Purdue University
Purdue e-Pubs

Department of Computer Science Technical Reports

Department of Computer Science

1986

Parallelism in Solving PDEs

John R. Rice
Purdue University, jrr@cs.purdue.edu

Report Number:
86-604

Rice, John R., "Parallelism in Solving PDEs" (1986). *Department of Computer Science Technical Reports.* Paper 523.
https://docs.lib.purdue.edu/cstech/523

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.

PARALLELISM IN SOLVING PDES

John R. Rice

Computer Sciences Department
Purdue University
West Lafayette, IN 47907

CSD-TR-604
July 1986
(Revised July 1988)

PARALLELISM IN SOLVING PDES

John R. Rice*
Department of Computer Science
Purdue University

CSD-TR 604

July 1, 1986

ABSTRACT

This paper examines the potential of parallel computation methods for partial differential equations (PDEs). We first observe that linear algebra does not give the best data structures for exploiting parallelism in solving PDEs, the data structures should be based on the physical geometry. There is a naturally high level of parallelism in the physical world to be exploited and we show there is a natural level of granularity or degree of parallelism which depends on the accuracy needed and the complexity of the PDE problem. We discuss the inherent complexity of parallel methods and parallel machines and conclude that dramatically increased software support is needed for the general scientific and engineering community to exploit the power of highly parallel machines.

* This work supported in part by Air Force Office of Scientific Research grant AFOSR-84-0385.

I. INTRODUCTION AND SUMMARY

This paper examines the potential for the use of parallelism in the solution of partial differential equations (PDEs). There are six principal points made as follows:

1. Linear algebra is not the right model for developing methods for PDEs and it is particularly inappropriate for parallel methods.
2. The best data structures for PDE methods are based on the physical geometry of the problem.
3. Physical phenomena have large components that inherently parallel, local and asynchronous. Parallel methods can be found to reflect and exploit this fact.
4. There is a natural granualarity associated with parallel methods for PDEs. The best number of "pieces" and processors depends on the complexity of the physical problem, the accuracy desired and properties of the iteration used.
5. Parallel machines are very messy and it is essential for most users that one have very high level PDE systems to hide this mess.
6. There is much to be gained to using regularity in parallel methods, but one should not carry this to extremes.

II. LINEAR ALGEBRA DOES NOT GIVE THE BEST DATA STRUCTURES

In recent years there have been numerous papers written about linear algebra on parallel/vector machines (see [Hwang, 1984], [Sameh, 1983] and [Ortega and Voight, 1985] for surveys and further references). Many machines have been designed to provide very high performance for linear algebra computations (see [Hwang and Biggs, 1984] and [Hwang, 1984] for surveys and further references). Most of this work is motivated or justified in some part by applications to solving PDEs. Solving large linear problems is an inherent step in solving PDEs and it is

usually the most expensive step, yet the thesis of this section is that most linear algebra approaches can be misleading for exploiting parallelism in solving PDEs.

A case in point is *nested dissection*. This was a breakthrough in solving PDEs, one that many people (including myself) had searched for over a period of decades. The original presentation [George, 196x] of nested dissection was inscrutable. If one starts (as everyone did) with the linear algebra problem $Ax = b$, then to discover nested dissection, one had to see that the matrix rearrangement such as shown in Figure 1 was the "right" way to eliminate the unknowns. However, if one expresses the reordering in terms of the underlying geometry of the PDE, one sees that nested discretion is a natural divide and conquer algorithm. It is then easy to understand why the method works so well, to see how to extend it to nonrectangular domains or to 3 dimensions or to finite element methods.

If one starts with a conventional matrix/vector representation of a PDE computation it is much harder to find efficient methods because the inherent structure of the PDE problem is so distorted by conventional matrix/vector representations. This is further illustrated in Figure 3 which shows the conventional matrix structure obtained by discretizing a second order PDE with derivative boundary conditions on the domain shown there. It is a computational tour-de-force to recover from Figure 3 the information that is superficially apparent in the domain picture.

The shortcomming of the conventional linear algebra approach is that the right data structure is not used, instead one should base the data structure on the underlying physical geometry. Figure 4 shows a domain which has been "exploded" to group "like-kinds" of elements together in a PDE problem. A method that is really successful in exploiting parallelism in this problem must "know" this structure, the most practical way to know it is to have it given explicitly in the data structure. More complex problems have other structure (interfaces, singular points, etc.) that can be incorporated in a similar way. It is not just parallelism in the computation that needs information such as seen in Figure 4, the control of numerical methods also need

it. Numerical models need to be more accurate (e.g., grids need refining) near special locations. The partitioning of the computations for rapid convergence in iterative methods is strongly influenced by this information.

