of the simulation activity required within a domain region. Simulation activity is bound to combat *units* that move through the domain. Computational activity is particularly intense when units from opposing sides become geographically close, causing the simulation of a battle. The future geographical position of units is unpredictable (or too costly to compute), so that we must treat the future distribution of workload as unknown. The battlefield simulation has also been implemented on the Flex/32 using a message-passing paradigm. There are numerous other examples of problems displaying many of the characteristics of these model problems; these examples include iterative relaxation solutions of partial differential equations and simulations of other physical processes in a spatial domain, e.g. gate-level or transistor level VLSI circuit simulation.

One obvious alternative to wrapping is to systematically *schedule* workload to processors during each phase of the computation so as to maintain a good load balance. The difficulties with this approach include the need to frequently estimate the computational requirements of each work unit, and the additional cost of the scheduling algorithm. On a

message-passing machine there may also be a significant cost for these workload reassignments. In some cases where the computational requirements of work units change fairly gradually from phase to phase of a computation, one may dynamically *remap* the workload as the computation evolves; each mapping is then based on the known distribution of the workload at the time of the mapping. A host of issues raised by dynamic remapping (particularly *when* to remap) have been discussed elsewhere [17],[18],[25]; even so, it is important to choose mappings which are effective during their period of use. This is especially true if the cost of remapping is very high, so that remapping is infrequent.

The main purpose of this paper is to point out principles for workload aggregation and assignment that appear to apply to a wide range of dynamic computational problems. We show how a simple set of parameters effectively controls performance, and note that performance depends strongly on the parameter choice. The principles we espouse form the basis of an automated aggregation and assignment system [26] under development; they also underlie a mechanism for run-time remapping of parallel computations [19]. In Section 2 we discuss workload aggregation and assignment in a general way. Sections 3,4, and 5 make these principles more concrete by applying them to three diverse problems. The implementation and performance of these problems on various parallel processors are discussed. The data we present clearly shows the trade-off between load balance and overhead due to fine-grained aggregation. Finally, in Section 6 we use a general stochastic model to formally explain wrapping's ability to balance load. This model shows that among all mappings of uniform partitions, wrapping minimizes the variance in processors' loads. We explain why minimization of variance is a desirable property for load balancing. We also show that under wrapping, a processor's variance decreases rapidly as the work unit size decreases, and we give empirical evidence that extensive (but not complete) aggregation still significantly improves load balancing.

2 Workload Aggregation and Assignment

Before examining the model problems in detail, we use this section to discuss aggregation and assignment in a general way. A major principle we espouse is the parametric aggregation of fine-grained workload (grains) without specific concern for their execution costs. This principle makes sense when either (i) we cannot or should not attempt to estimate the grains' execution costs, (ii) the execution costs are liable to change unpredictably as the computation progresses, or (iii) we expect the grains' execution costs to be roughly the same. The details of the aggregation method should depend in a general way on the computation, a fact we illustrate by example. We then describe how the aggregated work units are wrapped.

Consider a computation based on a one dimensional domain $[0, L]$ such that the update computation for a value at position p at step s depends only on values at points near p at step $s-1$ (e.g. our model fluids problem). While the computation considers a discretized

version of the interval, our aggregation method will view it as being continuous. To uniformly aggregate the workload on $[0, L]$ into n work units, we simply define the n work units $[0, L/n], (L/n, 2L/n], \ldots, ((n-1)L/n, L]$. n controls the degree to which we aggregate the underlying grains. During a given phase, a work unit's computational load is the collection of all grid point updates for grid points lying in the region's physical interval. During a given phase, all of the work for one work unit can be performed in parallel with the work for another unit (provided that the results from the previous step have been locally disseminated). Note that this type of aggregation is independent of the way in which workload is distributed.

The partitioning used for the one-dimensional case above extends easily in certain cases to a two-dimensional problem, like the battlefield simulation model problem. The domain is viewed as the set of all (x, y), $x \in [0, W]$, $y \in [0, H]$ for some positive W and H. We aggregate into $h \cdot w$ rectangular work units, each having height H/h and width W/w. Any computational work associated with a point (x, y) is performed by the processor holding the work unit containing (x, y). This type of aggregation is appropriate if every work unit's computation can occur concurrently with any other's during a phase. The parameters h and w control the granularity of the aggregation; note again the independence of the method from the workload distribution. This method clearly extends to domains with higher dimensions.

In many algorithms, no physically meaningful time direction is available. Given a choice of computational grain it is still straightforward to partition the computation into a sequence of phases, each phase consisting of completely independent computational grains. Grains within a phase can then be clustered into work units. We next illustrate a more sophisticated kind of clustering that changes the number and the composition of phases. It should be noted that this method's principles (described in the context of the sparse triangular solve below) can *also be applied to problems involving a physical time axis*, including the other model problems described here.

Our second model problem arises from a discretized two dimensional domain but in this case there are data dependency relations between variables corresponding to different mesh points. We use conjugate gradient type methods that employ the effective Incomplete LU form of preconditioning [3] for solving systems of linear equations arising from discretizations of two dimensional elliptic partial differential equations. In this algorithm, an incomplete factorization of the matrix arising from the discretized domain is carried out; in the simplest case the sparsity structure of the upper or lower triangular matrix arising from this incomplete factorization is identical to that of the upper or lower triangular portion of the original matrix describing the domain. We shall limit our discussion here to this simplest case of *zero fill* incomplete factorization.

Preconditioned conjugate gradient algorithms are iterative; one repeatedly performs matrix vector multiplications, inner products and solutions of the upper and lower triangular systems obtained from the incomplete factorization described above. The data dependencies in the problem make efficient solution of the triangular system particularly

challenging. Furthermore, the ability to solve the system efficiently is an essential component of a high performance algorithm, since the triangular system must be repeatedly solved.

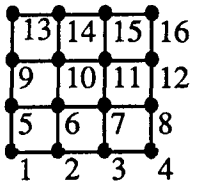
A grain for this problem is taken to be the work associated with solving for a single variable. Since the matrices generated from most discretizations of partial differential equations are quite sparse, there will typically be just a few floating point operations associated the solution of each variable.

The data dependencies inherent in the solution of a triangular system of equations of the type we consider can be described with a directed acyclic graph. This graph has a close relationship to the undirected graph describing the mesh, and plays a key role in the aggregation process. For the sake of specificity we will consider the solution of the *lower* triangular matrix obtained from the incomplete LU factorization. The directed acyclic data dependency graph contains the same links as did the undirected graph. The matrix describing the original mesh contains a row for every mesh point, the row's nonzero elements describe the point's dependencies on other mesh points. Consequently this matrix is dependent on the *order* (indices) given to the mesh points, as are the triangular matrices obtained from incomplete factoring. For a given ordering of mesh points, the links of the directed graph originate from the variables assigned lower indices and end in those assigned higher indices. To illustrate these relationships, consider a rectangular mesh with a five point template in Figure 3(a). The matrix M obtained from this mesh has a non-zero structure shown in Figure 3(b) ($*$ denotes a non-zero element). The zero fill lower triangular factor L is shown in Figure 3(c), and the directed graph that describes the data dependencies encounted in solving the equations Lx = b is depicted in Figure 3(d).

We hence have a directed acyclic graph (DAG) imposed on the mesh where the point index of a link's head is larger than its tail coordinate. The value at a point may be computed once the values at all of its ancestors in the DAG have been computed. Partitioning is achieved by coalescing points into rectangular work units, as shown in Figure 3(e).

Taking the data dependencies between grains into account allows one to identify an ordered sequence of sets of work units; the work units belonging to each set can be evaluated in parallel. The evaluation of each set of work units may be viewed as a step in a multi-step parallel computation where global synchronization occurs between steps. The computation associated with a point can vary depending on the number of precedence arcs into the point. The work units within a wavefront can be viewed as a one-dimensional array. Like the work units of the previous examples, the work unit here is a basic element of schedulable work, with potentially variant workload. The height and width of a work unit are parameters controlling the granularity of the aggregation.

The three preceding methods illustrate parameterized aggregation. For each method we can view the work units during a phase as having d-dimensional indices $(j_1, j_2, .., j_d)$ where d depends on the aggregation method (d is 1 or 2 for the methods presented). For the purpose of assignment we suppose that the available processors can be labeled with d-dimensional indices $(i_1, i_2, .., i_d)$ where $0 \leq i_k \leq P_k - 1$. The wrapping assignment consists



(a) 5-Point Mesh

$$\begin{pmatrix} B & D & 0 & 0 \\ D & B & D & 0 \\ 0 & D & B & D \\ 0 & 0 & D & B \end{pmatrix}$$

Block Matrix

$$B = \begin{pmatrix} * & * & 0 & 0 \\ * & * & * & 0 \\ 0 & * & * & * \\ 0 & 0 & * & * \end{pmatrix} \quad D = \begin{pmatrix} * & 0 & 0 & 0 \\ 0 & * & 0 & 0 \\ 0 & 0 & * & 0 \\ 0 & 0 & 0 & * \end{pmatrix}$$

Submatrices in Block Matrix

(b) Block Matrix from 5-Point Mesh

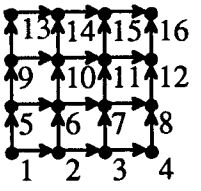
$$\begin{pmatrix} \hat{B} & 0 & 0 & 0 \\ D & \hat{B} & 0 & 0 \\ 0 & D & \hat{B} & 0 \\ 0 & 0 & D & \hat{B} \end{pmatrix}$$

Block Matrix

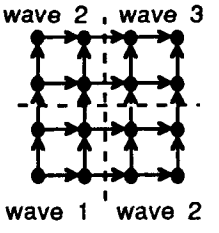
$$\hat{B} = \begin{pmatrix} * & 0 & 0 & 0 \\ * & * & 0 & 0 \\ 0 & * & * & 0 \\ 0 & 0 & * & * \end{pmatrix}$$

New Submatrix

(c) Lower Triangular Matrix From Incomplete LU Factorization



(d) Data Dependency DAG



(e) Block Wavefronts

Figure 3: Inter-relationships Between Mesh and Matrices in Factorization

simply of assigning work unit $(j_1, ..., j_d)$ to processor $(j_1 \bmod P_1, ..., j_k \bmod P_d)$. In the following three sections we illustrate how aggregation and wrapping is applied to three diverse problems.