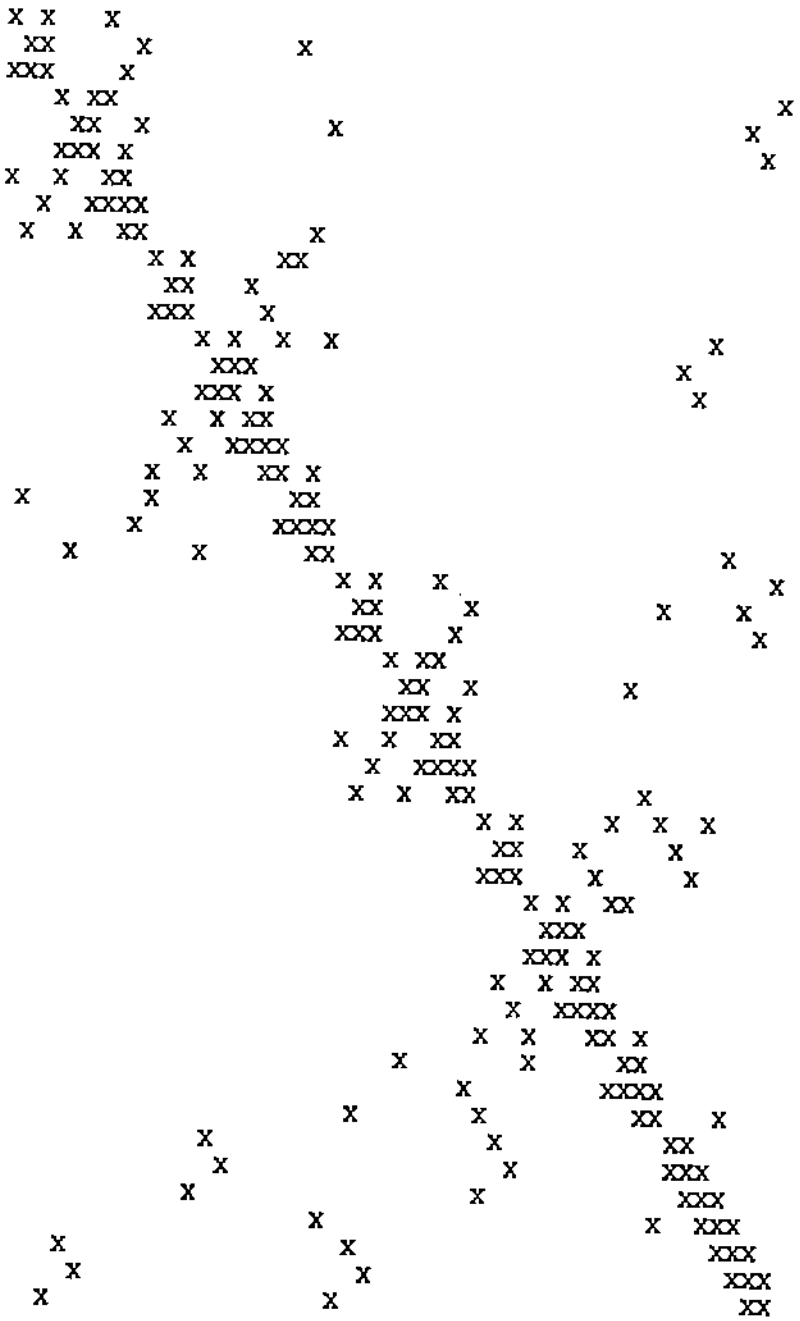

Figure 1. The pattern of non-zeros that occurs in solving Laplace's equation using the nested dissection ordering of the conventional matrix formulation using finite differences. The order of the unknowns eliminated is given below the pattern.

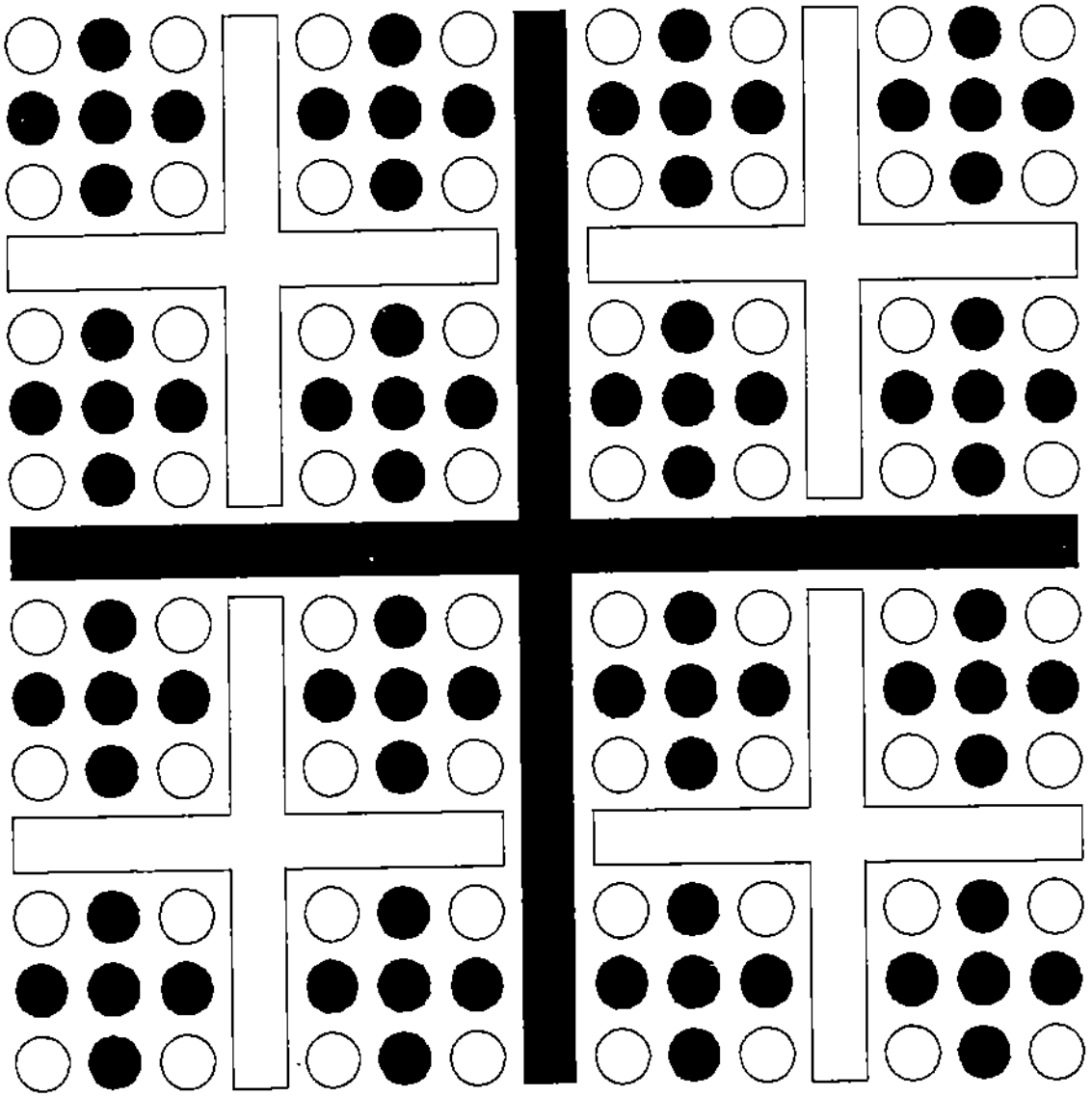

Figure 2. A visualization of the nested dissection ordering shown on a two dimension grid. The height of the surface indicates the level at which an unknown lies. Unknowns on the same level which are separated by a higher level may be eliminated independently of one another.