3 Fluids Problem

We first apply the principles of aggregation and assignment to a problem in fluid dynamics. This section describes the problem, reports the achieved performance on the Flex/32 multicomputer, and comments on the trade-offs between load imbalance and communication/synchronization overhead.

A one-dimensional fluid dynamics code serves as our first model problem. The code numerically simulates the density $\rho(r,t)$ and velocity $v(r,t)$ of a compressible fluid flowing through a tube, as a function of position r and time t. Our implementation employs the ETBFCT code [2] that solves the general continuity equation

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

where C_1, D_1, D_2, D_3 are problem dependent constants or functions. For the sake of simplicity, all of our experiments set these functions equal to zero, and use the equation of state $v(r,t) = \rho(r,t)/2$ (which makes this Berger's equation). The one dimensional tube model is represented by a grid having one point every h spatial units; we will call this the *coarse* grid. We assume that the value of ρ is known at every grid point at time $t = 0$, that the fluid flows from left to right, and that the density and velocity values of fluid entering the tube are given as needed. Presented with the fluid density and velocity values throughout the domain at time t_0 and the density and velocity of fluid entering the tube at time $t_0 + \Delta t$, ETBFCT numerically integrates the continuity equation in t to solve for fluid behavior at time $t_0 + \Delta t$. ETBFCT has second-order accuracy.

Variant computational behavior is caused by employing an adaptive gridding technique proposed in [1]. Every $5\Delta t$ time units the solution behavior is examined, and additional *fine* grid points with a spacing of $h/6$ are added in subregions where the examination predicts that truncation error will exceed a user defined limit (the mechanism employed in [1] is quite detailed and is beyond the scope of this paper). Every coarse grid point in such a region has a corresponding fine grid point at exactly the same tube position. Similarly, fine grid points are removed from regions where they are no longer needed (coarse grid points are never removed). Figure 4 illustrates how patches of grid points might be applied to the domain. At time t_0, the computation first integrates only the coarse grid, from time t_0 to time $t_0 + \Delta t$. Then, each fine grid is integrated 6 times where each integration uses a time step of $\Delta t/6$; boundary values at the ends of subgrids are interpolated from the corresponding coarse grid values. After fully integrating the fine grids, function values at coarse grid points covered by fine grid points are replaced with the improved function values

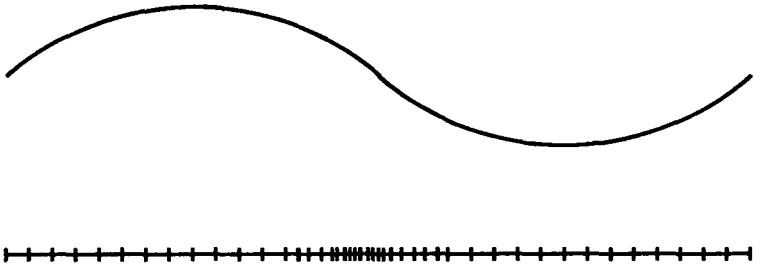


Figure 4: Dynamic Regridding of One Dimensional Domain

from the corresponding fine grid points. In its fullest generality, the method described in [1] allows a recursive attachment of finer grids to fine grids.

The behavior of the physical fluid system is governed by *boundary conditions*, or the behavior of fluid entering the system; the continuity equation above allows the introduction of fluid at any point in the domain. Fine subgrids tend to appear in regions containing sharp gradients in the fluid density. Gradients can be introduced by the boundary conditions, and can also form of their own accord. During the course of the computation, future regridding activity can be unpredictable because the future behavior of the boundaries can be unpredictable or too complex to effectively pre-analyze. More importantly, it is unpredictable because the only way to determine what the fine grid distribution at time t will be is to simulate the system up to time t. Despite the unpredictability, there is positive correlation in workload density in time and space. If domain points r_i and r_j are spatially close and if a fine grid covers point r_i at time t_0, then it is likely that the fine grid covers r_j at time t_0, and covers both points at time $t_0 + \Delta t$. A wrapped fine-grained aggregation will increase the chance that points r_i and r_j (and the subgrids covering them) are balanced by residing in different processors.

Two factors dominate the execution time of a parallel implementation of our fluids code: communication cost and computation cost. Using ETBFCT the density at position r_0 at time $t_0 + \Delta t$ depends functionally on the density values for all grid points between $r_0 - 5h$ and $r_0 + 5h$ at time t_0 (approximately 50 floating point operations are used to update the density at a coarse grid point). This dependence requires inter-processor communication for points that are functionally dependent but reside on different processors. Figure 5 shows aggregation leading to exactly as many work units as there are processors. If processor j is assigned a partition containing points $[r_0, \ldots, r_0 + Mh]$, then processor j must send

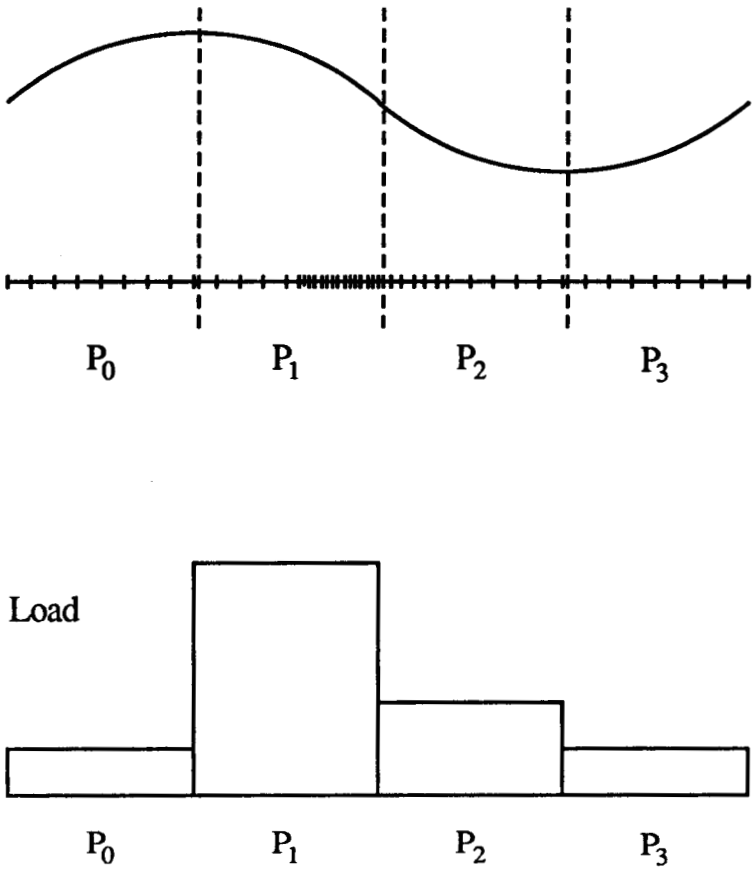


Figure 5: High Degree of Aggregation and Load Balance

points $[r_0, \ldots, r_0 + 4h]$ to processor $j-1$, and will receive points $[r_0 - 6h, \ldots, r_0 - h]$ from processor $j-1$. A similar exchange with the partition's right boundary points is undertaken with processor $j+1$. This mapping minimizes the average volume of data that must be communicated between processors. It is clear though that this aggregation inadequately balances the load. Figure 6 illustrates the same situation with less extensive aggregation and shows that the load balance is significantly better. On the other hand, the smaller work units require more inter-processor communication due to the increased number of partition interfaces. In fact, this degree of aggregation requires proportionally more computation because of a fixed additional computation cost associated with every partition, regardless of size. The optimal degree of aggregation must somehow balance the benefits and costs of wrapping.

Before describing the performance of our method on the Flex/32 [14], we briefly describe its architecture. The Flex/32 at NASA Langley Research Center has twenty NS32032 based processors. Two of the processors are used as hosts and as general purpose UNIX machines; the remaining eighteen processors are used for parallel processing. Each processor has approximately 1M bytes of local memory; there is a global memory with approximately 2.25M bytes. Our implementation of the fluids code uses the shared memory only as medium through which messages are passed. Even in a message-passing machine much of the cost of communication lies not in electronic transmission time, but in message packing, unpacking, and general system overhead. Consequently, the performance of our code on the Flex/32 should be reflective of message-passing machines with fast communication channels, relatively small packet sizes, but not necessarily fast operating system support.

The trade-off between load balance and communication overhead is apparent from the following experiment. The coarse grid is composed of 512 points, and eight processors are used (using more processors did not significantly improve performance for this size of coarse grid). The initial fluid density is constant throughout the domain; dynamic regridding is caused by the injection of a wave at the left end of the domain. The computation is run for two hundred time steps. The domain was alternately divided into 64, 47, 32, 16, and 8 (approximately) equal length subintervals which were wrapped. The uniformity of the coarse grid ensures that each subinterval contains the same number of coarse grid points (with the possible exception of the rightmost subinterval). Figure 7 shows the running time in seconds plotted as a function of the number of coarse grid points in a subinterval. Technical reasons require that at least eight points be aggregated into a work unit. The optimal performance is achieved when a work unit covers sixteen coarse grid points. The speedup at the optimal point was 5.1. Thus we see that the tradeoffs have a significant impact on performance, and that neither the least extensive nor most extensive degree of aggregation achieve the best performance. Somewhat better speedups have been achieved on this problem by dynamically changing the extent of aggregation between time-steps [17].