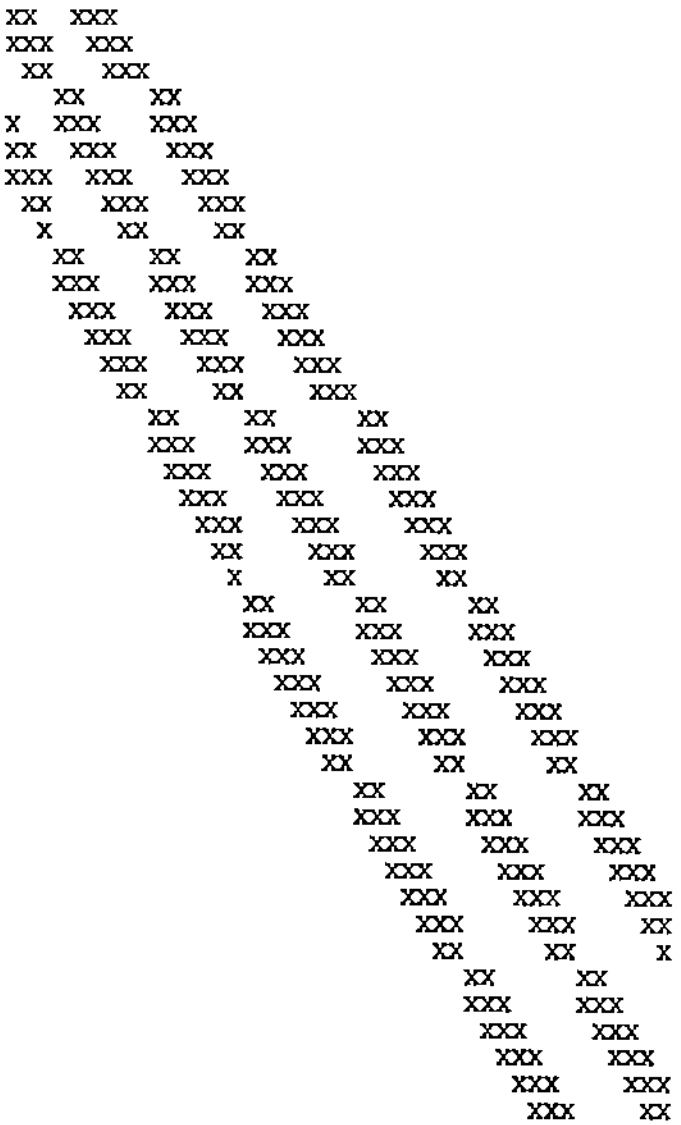

Figure 3. The conventional matrix structure (A) obtained from a 9-point finite difference discretization on the domain (B).

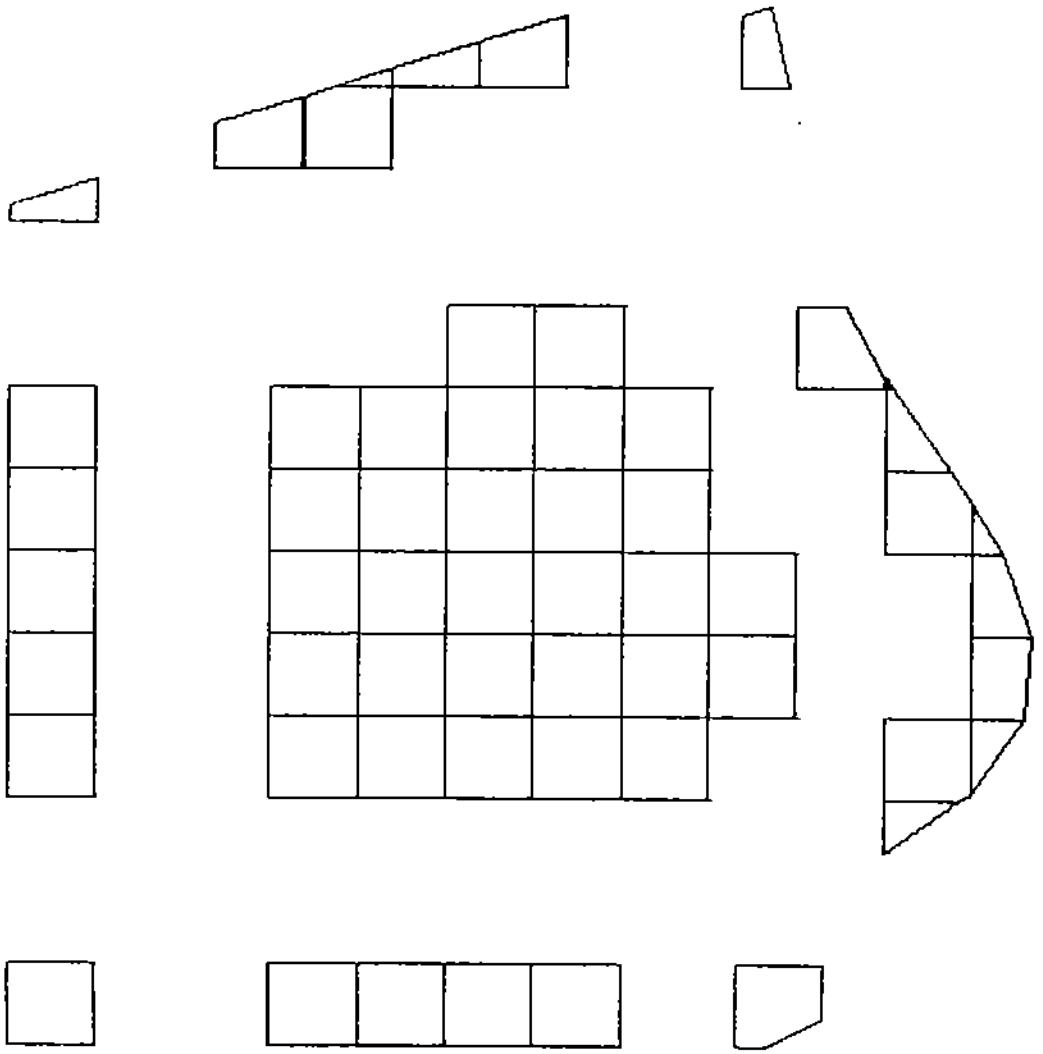

Figure 4. An exploded view of a physical domain which shows the elements of a "like" nature grouped together. The groupings are the first step in determining an appropriate structure in the problem of an efficient parallel method.

III. PARALLELISM IS (ALMOST) UNLIMITED IN SOLVING PDEs

We claim that the physical phenomena that PDEs model are inherently local in space and asynchronous. Locality means that the phenomena are inherently amenable to parallel methods, the computation done at point A does not depend on anything being done at the physically distant point B. There are logical limits to the potential parallelism, we do not foresee much parallelism in time (as opposed to space) except for very special situations. There is also some sequentiality in local computations, one must compute values of coefficient functions in an equation before one can use the equation. For specific applications one can often reduce the sequential work dramatically by preprocessing computations (i.e., computing everything possible as soon as possible).

The preceding observations are based on asymptotic considerations, i.e., if the physical domain is big enough and the accuracy required is high enough then any fixed number N of processors can be used profitably. We argue, however, that there is natural optimal or appropriate granularity and number N of processors associated with any particular PDE computations. We measure granularity in terms the number N of *elements* of the computation or model of the physical object. For simplicity we ignore any cases where computational elements do not correspond naturally with physical elements. The two extremes are:

(i) $N = 1$ processor gives 1 element which gives a sequential computation which gives very limited speed.

(ii) N very large (one Cray 2 per atom in a river?) gives a huge number of elements which gives very high parallelism which gives almost unlimited speed.

There are four considerations (at least) besides cost which lead to the existence of an optimal granularity, they are

1. Every problem has an *acceptable solve time* beyond which solving it faster does not matter.

2. Every problem has an *acceptable accuracy* beyond which more accuracy does not matter.

3. For a fixed physical problem, the number of interfaces between elements grows with the number of elements, thereby increasing the complexity and communication requirements of the computation. This growth might be very slow.

4. For a fixed problem and method, the total work might eventually grow faster with N than parallelism reduces it because of slower convergence of iterative methods, etc.

Having identified granularity with N, we see that the independent variables in an application design are N, the desired elapsed time T and the required accuracy ε. Assume now that ε behaves in a known way, that it is fixed and we only consider choosing N to achieve a specified T value. Figure 5 shows an idealized plot of cost versus time to solve a particular problem using a fixed number N of processors. The key points are that there is a lower limit on time (because processors can go only so fast) and that cost quickly reaches a plateau as the time increases. Figure 6 shows a different view of the situation, cost versus N for a fixed time to solve a particular problem. Again there is a lower limit because processors can go only so fast, but there is also an optimum. As N increases the cost starts to increase because of idle processors and/or increased communication (overhead) costs.

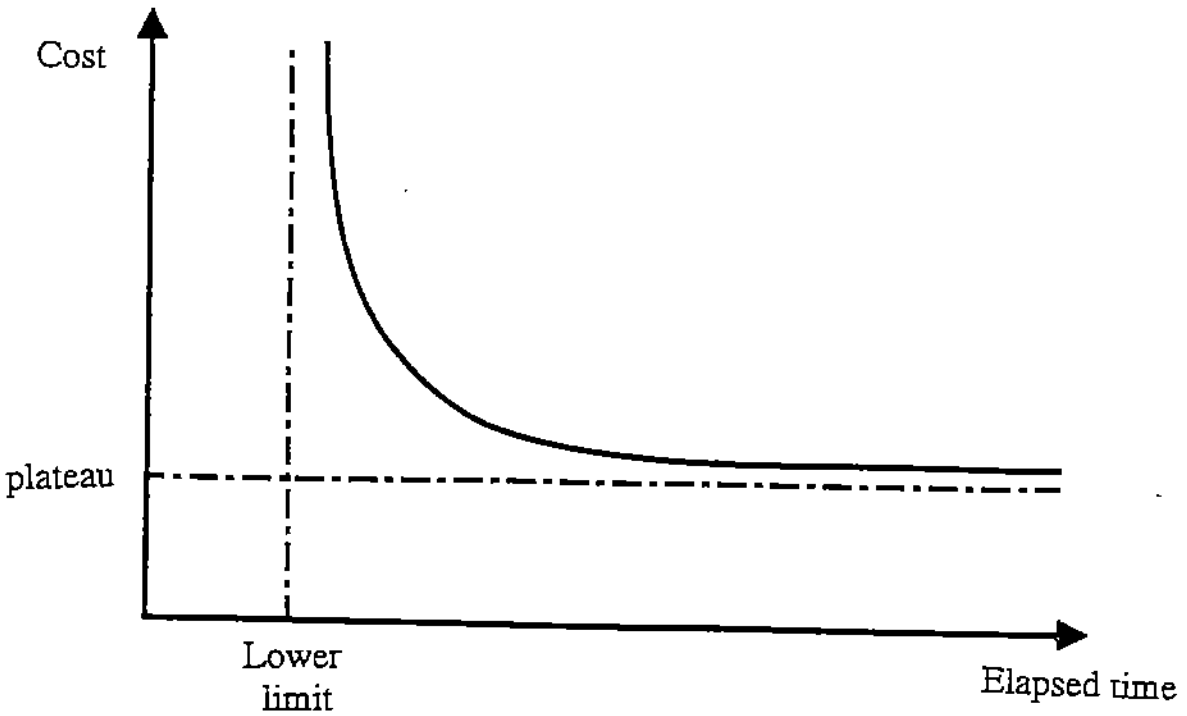

Figure 5. Cost versus elapsed time to solve a particular problem using N processors.