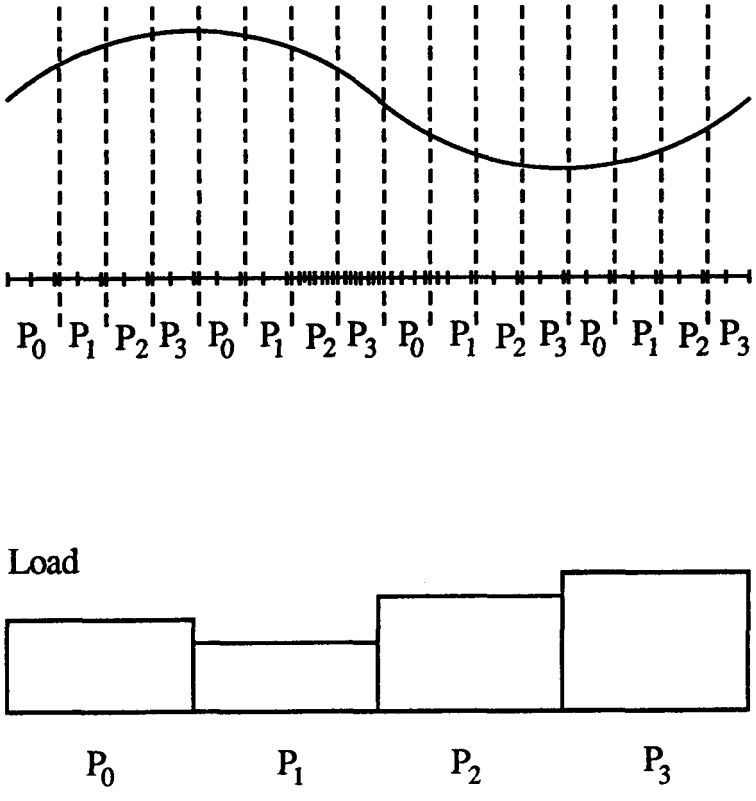


Figure 6: Low Degree of Aggregation and Load Balance

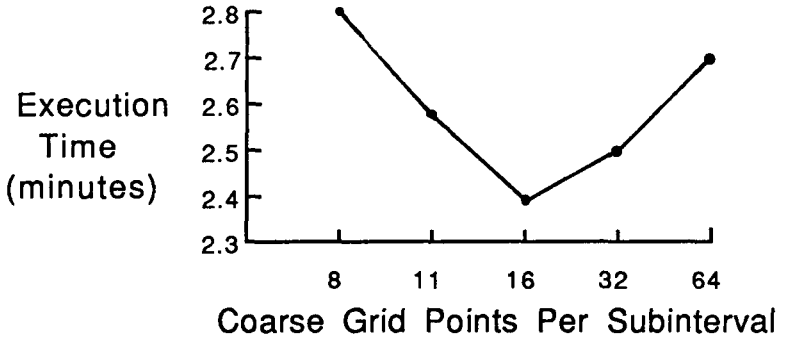


Figure 7: Fluids Code Performance as Function of Aggregation

4 Triangular Solve

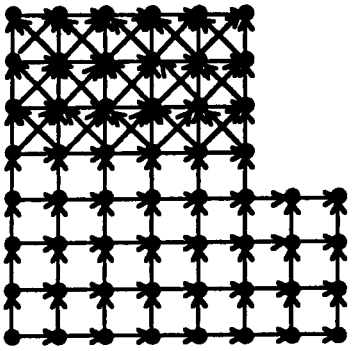
Our second model problem considers the solution of a class of linear equations arising from the use of a style of pre-conditioned conjugate gradient algorithms. Our methods are implemented using a shared-memory paradigm on the Encore Multimax [4]; we also implemented our method using message-passing on the Intel iPSC [21] . Again we observe trade-offs between load balancing and communication/synchronization overhead, and note the dependence of those trade-offs on the architecture used.

4.1 Problem Partitioning

4.1.1 Overview

In the simplest form of incomplete LU preconditioning, the factors L and U (where A = LU) have the same sparsity structure as the lower and upper portions of A respectively [10]. Intuitively this means that if the solution for variable x_j in the equation Ax = b depends on the solutions for all variables in a set S, then the solution for x_j in Lx = b depends only on the solutions for variables in S with indices smaller than j. A prior knowledge of the sparsity structure is used when generating the problem mapping in the example to follow. Prior knowledge is not needed when a matrix oriented version of the problem mapping is used [24].

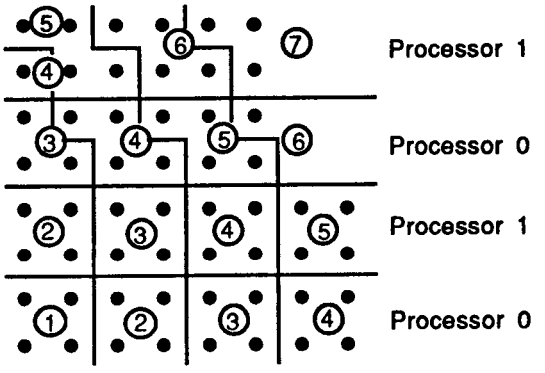
As discussed in Section 2, the data dependencies involved in solving the triangular incompletely factored matrix for a given mesh may be represented by a directed acyclic graph. This DAG is simply a directed version of an undirected graph on the problem mesh. Figure 8(a) depicts the directed acyclic graph describing the data dependency relations between variables to be solved for in a problem arising from an L-shaped mesh. The direction of the links in the DAG depends on the ordering of the variables; here we assume that variables are ordered from left to right beginning with the bottom row of mesh



(a) Data Dependency
DAG on Domain Points

11	12	13	14	15	16			Processor 1
9	10	11	12	13	14			Processor 0
7	8	9	10	11	12			Processor 1
5	6	7	8	9	10			Processor 0
4	5	6	7	8	9	10	11	Processor 1
3	4	5	6	7	8	9	10	Processor 0
2	3	4	5	6	7	8	9	Processor 1
1	2	3	4	5	6	7	8	Processor 0

(b) Mesh Points Labeled by Wavefront



(c) Wrapped Assignment of Blocks
To Processors

Figure 8: Aggregation and Assignment of Problem Arising from an L-Shaped Mesh

points.

The points in the DAG can be separated into disjoint wavefronts using a topological sort. Sets of points sharing no data dependencies are peeled from the graph, these wavefronts are assigned consecutive numbers (Figure 8(b)). All points in any given wavefront can be solved for simultaneously; a point in wavefront i requires data only from wavefronts $j < i$.

We now briefly describe how the parameterized aggregation hinted at in Section 2 is carried out. We assign consecutive rows of the mesh to *row blocks* and assign these blocks to processors in a wrapped manner. Figure 8(c) illustrates a wrapped assignment of row blocks to processors. Because of the block dimensions, this assignment wraps *two* consecutive mesh rows per processor (as shown, this does not imply that a processor receives only two rows). We also solve for the variables corresponding to *two* wavefronts in the blocks containing the first mesh row during each phase. Note that the data dependencies depicted in Figure 8(c) allow one processor to perform as much work as is desired from the first mesh row, since no point in the row depends on data from any other row. For points in other mesh rows, we perform as much work as is allowed by the data dependency relations. Note that during each phase of the computation, we assume the availability of any information produced during a previous phase.

As shown above, we may aggregate work using two parameters—the *block size*, or number of mesh rows per row block, and the *window size*, or number of wavefronts per block.[1] The selection of these parameters gives rise to block wavefronts, which can be viewed as phases of the problem. This selection affects the number of phases required to solve the problem. Note that the assignment illustrated in Figure 8(b) leads to 16 phases, while that in Figure 8(c), leads to only 7 phases. As shown in [24], the choice of window size determines the number of phases in the problem. In message-passing machines reducing the number of phases reduces the number of communication startups throughout the problem. In machines that efficiently support shared memory, a reduction in phases reduces synchronization costs. Increasing the block size also reduces communication between processors in machines utilizing fast local memory. As observed with the fluids problem, the reduction in overhead achieved with extensive aggregation comes at the risk of load imbalance.

We emphasize that the distribution of work during a given phase depends on both the shape of the domain and the data dependency relations between mesh points. Table 1 shows the number of floating point adds and multiplies each block in Figure 8(c) must perform during each phase. A detailed analysis of the principles involved in the generation of parameterized mappings such as those described above, along with performance analyses is found in [24].

[1] Rigorously, the number of wavefronts per block in blocks containing the first mesh row.

	0	0	0	2	10	10	10	Block 4
	0	0	5	12	12	3	0	Block 3
	0	6	8	8	8	0	0	Block 2
	4	6	6	6	0	0	0	Block 1
Phase	1	2	3	4	5	6	7	

Table 1: Floating Point Operations per Block per Phase

4.2 Experimental Results

4.2.1 Preliminaries

The figures discussed in the current section depict timing measurements made during a forward substitution computation. The matrix utilized was generated through the zero fill incomplete factorization of a square mesh employing a 5-point template, and an L shaped mesh which employed both 5-point and 9-point templates.

The machines used for this investigation were the Encore Multimax [4] and the Intel iPSC hypercube multiprocessors [21]. The Intel iPSC multiprocessor is a 80286 based multiprocessor with a hypercube interconnection network linking the nodes. The iPSC used in this investigation has 32 processors, each of which has 4.5 megabytes memory.

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. This programming methodology is further described in [12]. 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.

We present data from the Encore Multimax and the Intel iPSC that elucidates the effects of aggregation on multiprocessor performance. We first examine the execution time obtained using 14 processors from the forward solve of the zero fill factorization on a discretized L shaped region similar to that depicted in figure 8. A 100 by 20 point base is discretized using a 5 point template, on top of this lies a 80 by 80 point mesh discretized using a 9 point template. We aggregate using a window size equal to the block size, and display the execution times measured on the Multimax.

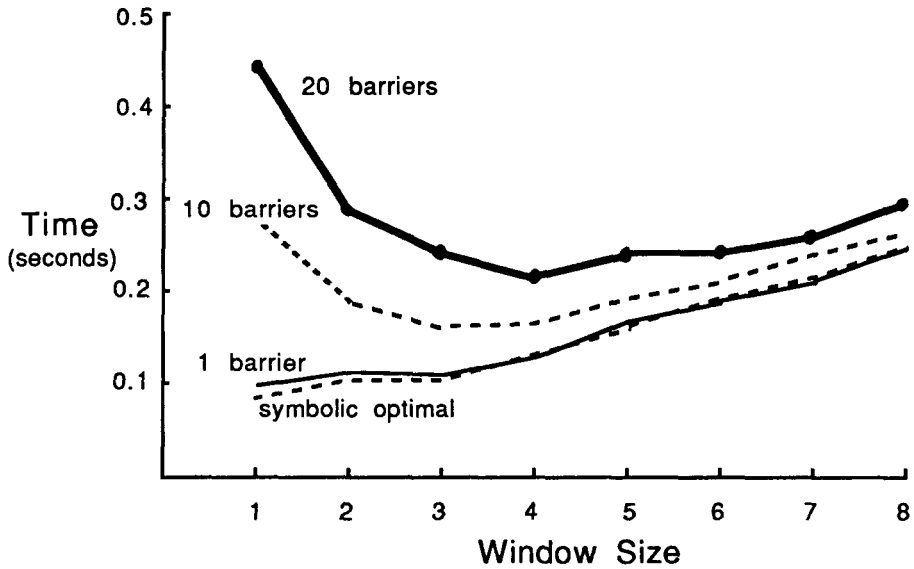


Figure 9: Effects of Window Size on Execution Time: Multimax

Note that the matrix obtained from this mesh is extremely sparse; there are no more than four non-zero off diagonal elements in any matrix row. The parallelism encountered here is consequently quite fine grained. This matrix, along with the others described here, has unit diagonal elements, so that the forward solve does not involve divisions.

Barrier (or global) synchronization between phases was utilized. When timed separately, this synchronization was found to require 46 microseconds; this compares to approximately 10.8 microseconds required for a single precision floating point multiply and add. It is not clear that future architectures utilizing much faster processors and more general interconnection networks will allow for synchronization costs that are as small relative to the costs of floating point computation. We consequently explored the performance effects of aggregation when either one, ten or twenty barriers were used for synchronization. In Figure 9 we see that the computation time as a function of aggregation is nearly convex and that the minimum time, as expected, occurs with greater degrees of aggregation as the cost of synchronization increases. The time required by a separate sequential program was 951.76 ms, consequently the optimum speedup was 9.49, 5.95 and 4.47 when one, ten and twenty barriers were used. We note also that the optimum performance is observed for increasingly aggregated problems as the cost of synchronization increases. Note also that performance is strongly determined by the choice of window size.

It is possible to symbolically estimate the optimal speedup in the absence of synchronization delays, given the assignment of work to processors characterizing a particular

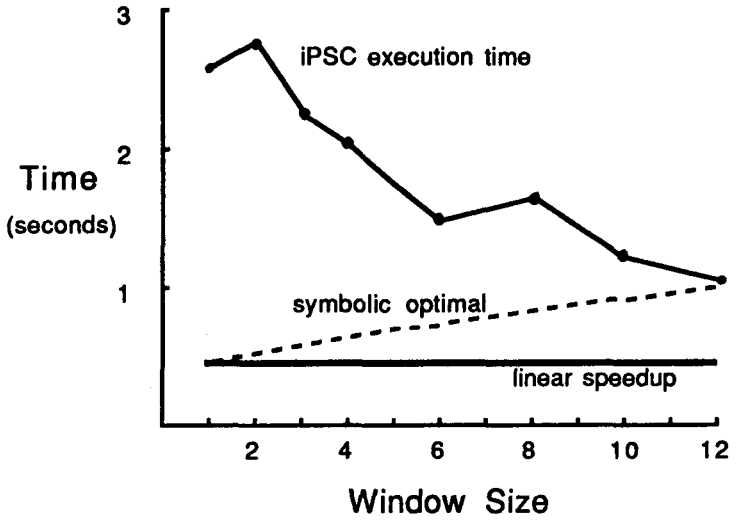


Figure 10: Effects of Window Size on Execution Time: iPSC

window and block size. For each window size, the time required for a separate sequential code to solve the problem was divided by the estimated optimal speedup. This yields the amount of time that would be required to solve the problem in the absence of any sources of inefficiency other than load imbalance. The results of these calculations are also plotted in Figure 9 where they are denoted as the symbolically estimated optimal computation time [24]. Note that the symbolically estimated optimal computation time increases with window size, as increases in granularity limit the available parallelism. The difference between the observed and estimated optimal times decreases with window size. Fewer phases are required to complete the computation and consequently the cost of synchronization becomes negligible.

While the plot in Figure 9 suggests convexity, distinctly non-convex performance curves are obtained by fixing the block size, and varying the window size. See [24] for a more extensive discussion of the this type of problem performance on the Multimax.

This type of forward solve problem was also implemented on an Intel iPSC. The iPSC version employed sixteen processors on a 160×160 point domain, discretized using a 5-point stencil. Figure 10 plots a roughly decreasing execution time as a function of degree of aggregation.

The window size was equal to the block size, and both were varied between one and twelve. The time required to perform a separate sequential calculation on one node of the iPSC was 7.198 seconds , the best speedup was consequently 6.381. Little further

advantage was noted when 32 processors were utilized. We note that the execution time *tends* to decrease as a function of increased aggregation. The local maxima at window sizes two and eight are non-intuitive, and were investigated further. While we have not found a completely satisfactory explanation for these enigma, it does appear that they are an operating system response to irregular and asynchronous message traffic. The iPSC has communication costs that are quite high, and consequently reducing the number of phases required to solve the problem has (generally) a very significant payoff. Once again we note the strong dependence of performance on aggregated work unit size.

We also depict the symbolically estimated optimal computation time in figure 10; observe that this increases with window size. While we were unable to obtain results from the parallel code for window sizes of greater than 12, it is clear that the execution time will increase with greater degrees of aggregation due to the reduction in available parallelism.

5 Battlefield Simulation

Unlike our first two model problems, the third is not specifically numerical. We consider a distributed battlefield simulation which was developed for the purpose of studying issues in the parallel processing of time-driven non-numerical simulations. The simulation is based on Zipscreen [7,9] which was developed by the BDM Corporation; in turn, Zipscreen is a simplified version of the CORBAN [8] simulation. Both Zipscreen and CORBAN view a battlefield as a two dimensional domain tessellated by hexagons. Combat units from opposing sides move through this domain. Zipscreen focuses on the *perception*, *combat*, and *movement* activities found in CORBAN at every time step. The perception activity is performed by every unit, and consists of creating a list of all enemy units on its own hex, and on adjacent hexes. The unit enters combat with units found on this list, calculates losses that it inflicts on enemy units, and reports those losses to the afflicted units. At the end of a time step, every unit moves, possibly changing hex locations.

A two-dimensional domain tessellated by hexagons can be viewed as a "rectangular" array of hexagons. This is seen in Figure 11 where the "rows" are clearly defined while the hexes in a "column" zig-zag vertically. As before, we will aggregate elemental units of the domain—the hexes. Aggregation consists of covering the domain with blocks each w hexes wide and h hexes tall (with the possibility of some deviation from these dimensions at the edges of the domain). These blocks themselves form a rectangular array that we index by "block row" and "block column". To assign the blocks using wrapping we view the processors as forming a r by c rectangular array of processors. Then, block (k, m) and all the hexes it contains is assigned to processor $(k \bmod r, m \bmod c)$.

It is not difficult to see that like the other two model problems, workload intensity is positively correlated in space. Computational activity is most intense where battles occur. Only units which lie on the same or adjacent hexes may battle each other. Furthermore, battles (and hence battlefield simulations) tend to be localized in space. The knowledge

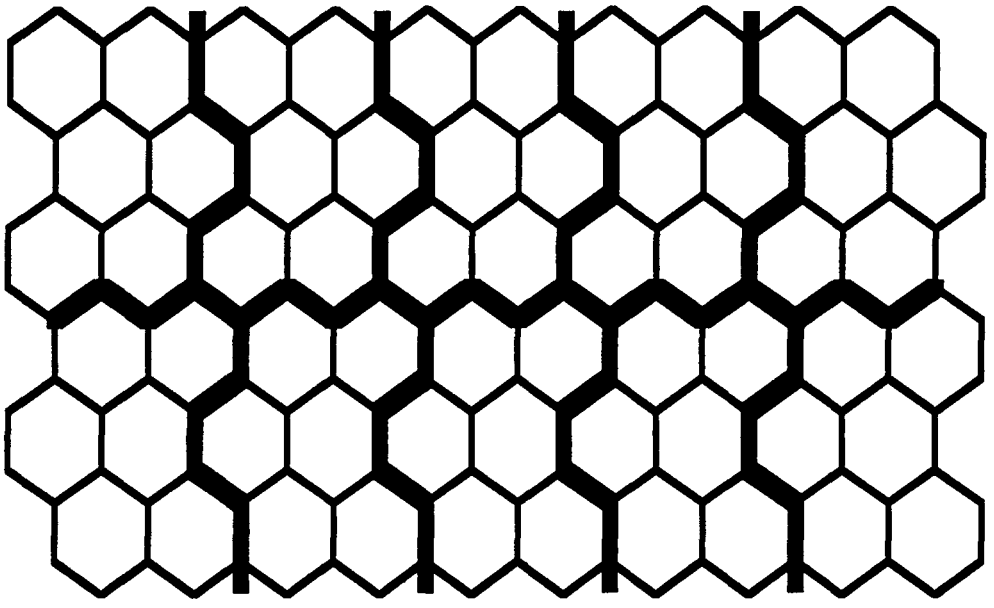


Figure 11: Rectangular Partitioning of Hexagon Domain

that a particular hex contains an engaged unit makes it likely that the hex lies in a region where the main battle-lines are drawn. It is interesting to note that even in this problem, the workload lies largely in one dimension—the battle-line.

The choice of h and w has a significant impact on the volume of inter-processor communication. All units lying on the border of a block must perceive any unit lying on adjacent hexes in another processor, must report any losses it inflicts on such units to another processor, and potentially moves from one processor to another. Decreasing either h or w increases the total number of hexes that lie on such boundaries, and increases the average volume of communication. On the other hand, decreasing h and w improves load balance, so once again we have the load balance/overhead trade-offs controlled by the aggregation parameters.

Zipscreen is implemented on NASA Langley's Flex/32 using a message-passing paradigm. Once again the global memory was used only to support message-passing. The results of the experiment reported below are representative of many performed on the Flex/32. We place 1000 units, 500 to a side, on a 32×32 hex domain. Sixteen processors are used. The initial placement separates the two sides with a diagonal line; all units are initially within a few hex's of the diagonal. Within these parameters the units are placed randomly. A unit has one direction of movement (six directions corresponding to the six hex sides are possible). Most units are randomly directed to move in a direction towards the opposing side (three directions possible); movement speeds are set so that a unit remains on a hex

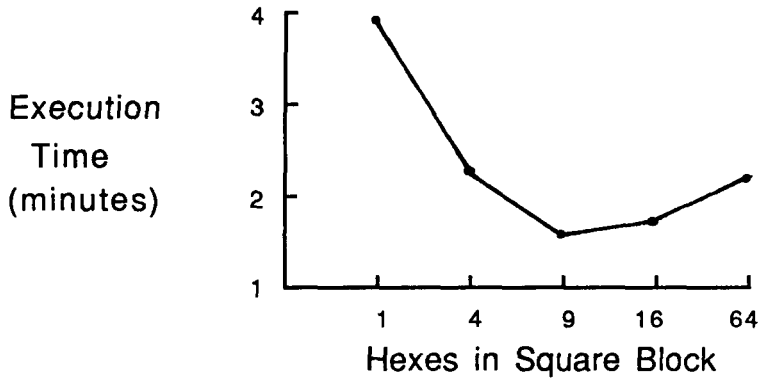


Figure 12: Battlefield Simulation Performance

for approximately five time-steps. A small fraction of the units are directed to remain stationary. The simulation runs for fifty time-steps. Units which vector off of the domain simply disappear. Heavy combat simulation occurs in the early time-steps, when the opposing sides engage. Computational costs dominate the run-time in this period. During later time-steps the two sides have largely passed through each other, and combat is limited to skirmishes between stationary units of one side, and moving units of the opposing side. The run-time of this latter period is dominated by communication. This scenerio is intended more to test the mapping during different types of problem behavior than it is intended to reflect actual battlefield simulations. More realistic battlefield simulations require a more intelligent movement mechanism than that found in Zipscreen. Further details concerning these experiments are reported in [15,16].