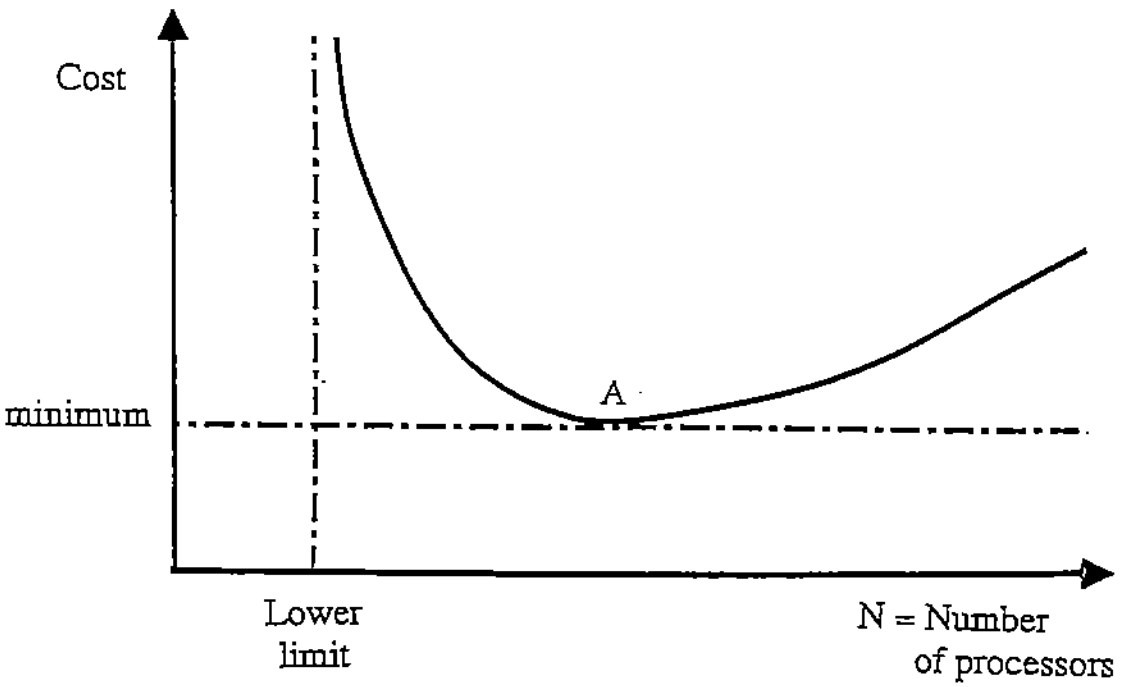

Figure 6. Cost versus the number of processors N used to solve a particular problem in a fixed elapsed time. The point A gives the minimum cost using an optimal number of processors.

We can replot the information of Figures 5 and 6 in the (N, T) plane and show two curves: the limiting curve of what is possible and the curve of optimal combinations of T and N. This is shown in Figure 7, the shapes are purely conjectural, one does not know what they are. It is true that cost decreases monotonically from point C to D.