Figure 12 graphs performance of a simulation on a 32×32 hex domain, using sixteen processors. The execution time (in minutes) is plotted as a function of aggregated hex size assuming that the aggregated blocks are square with sides 1,2,3,4, and 8 hexes in length. Like the other model problems, the tradeoffs between load balance and communication are quite clear. Speedups using sixteen processors tend to be about 8; speedups using eight processors tend to be about 5.5. We expect that larger problems would improve the sixteen processor speedups, but memory constraints keep us from verifying this. These speedups are actually quite reasonable considering that the mapping is static, and the workload is highly variable. In fact, we will later discuss measurements indicating that the load is fairly well balanced; our speedup measurements reflect the fact that a significant amount of time is spent in inter-processor communication.

6 Analytic Model

Each of our model problems have workloads that at a given phase can be viewed as one-dimensional. We will develop a one-dimensional analytic model which formalizes some of our intuition about wrapping and its affects on performance. Specifically, we give stochastic arguments showing that among a class of "balanced" mappings, wrapping minimizes the variance of a processor's workload. The implications of this result are (i) that decreasing the work unit size better balances a workload, and (ii) that among all mapping strategies, wrapping offers some useful analytic properties. Then we look at the effects on processor variance/covariance achieved by decreasing the extent of aggregation (or equivalently, increasing the number of work units). The model predicts that inter-processor load correlation approaches unity in the square of the number of work units. While we have not established a general analytic relationship between load correlation and load balance under wrapping, we do give heuristic arguments and discuss some limited analytic and empirical results concerning this relationship.

Performance curves of the type seen in figures 7, 9, 10, and 12 can be viewed as the sum of an execution cost curve with a communication/synchronization cost curve. The latter curve is quite dependent on the architecture, while the qualitative effects of load imbalance are largely machine independent (provided that the processors are homogeneous). The analysis in this section concerns only the execution cost curve, and factors which affect it. By coming to understand how load balance in isolation is affected by the aggregation decision, we are better able to understand the tension between load imbalance and communication/synchronization overheads.

6.1 Model Preliminaries

Consider the behavior of a computation on a real line interval during one phase; without loss of generality we assume that the computation is performed on $[0,1]$. While an actual computation will discretize this interval, for the sake of modeling ease we will assume that every point $p \in [0,1]$ has a certain *work intensity* associated with it. Furthermore, we suppose that the computation advances in time (or phases), and that the work intensity at point p depends on "time". Recall that all of our model problems exhibit a sense of time. Let $W_t(p)$ denote the execution intensity associated with computation at point p, at time t. To determine the execution time required by the computation over the region $[a,b]$ at time t we simply integrate $W_t(p)$ from $p = a$ to $p = b$. We assume that the intensities $W_t(p)$ are unknown, but we are willing to model our uncertainity by assuming that $W_t(p)$ is a random variable, and that $W_t(p)$ can be viewed as a second order stationary process [22] over $p \in [0,1]$. Thus we suppose that $E[W_t(p)] = \mu(t)$ for all $p \in [0,1]$, that $var[W_t(p)] = \sigma^2(t)$ for all $p \in [0,1]$, and that $Cov[W_t(p), W_t(q)]$ depends only on $|p - q|$. These assumptions are reasonable if we are unwilling or unable to differentiate between the likely behavior of the computation at p and at q. We do *not* assume that $W_t(p) = W_t(q)$,

we simply assume that we have the same degree of uncertainity about $W_t(p)$ and $W_t(q)$.

The relationship between the work intensity at point p, and the intensity at a nearby point q is of considerable interest. It makes intuitive sense that these intensities for nearby points be positively correlated—positive correlation is a way of modeling spatial locality in workload intensity. Furthermore, it seems intuitive that the strength of the correlation should diminish with the distance between p and q. Figure 13(a) shows the measured autocorrelation function [20] at one time-step in the fluids model problem (function values for large distances are not shown, but the trend shown in the figure continues). The autocorrelation $a(d)$ function for a second order stationary process is a statistical estimate of correlation in points d units distant from each other. Not only do we observe diminishing correlation as a function of distance, we also observe that correlation between nearby points can reasonably be modeled as a linear function of distance. Figure 13(b) shows the measured correlation between blocks at two different phases in the triangular system solution example of Section 4. Again we see locally decreasing correlation as a function of distance (between blocks), but should be aware that the sizes of the sample sets defining each autocorrelation value are quite small.

Our intuition and measurements lead us to the following model of correlation. We let α be some positive number less than two, and assume that the covariance between $W_t(p)$ and $W_t(q)$ is given by

$$Cov[W_t(p), W_t(q)] = \sigma^2(t)(1 - \alpha|p - q|).$$

For the purpose of tractability we are assuming that correlation *throughout the domain* is a linear decreasing function of the distance.

The execution time associated with doing all the work in a subinterval $[a, b]$ at time t is

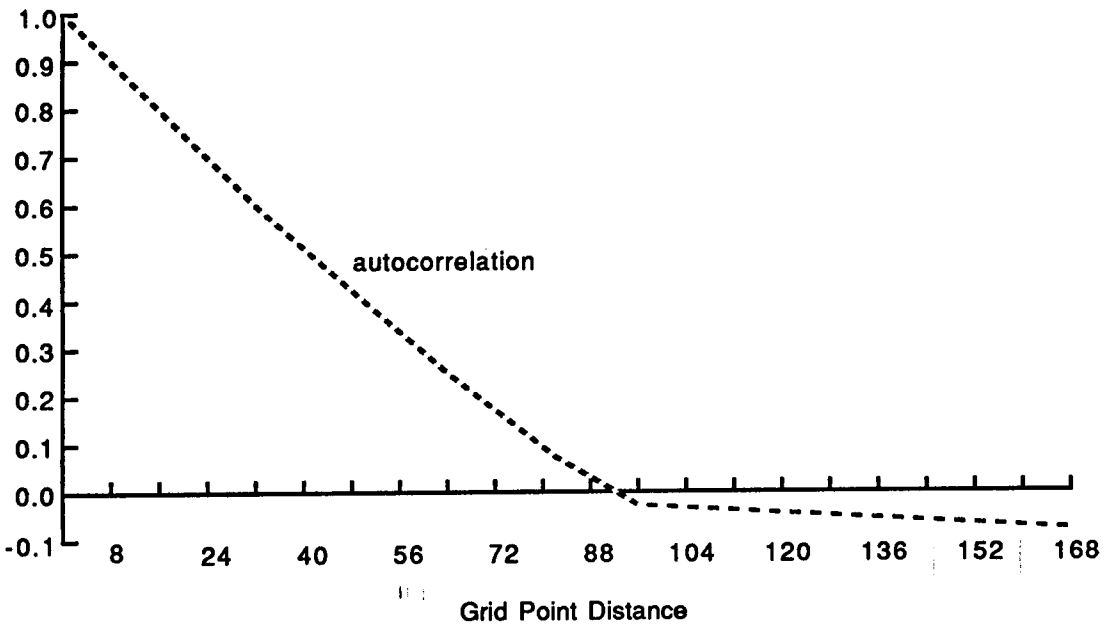
$$T(a, b, t) = \int_a^b W_t(p)\ dp.$$

$T(a, b, t)$ has mean value $(b - a)\mu(t)$. The variance of $T(a, b, t)$ is derived by noting first that

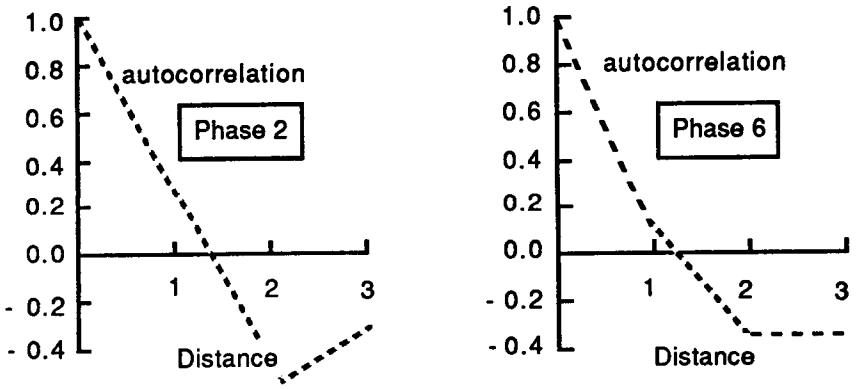
$$\begin{aligned} var[T(a, b, t)] &= E[(T(a, b, t) - (b - a)\mu(t))^2] \\ &= E[\hat{T}(a, b, t)^2] \end{aligned}$$

where $\hat{T}(a, b, t)$ is the integral from a to b of $\hat{W}_t(p) = W_t(p) - \mu(t)$, a stochastic process with mean 0 and identical covariance structure as $W_t(p)$. The variance of the workload interval is given by

$$\begin{aligned} var[T(a, b, t)] &= E[\int_a^b \hat{W}_t(p)\ dp \int_a^b \hat{W}_t(q)\ dq] \\ &= \int_a^b \int_a^b E[\hat{W}_t(p)\hat{W}_t(q)]\ dq\ dp \end{aligned}$$



(a) Autocorrelations from Fluids Problem



(b) Autocorrelations from Triangular Solve Problem

Figure 13: Correlation in Model Problem's Workload

$$
\begin{aligned}
&= \int_a^b \int_a^b Cov[\hat{W}_t(p), \hat{W}_t(q)]\, dq\, dp \\
&= \int_a^b \int_a^b \sigma^2(t)(1-\alpha|p-q|)\, dq\, dp \\
&= \sigma^2(t)(b-a)^2\,(1-\alpha(b-a)/3)\,.
\end{aligned}
$$

Following a uniform aggregation of the domain points into n work units, the ith work unit is the integral $T(i/n,(i+1)/n,t)$, and is denoted as c_i. [2] The random vector of work units is denoted $C =< c_0,\ldots,c_{n-1} >$. We are interested in the covariance matrix σ_C^2 for the work units. Using an approach similar to the one outlined above, we find that the covariance between units i and j is

$$Cov[c_i,c_j] = (\sigma_C^2)_{ij} = \frac{\sigma^2(t)}{n^3}(n-\alpha|i-j|). \tag{1}$$

Similarly, the variance of work unit i is given by

$$var[c_i] = (\sigma_C^2)_{ii} = \frac{\sigma^2(t)}{n^3}(n-\alpha/3). \tag{2}$$

At this point it is worthwhile to relate our analytic model to the three model problems. The length of a work unit, $1/n$, directly models the extent of aggregation. The value of c_i models the execution time of entities such as (i) subintervals in fluids problem domain, (ii) an aggregated work unit in the triangular solve problem, (iii) a sequence of hexes along the battle-line in the battlefield simulation problem. The randomness of c_i models the uncertainty in execution time of each of these entities; the distance dependent decreasing correlation between c_i and c_j models the locality of workload intensity we have noted in each of the model problems. With these ideas in place we turn to the problem of effectively mapping the work units onto processors.

For simplicity we assume that the number of processors, P, divides n evenly. A convenient way to describe an assignment of work units to processors is by a $P \times n$ *assignment* matrix whose $ij-th$ entry is 1 if work unit c_j is assigned to processor i, and is 0 otherwise. Given assignment matrix $\mathcal{A}$, the multiplication $\mathcal{A}C$ yields a $P \times 1$ random vector whose jth component is the sum of the execution times of all work units assigned to processor j. The vector of mean processor loads is the matrix-vector product $\mathcal{A}[\mu(t)/n]$, where $[\mu(t)/n]$ is the vector of C's means. The covariance matrix of $\mathcal{A}C$ is the product $\mathcal{A}\sigma_C^2\mathcal{A}^T$, where $\mathcal{A}^T$ is the transpose of $\mathcal{A}$.

Under the assumption that all work during one phase must be completed before any work in the next phase is begun, the phase execution time (ignoring any communication overhead) is the maximum processor execution time, or $\max\{(\mathcal{A}C)^T\}$; a random quantity since C is a random vector. We are interested in finding an assignment matrix which minimizes the execution time in some sense. It is natural then to look for the assignment which minimizes the *expected* value of $\max\{(\mathcal{A}C)^T\}$.

[2] c_i's dependence on time t is dropped here for notational convenience.

6.2 Correlation and Load Balance

We will restrict our attention to balanced assignments that assign an equal number of work units to each processor. There are extreme conditions when a balanced assignment will not minimize $E[\max\{(\mathcal{A}C)^T\}]$; for example, if $W_t(p)$ is Gaussian with $\sigma(t)/\mu(t)$ large and $\mu(t)$ close to zero, it may be optimal to put all work in one processor. However, these conditions also require an interpretation of negative work, which has no meaning here. We will avoid these problems by simply assuming that $\sigma(t)/\mu(t)$ is small, and that $\mu(t)$ is large. There are many balanced assignments; in our assumed framework, two balanced assignments need not have identical expected maximum processor finishing times.

The correlation between processor workloads under a balanced assignment has a strong impact on $E[\max\{(\mathcal{A}C)^T\}]$. One extreme case is when the correlation is exactly 1, so that every processor has exactly the same execution time. In this case the expected maximum processor execution time is identical to the mean processor execution time. A moment of reflection reveals that the expected maximum can get no smaller. At the other extreme consider two processors whose loads have a correlation coefficient of -1. Whenever one processor's load is r above the mean load, the other's is r below the mean load. The expected maximum between the two is large, since one of the loads is guareenteed to exceed the mean. It is clear then that maximized correlation between workloads is good, in that it reduces the idle time of processors waiting for the most heavily loaded processor to finish the phase. We should therefore attempt to find the assignment matrix that in some sense maximizes inter-processor covariance. It follows from the definition of the processor load covariance matrix that the sum of all matrix entries is equal to the sum of all work unit covariance matrix entries;

$$\sum_{i=0}^{P-1}\sum_{j=0}^{P-1}\left(\mathcal{A}\sigma_C^2\mathcal{A}^T\right)_{ij} = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1}(\sigma_C^2)_{ij}. \tag{3}$$

This relationship holds independently of the assignment matrix $\mathcal{A}$. Thus, to maximize the sum of inter-processor covariance terms we need only minimize the sum of processor variances. We are able to show that a wrapped assignment of the n work units minimizes the sum of processor variances, and so can be expected to achieve a small value of $E[\max\{(\mathcal{A}C)^T\}]$.

6.3 Minimizing Sum of Processor Variances

To show that a wrapped assignment minimizes the sum of processor load variances we demonstrate a procedure that takes any assignment and constructs another whose sum of processor load variances is no larger. The repeated application of this procedure produces a wrapped assignment.

Let $\mathcal{A}_1$ be any assignment matrix describing a balanced assignment. Without loss of generality, we assume that under $\mathcal{A}_1$ the processors are numbered so that P_0 is assigned

c_0, P_1 is assigned the smallest indexed c_i that is not assigned to P_0, and in general P_j is assigned the smallest indexed c_k that is not assigned to any of $P_0, P_1, \ldots, P_{j-1}$. The labels P_j will be used to both identify a processor, and the workload assigned to that processor.

We will say that c_j is *in place* if c_j is assigned to processor $P_{j \bmod P}$. Note that all work units are in place under a wrapped assignment. We construct another balanced assignment $\mathcal{A}_2$ by finding the smallest indexed c_i that is not in place, and by putting it in place. Let c_f denote this unit, let P_S denote the *source* processor which has c_f under $\mathcal{A}_1$, and let P_T denote the *target* processor $P_{f \bmod P}$. Let c_g be the smallest indexed work unit assigned to P_T such that $g > f$. $\mathcal{A}_2$ is constructed from $\mathcal{A}_1$ by giving c_f to P_T, and c_g to P_S. We will prove that the sum of processor variances under $\mathcal{A}_2$ bounds that sum under $\mathcal{A}_1$ from below.

For any processor P_i, let $\mathcal{A}(i)$ denote the set of work units assigned to it under $\mathcal{A}$. By definition the variance of P_i's work load is given by

$$var[P_i] = \left(\mathcal{A}\sigma_C^2\mathcal{A}^T\right)_{ii} = \sum_{c_j \in \mathcal{A}(i)} var[c_j] + \sum_{<c_j,c_k> \in \mathcal{A}(i)\times\mathcal{A}(i)} Cov[c_j, c_k]. \tag{4}$$

The sum of work unit variances depends only on the number of units assigned to P_i. The second component of the expression above is a sum of *unit covariance terms* (*uc* terms), that depends on the assignment chosen. Similarly, the covariance between processors P_i and P_j is given by a sum of *uc* terms:

$$\begin{aligned} Cov[P_i, P_j] &= \sum_{<c_k,c_m> \in \mathcal{A}(i)\times\mathcal{A}(j)} Cov[c_k, c_m] \\ &= \sum_{<c_m,c_k> \in \mathcal{A}(j)\times\mathcal{A}(i)} Cov[c_m, c_k] \\ &= Cov[P_j, P_i]. \end{aligned} \tag{5}$$

It is clear from (4) that the variance of any processor other than P_S or P_T is by unaffected by swapping c_f and c_g. To prove the desired result we need only show that the swap does not increase $var[P_S] + var[P_T]$. The change in processor variances caused by the swap is entirely due to changes in the sum of *uc* terms in each processor. After swapping c_f and c_g, each work unit c_i assigned to P_S loses the *uc* term $Cov[c_f, c_i]$ and gains the term $Cov[c_g, c_i]$. We let Δ_{L_S} denote the sum of all such changes among work units in P_S less than (or to the left of) f, and let L_S denote the number of such work units. Similarly Δ_{R_S} denotes the sum of changes among work units in P_S greater than (or to the right of) g and R_S denotes the number of such units; Δ_{M_S} denotes the sum of changes among units in P_S with indices between f and g. Expressions for these quantities are derived using equation 1:

$$\Delta_{L_S} = \sum_{\substack{c_i \in \mathcal{A}_2(S) \\ i<f}} (Cov[c_g, c_i] - Cov[c_f, c_i]) = -\frac{\sigma^2(t)}{n^3}(g-f)L_S\alpha;$$

$$\Delta_{R_S} = \sum_{\substack{c_j \in A_2(S) \\ j>g}} (Cov[c_g, c_j] - Cov[c_f, c_j]) = \frac{\sigma^2(t)}{n^3}(g-f)R_S\alpha;$$

$$\Delta_{M_S} = \sum_{\substack{c_k \in A_2(S) \\ f<k<g}} (Cov[c_g, c_k] - Cov[c_f, c_k]) = \frac{\sigma^2(t)}{n^3} \sum_{\substack{c_k \in A_2(S) \\ f<k<g}} (2k - f - g)\alpha.$$

The change in P_S's variance after the swap is the sum $\Delta_{L_S} + \Delta_{M_S} + \Delta_{R_S}$.

We can similarly describe the change in P_T's variance with the definitions

$$\Delta_{L_T} = \sum_{\substack{c_i \in A_2(T) \\ i<f}} (Cov[c_f, c_i] - Cov[c_g, c_i]) = \frac{\sigma^2(t)}{n^3}(g-f)L_T\alpha;$$

$$\Delta_{R_T} = \sum_{\substack{c_j \in A_2(T) \\ j>g}} (Cov[c_f, c_j] - Cov[c_g, c_j]) = -\frac{\sigma^2(t)}{n^3}(g-f)R_T\alpha.$$

No term analogous to Δ_{M_S} is necessary since there are no work units in P_T with indices between f and g.