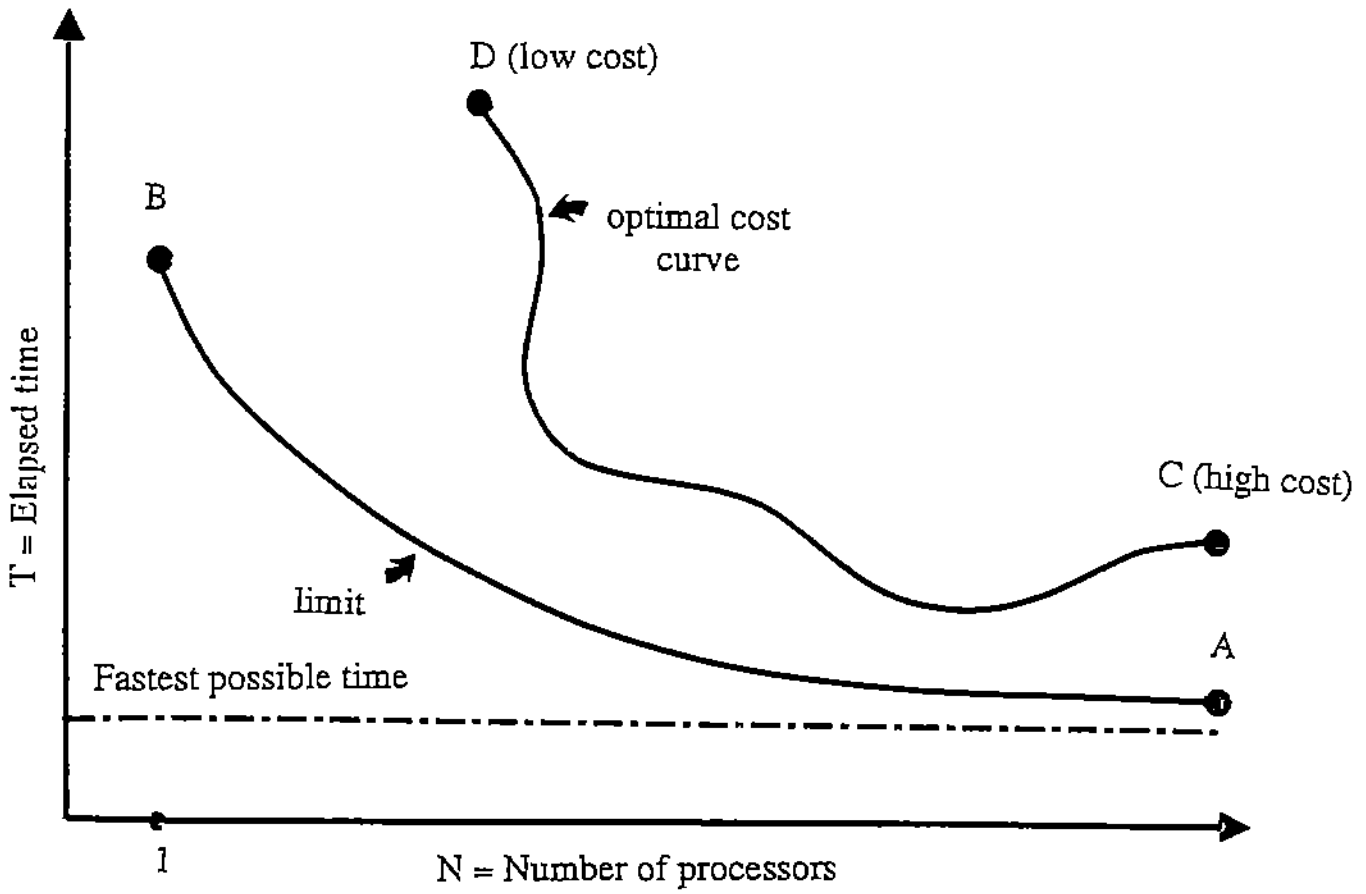

Figure 7. The (N, T) plane showing the limiting curve (A to B) of what is possible and the locus (C to D) of optimal cost combinations of N and T.

Thus we see that while in principle there might be no limit on the amount of parallelism that can be used in solving PDEs, there is definitely such a limit for any fixed application. Very little is known about actual values for real problems. I believe we are very far from the methods that give optimal time or cost in solving PDEs. On the other hand, I find it very convincing to argue that many real problems are very complex and that to achieve "engineering" accuracy and "reasonable" elapsed time with even a low cost method (never mind optimal cost) will use

thousands of processors.

IV. PARALLEL METHODS REQUIRE NEW SOFTWARE SYSTEMS

Parallel machines are already rather complex, much more so than previous computers. They will become even more complex as it is discovered that a mixed set of capabilities provides more efficient computing. There will be variety is everything: processors (integer, floating point, graphics, vector, FFT, ...), memory (local, global, cache, archival, read only, ...), I/O (keyed, text, graphics, movies, acoustical, analog, ...), communication (message passing, packets, buses, synchronous/asynchronous, hypercubes, high/low speed, long haul, ...). The difficulty in managing (programming) this complexity is easily an order of magnitude higher than for present machines. The difficulty is compounded by the fact that changes in the capabilities available will become much more frequent.

The current programming methodology for solving PDEs is that of Fortran. One has a fairly intelligible language where one can exert fairly direct control of the machines resources. Each Fortran statement is typically implemental by 5-10 machines instructions. There must, I believe, always be such a language and I believe that Fortran will be expanded to handle the greater complexity of the machines. It might also be replaced by another moderate level language with such capabilities, e.g. Ada or C suitably enhanced. However, it will no longer be reasonable to expect the end-user scientists and engineers, the people who solve PDEs, to learn how to manage this complex computational environment. They will generally not do a very good job of it and, even if they did a good job, it would be a great waste of talent and duplication of effort. The potential benefits of parallel computation will not be achieved if every user has to master (even partially) how to manage such complex machines.