The change in the sum of P_S's variance with P_T's variance is given by the sum of all the Δ terms defined above. We will show that the sum of Δ terms is bounded from above by 0. At this point a number of observations are useful. Since all c_i with $i < f$ are in order, it follows that $L_T \leq L_S$. Thus $\Delta_{L_S} + \Delta_{L_T} \leq 0$. It remains to show that $\Delta_{R_S} + \Delta_{R_T} + \Delta_{M_S} \leq 0$. We know that

$$\Delta_{R_S} + \Delta_{R_T} = -\frac{\sigma^2(t)}{n^3}(R_T - R_S)(g-f)\alpha; \tag{6}$$

furthermore, since $n/P = L_T + R_T$, we must also have $R_S \leq R_T$. We proceed to show that the magnitude of Δ_{M_S} is no greater than the magnitude of (6) and consequently prove the larger result.

$m = n/P - L_S - R_S$ is the number of work units in P_S whose indices lie strictly between f and g. Δ_{M_S} is maximized when the indices of these units are as large as possible; when $k = g-1, g-2, \ldots, g-m$. With such indices, the sum of c_g's uc terms in P_S is

$$\frac{\sigma^2(t)}{n^3} \sum_{i=1}^{m} (n - i \cdot \alpha).$$

Likewise, the sum of c_f's uc terms in P_S is

$$\frac{\sigma^2(t)}{n^3} \sum_{i=1}^{m} (n - (g - f - i)\alpha).$$

From this, we see that Δ_{M_S} when maximized can be written as

$$\Delta_{M_S} = \frac{\sigma^2(t)}{n^3} \sum_{i=1}^{m} (n - i \cdot \alpha) - \frac{\sigma^2(t)}{n^3} \sum_{i=1}^{m} (n - (g - f - i)\alpha) = \frac{\sigma^2(t)}{n^3} m(g-f)\alpha.$$

But note that

$$\begin{aligned} m &= n/P - L_S - R_S \\ &\leq n/P - L_T - R_S \\ &= n/P - L_T - R_T + (R_T - R_S) \\ &= (R_T - R_S), \end{aligned}$$

so that

$$\Delta_{R_S} + \Delta_{R_T} + \Delta_{M_S} = \frac{\sigma^2(t)}{n^3}(-(R_T - R_S)(g-f)\alpha + m \cdot (g-f)\alpha) \leq 0.$$

Consequently, swapping c_f and c_g does not increase $var[P_S] + var[P_T]$. Furthermore, the swap does not affect the sum of other processors' variances. Repeatedly applying this procedure leads to the assignment where every work unit is in place, an assignment we will call the *perfect* assignment. This discussion has proved the following theorem.

Theorem 1 *Let P and n be given such that P divides n evenly, and let $\mathcal{A}_P$ be the perfect $P \times n$ assignment matrix. Then for any $P \times n$ assignment matrix $\mathcal{A}$,*

$$\sum_{i=0}^{P-1} (\mathcal{A}_P \sigma_C^2 \mathcal{A}_P{}^T)_{ii} \leq \sum_{i=0}^{P-1} (\mathcal{A} \sigma_C^2 \mathcal{A}^T)_{ii}$$

□

Theorem 6.1 is powerful in its generality. One of its immediate implications is that among balanced workload assignments which also balance processor variances, the perfect assignment minimizes that common processor variance. We have offered heuristic reasons explaining why minimizing processor variance should keep the expected load of the busiest processor low. But so far, the link between low processor variance and good load balance rests just on heuristic reasoning. In at least one special case it is possible to show that a perfect assignment produces the stochastically optimal load balance. The conditions leading to this result are that the workload intensities $W_i(p)$ form a Gaussian stochastic process [13] and that the number of processors is two. Then it is straightforward to show that for any n, the vector of n work units C has a jointly normal distribution (see [13] for a definition of joint normality) and that under *any* assignment, the two processors' workloads are jointly normal. The following lemma concerning bivariate normal distributions can be proven using first principles in probability.

Lemma 2 *Let random variables P_1 and P_2 be jointly normal with mean vector* $\begin{bmatrix} \mu \\ \mu \end{bmatrix}$ *and covariance matrix* $\begin{bmatrix} \sigma_1^2 & \tau \\ \tau & \sigma_2^2 \end{bmatrix}$. *Then*

$$E[\max\{P_1, P_2\}] = \mu + \left(\frac{\sigma_1^2 + \sigma_2^2 - 2\tau}{2\pi}\right)^{1/2}.$$

□

Since relation (3) is independent of the assignment and wrapping minimizes the sum of processor variances, it follows from Lemma 6.1.1 that in at least one case, wrapping optimally balances the load (at least among balanced assignments). It is not clear at this time whether this result extends to general numbers of processors. But, Theorem 6.1 does offer a strong theoretical reason for choosing the wrapping paradigm.

6.4 Aggregation and Load Balance

If we accept the supposition that minimizing processor variance reduces phase execution time, then our results imply that we should aggregate as little as possible for the best balance of load. This is in complete agreement with intuition, and is further illustrated by example. Suppose that the finest possible granularity leads to n work units. A coarser granularity with $n/2$ work units is achieved by combining the leftmost two work units into one, the next two work units into one, and so on. In our analytic model any assignment $\mathcal{A}$ of the doubled-units is statistically identical with that assignment of the fine work units which assigns the first two units to the processor holding the first doubled-unit, the third and fourth units to the processor holding the second doubled-unit, etc. Our results say that any assignment of the doubled-units is inferior to the wrapped assignment of the finest grained units. Consequently, to achieve the best load balance (using wrapping) we should aggregate as little as possible.

This conclusion is borne out by measurements made on the battlefield simulation. Zipscreen conveniently lends itself to measuring load balance (largely) in isolation, as each time step is composed of a computational period (the perception and combat activities), followed by a communication period (movement and exchange of relevant block boundary information). Our metric is the average processor utilization during the computational period, taken to be the average time spent doing computation, divided by the time spent by the last processor to finish that period. Figure 14 plots average efficiency as a function of work unit size. The measurements are averaged over a large set of computational periods. As expected, low degrees of aggregation produce better efficiencies. In fact, the performance due to load balancing is somewhat better than represented for low degrees of aggregation as these measurements still include a certain amount of additional overhead.

6.5 Covariance Structure Under Wrapping

It is important to remember that our analytic results have only concerned the effects of load balancing on performance—we have ignored the additional overhead of communication and synchronization. If decreasing the size of work units improves load balance while increasing overhead, we would like to know how much aggregation can be tolerated and still achieve reasonable balances of load. The analysis in this section helps to clarify this issue by studying the behavior of processor variance/covariance under wrapping, as we aggregate less and less. We then link our results about variance/covariance to the

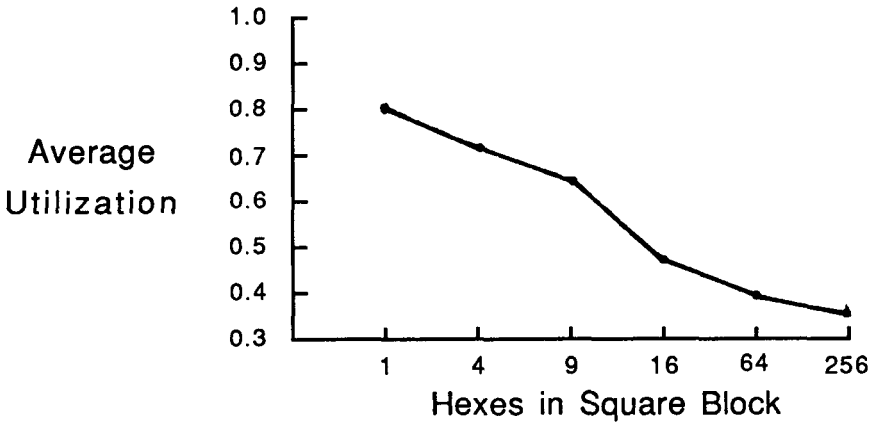


Figure 14: Computational Efficiency as Function of Aggregation

special case described by Lemma 6.1.1, and see there that load balance improves linearly in the number of work units. We then observe somewhat better response to decreasing aggregation in our battlefield simulation problem.

The covariance matrix for a set of random variables that are perfectly correlated has identical entries in every matrix position. Optimal performance is achieved if the processors' loads are perfectly correlated (although this is impossible under our analytic model). The analysis to follow shows what the single limiting matrix value would be for perfectly correlated loads, and that as the number of work units n increases, the difference between an arbitrary covariance matrix entry and the limiting value decreases as a function of n^2.

Every processor has the same variance in a perfect assignment; as seen before, this variance has two components. The sum of work unit variances is just

$$\sum_{i=1}^{n/P} \frac{\sigma^2(t)}{n^3}(n - \alpha/3) = \sigma^2(t)\left(\frac{1}{nP} - \frac{\alpha}{3n^2P}\right). \tag{7}$$

We now consider the sum of uc terms which also contribute to the processor variance. Under a perfect assignment, when c_m and c_k are assigned to the same processor, then

$$Cov[c_k, c_m] = \frac{\sigma^2(t)}{n^3}(n - jP\alpha)$$

where $j = |k - m|/P$, which is an integer. Within a processor, the work unit with the kth largest index ($k = 1, \ldots, n/P$) has uc terms with $k - 1$ other work units having smaller indices. Likewise, it has uc terms with $n/P - k$ work units having larger indices. For fixed k, the sum of these two groups is

$$S_k = \frac{\sigma^2(t)}{n^3}\left(\sum_{j=1}^{k-1}(n - jP\alpha) + \sum_{i=1}^{n/P-k}(n - iP\alpha)\right).$$

We are interested in the sum of all such groups, found by straightforward algebra:

$$\sum_{k=1}^{n/P} S_k = \frac{\sigma^2(t)}{n^3}\left(\frac{n^3}{P^2} - \frac{\alpha n^3}{3P^2} - \frac{n^2}{P} + \frac{\alpha n}{3}\right).$$

A processor's complete variance is found by adding this quantity to the sum given by equation (7)

$$\begin{aligned} Var[P_i] &= \frac{\sigma^2(t)}{n^3}\left(\frac{n^3}{P^2}(1-\alpha/3) + \frac{\alpha n}{3}(1-1/P)\right) \\ &= \sigma^2(t)\left(\frac{1}{P^2}(1-\alpha/3) + \frac{\alpha}{3n^2}(1-1/P)\right). \end{aligned} \tag{8}$$

We can determine $Cov[P_i, P_j]$ under a perfect assignment in an entirely similar fashion. Without loss of generality we suppose that $i < j$. The work unit in P_i with the kth largest index has $k-1$ uc terms with work units in P_j having smaller indices. The sum of all these uc terms is

$$\frac{\sigma^2(t)}{n^3}\sum_{m=1}^{k-1}(n - ((j-i) + (m-1)P)\alpha).$$

Similarly, it has $n/P - k + 1$ uc terms with work units in P_i having larger indices. The sum of these terms is

$$\frac{\sigma^2(t)}{n^3}\sum_{m=1}^{n/P-k+1}(n - (n/P - (j-i) + (m-1)P)\alpha).$$

Accumulating these two sums over $k = 1$ to $k = n/P$ it is straightforward to derive

$$\begin{aligned} Cov[P_i, P_j] &= \frac{\sigma^2(t)}{n^3}\left(\frac{n^3}{P^2} - \frac{\alpha n^3}{3P^2} + \frac{\alpha n}{3} - \frac{n(j-i)\alpha}{P}\right) \\ &= \sigma^2(t)\left(\frac{1}{P^2}(1-\alpha/3) + \frac{\alpha}{3n^2} - \frac{(j-i)\alpha}{Pn^2}\right). \end{aligned} \tag{9}$$

From equations (8) and (9) we see that as n gets large, the covariance matrix entries all approach the value $(\sigma^2(t)/P^2)(1-\alpha/3)$. This is to be expected; as n increases, the "distance" between two processors' work units becomes smaller and smaller, so that the correlation between their work units becomes higher. Furthermore, the convergence towards statistical identity is fairly rapid, as the diminishing terms in (8) and (9) decrease in n^2.

Figure 14 shows that load balancing tends to degrade slowly as the degree of aggregation increases. Most of the gain achievable by low degrees of aggregation are obtained with significant aggregation, and hence lower overhead. Accepting our analytic model's description of processor load variance, this means that the quadratic convergence of covariance towards statistical identity is coupled with a super-linear convergence of average

processor utilization towards its optimal value. Less impressive results are derivable from Lemma 6.1.1. The value τ is processor covariance, predicted by our model to approach its maximum as a square function of the number of work units. Lemma 6.1.1 implies that the expected maximum then declines as a *linear* function of the number of work units, as τ appears within a radical. The symbolic optimal execution times depicted in Figures 9 and 10 decrease roughly linearly as the number of work units increases; note however that those calculations relate to an entire computation, not just one phase. While these observations shed some light on the relationships between degree of aggregation, processor variance, and load balance, it is important to bear in mind that such relationships are likely to be problem dependent.

6.6 The Role of Time

The derivations in previous subsections explicitly denoted the dependence of the expected phase finishing time $E[\max\{(\mathcal{A}_P C)^T\}]$ on "time", or phase number. Theorem 6.1 concerns only one phase, and typically we are concerned with finding a single assignment for many phases. However, if the degree of aggregation is chosen before running the computation, then our results say that the processor variance *at every phase* is minimized by wrapping. This does not necessarily mean that the variance of the sum of a processor's phase execution times is minimized, although this would be true if the phases were probabilistically independent. Our analytic model might be extended to treat workload correlation *in time*, and could potentially give further insights into the properties of a wrapped assignment over the entire computation, not just one phase.

7 Summary

A large class of computations exhibit a high degree of fine-grained parallelism, but have execution requirements that are either unpredictable, or (because of the fine-grained nature of the problem) are too costly to pre-analyze. The parallelism inherent in these problems suggests that parallel processing can be used, but the uncertainity in the workload makes mapping that workload onto medium-scale multiprocessors a difficult problem. This paper advocates simple principles for aggregating and mapping workload under these conditions. The aggregation is defined by a small number of parameters. These parameters effectively control the trade-offs between the good load balance achieved by little aggregation, and the low communication/synchronization costs achieved by extensive aggregation. We illustrated these principles and trade-offs using three real-life model problems, implemented on different multiprocessors. The performance observed shows that aggregation and wrapping is a viable method for mapping workload onto multiprocessors; dynamic problems of the sort studied here are generally recognized as being difficult to map. We also developed an analytic model which explains in part the inter-relationship between aggregation and load balance, and which shows that the mapping algorithm we study minimizes the variance in

processors' loads. Conclusions drawn from the model are shown to be in agreement with observations from our model problems.

Perhaps the most important question we have left unanswered is how one chooses the aggregation parameters. When the computation's behavior and costs at every step are completely known, then it may be possible to optimally pre-schedule aggregation changes [19]. We are currently considering schemes that attempt to dynamically estimate and adjust granularity, and will address this important issue in a subsequent paper.

Acknowledgements: Discussions with Marina Chen were very useful; we also thank Kay Crowley and Frank Willard for their programming help. Without Phil Kearns we would not have been able to insert Postscript figures directly into this document. We are also indebted to Bob Voigt for his support.

References

[1] M. J. Berger and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics*, 53:484–512, 1984.

[2] D. L. Book, editor. *Finite-Difference Techniques for Vectorized Fluids Dynamics Calculations.* Springer-Verlag, New York, 1981.

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

[4] *Multimax Technical Survey.* Technical Report 726-01759 Rev A, Encore Computer Corporation, 1986.

[5] G.C. Fox and S. W. Otto. *Concurrent Computation and the Theory of Complex Systems.* Technical Report CALT-68-1343, Caltech Concurrent Computation Program, 1986.

[6] G. A. Geist and M. T. Heath. Matrix factorization on a hypercube multiprocessor. In *The Proceedings of the Hypercube Micropro*, pages 161–180, September 1986.

[7] J.B. Gilmer. *Documentation, State-Space Reconciliation Version of the Zipscreen Prototype Simulation.* Technical Report, BDM Corporation, 1986.

[8] J.B. Gilmer. *Statistical Measurements of the CORBAN Simulation to Support Parallel Processing.* Technical Report BDM/ROS-86-0326, BDM Corporation, 1986.

[9] J.B. Gilmer and J.P. Hong. Replicated state-space approach for parallel simulation. In *Proceedings of the 1986 Winter Simulation Conference*, Washington, D.C., 1986.

[10] G. Golub and C. F. Van Loan. *Matrix Computations.* Johns Hopkins University Press, Baltimore Maryland, 1985.

[11] I. Ipsen, Y. Saad, and M.H. Schultz. Complexity of dense linear system solution on a multiprocessor ring. *Lin. Algebra Appl.*, 77:205–239, 1986.

[12] J. F. Jordan, M. S. Benten, and N. S. Arenstorf. *Force User's Manual.* Department of Electrical and Computer Engineering 80309-0425, University of Colorado, October 1986.

[13] H.J. Larson and B.O. Shubert. *Probabilistic Models in Engineering Sciences.* Volume 1, Wiley, New York, 1979.

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

[15] D.M. Nicol. *Mapping domain-oriented time-driven simulations onto message-passing parallel architectures.* Technical Report 87-51, ICASE, September 1987. To appear in the Proceedings of the 1988 SCS Conference on Distributed Simulation, San Diego, February 1988.

[16] D.M. Nicol. Performance issues for domain-oriented time-driven distributed simulations. Technical Report 87-49, ICASE, August 1987. To appear in the Proceedings of the 1987 Winter Simulation Conference, Atlanta, December 1987.

[17] D.M. Nicol and P.F. Reynolds Jr. *Optimal Dynamic Remapping of Parallel Computations.* Technical Report 87-49, ICASE, July 1987.

[18] D.M. Nicol and J.H. Saltz. *Dynamic Remapping of Parallel Computations with Varying Resource Demands.* Technical Report 86-45, ICASE, July 1986. To appear in *IEEE Transactions on Computers.*

[19] D.M. Nicol and J.H. Saltz. *Optimal Pre-scheduling of Problem Remappings.* Technical Report 87-52, ICASE, September 1987.

[20] R. S. Pindyck and D. L. Rubinfeld. *Econometric Models and Economic Forecasts.* McGraw-Hill, New York, 1976.

[21] J. Rattner. Concurrent processing: a new direction in scientific computing. In *AFIPS Conference Proceedings, National Computer Conference*, pages 157–166, 1985.

[22] H.S. Ross. *Stochastic Processes.* Wiley, New York, 1983.

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

[25] J.H. Saltz and D.M. Nicol. Statistical methodologies for the control of dynamic remapping. In *Proceedings of the ARO Conference on Medium Scale Multiprocessing*, Stanford University, 1986.

[26] Joel Saltz and M.C. Chen. Automated problem mapping: the crystal runtime system. In *The Proceedings of the Hypercube Microprocessors Conf., Knoxville, TN*, September 1986.

[27] Y. Won and S. Sahni. Maze routing on a hypercube multiprocessor computer. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 630–637, St. Charles, Illinois, August 1987.

NASA National Aeronautics and Space Administration

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA CR-178379 ICASE Report No. 87-39		

4. Title and Subtitle	5. Report Date
PRINCIPLES FOR PROBLEM AGGREGATION AND ASSIGNMENT IN MEDIUM SCALE MULTIPROCESSORS	September 1987
	6. Performing Organization Code
7. Author(s) David M. Nicol and Joel H. Saltz	8. Performing Organization Report No. 87-39
	10. Work Unit No. 505-90-21-01
9. Performing Organization Name and Address Institute for Computer Applications in Science and Engineering Mail Stop 132C, NASA Langley Research Center Hampton, VA 23665-5225	11. Contract or Grant No. NAS1-18107
	13. Type of Report and Period Covered Contractor Report
12. Sponsoring Agency Name and Address 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 IEEE Trans. Comput.

16. Abstract One of the most important issues in parallel processing is the mapping of workload to processors. This paper considers a large class of problems having a high degree of potential fine grained parallelism, and execution requirements that are either not predictable, or are too costly to predict. The main issues in mapping such problems onto medium scale multiprocessors are those of aggregation and assignment. We study a method of parameterized aggregation that makes few assumptions about the workload. The mapping of aggregate units of work onto processors is uniform, and exploits locality of workload intensity to balance the unknown workload. In general, a finer aggregate granularity leads to a better balance at the price of increased communication/synchronization costs; the aggregation parameters can be adjusted to find a reasonable granularity. The effectiveness of this scheme is demonstrated on three model problems: an adaptive one-dimensional fluid dynamics problem using message passing, a sparse triangular linear system solver on both a shared memory and a message-passing machine, and a two-dimensional time-driven battlefield simulation employing message passing. Using the model problems we study the trade-offs between balanced workload and the communication/synchronization costs. Finally, we use an analytic model to explain why the method balances workload and minimizes the variance in system behavior.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
parallel processing, problem mapping, problem aggregation, multiprocessors	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	39	A03