The solution to this problem is to substantially raise the level of the user's "programming" language. He must be able to say in a natural and succinct way what is to be done. In the PDE

context they should be able to say things like:

1. Solve $(1 + x^2)u_{xx} + u_{yy} - \sin(\alpha y)u = \text{Force } 2(x, y)$

   on the Domain #12

   with $u = 1$ on the boundary.

2. Use finite differences with a 40 by 40 grid

   plus SOR iteration

3. Show me plots of u , u_x and u_y

In fact, we must aim eventually for the situation where statement 2. is replaced by

2a. Obtain an accuracy of about 0.5 percent

Then, between such a program and the Fortran level is a layer of software which has two components. The first is a set of *problem solving modules* written by people who are relatively expert in solving the problems at hand and experienced in how parallelism (or other special capabilities available) can be exploited. There will be different methods (or, at least, different implementations) in the modules suitable for important subclasses of machines.

The second component of this layer is a set of *computation management facilities* written by people who are relatively expert in memory management, network scheduling, program transformations, etc. They have spent the time to learn how to provide such facilities well and have embedded much of their expertise into their software. These two components are then integrated to provide a bridge between the high level user input and a Fortran-like program targeted for the particular machine (or machines) to be used to solve the problem.

The obvious advantage of this methodology is that, if it works, there is a dramatic reduction in programming effort. This is, of course, the goal of introducing the methodology. Note that this not being done just to reduce software costs, the "mass-market" viability of parallel computation depends on introducing a methodology which hides the underlying complexity from most

users.

The obvious disadvantage of this methodology is that the intermediate layer might introduce so much in efficiency that the power of parallelism is seriously weakened or even lost. It is clear that no foreseeable software for managing a computation can be as clever, resourceful and effective as clever, experienced people. This fact is a smokescreen that obscures a much more relevant "fact": people, even clever and experienced ones, almost never get close to "optimal" computations because they do not take the time to do it, it is inordinately expensive to do so. The result is that a good software system, one with many flaws which does many obviously stupid things, consistently can produce moderately good implementations which are significantly better than the ones people consistently produce. Scientific evidence to support this fact is scarce, but there is one solid data point.

Figure 8 shows a program written in DEQSOL, a high level PDE problem solving language under development at Hitachi [Umetani, 198x]. No attempt is made here to explain DEQSOL. Hitachi has two PDE application programs that were written in FORTRAN prior to their vector supercomputer and DEQSOL efforts. These programs were brought into their vectorizing Fortran compiler environment and hand tuned to run well on their machines. The problems being solved were later reprogrammed in DEQSOL which produces a Fortran program which then use the vectorizing Fortran compiler but no hand tuning. The results of this experiment are shown below.

	A	B
FORTRAN:		
lines of code	1361	1567
execution time (sec.)	2.3	5.8
DEQSOL:		
lines of code	127	132
execution time	0.6	1.8
speed up factor	3.8	3.2

We see that not only was the programming effort reduced by the least an order of magnitude, but there was also a very worthwhile *gain* in execution speed. Keep in mind that a speedup of 3 or 4

is the typical total benefit achieved from using vector hardware on Cray and Cyber 205 machines.

We illustrate the power that can be achieved using such high level languages by considering the Plateau problem:

$$(1 + u_x{}^2)u_{xx} - 2u_x u_y u_{xy} + (1 + u_y{}^2)u_{yy} = 0$$
$$u(x, y) \text{ given on the boundary of a region } R \qquad (1)$$

This is classical difficult PDE problem, its solution is the surface that a soap film takes on for a wire frame bent according to the value specified on the boundary of R. We solve this problem for the domain R and wire frame shape seen in the later figures (and explicitly defined in Figure 9). The high level language used is that of ELLPACK [Rice and Boisvert, 1985], one that provides modules and facilities for solving linear PDEs.

Figure 9 shows an ELLPACK program to implement Newton's method for (1). We do not explain the ELLPACK language here. A simple initial guess is made and the convergence is quite rapid in spite of the fact that the solution has a singularity (the wire has a sharp bend) along one side. The maximum differences between iterates are: 1.24, .30, 9.6×10^{-3}, 2.4×10^{-5} and 5×10^{-7}. The round off level (on a VAX 11/780) is reached at five iterations.

```
dom      x = [0:1] ,          /* 3D DIFFUSION PROBLEM */
          y = [0:1] ,
          z = [0:2] ;
tdom     t = [0:5] ;
mesh     x = [0:1:0.1] ,
          y = [0:1:0.1] ,
          z = [0:2:0.1] ,
          t = [0:5:0.001] ;
var      T ;              /* Temperature */
const
   rho = 1 ,          /* Density */
   c = 1 ,            /* Constant */
   k = 1 ,            /* Diffusion Constant */
   u = 0 ,            /* x-axis Velocity */
   v = 0 ,            /* y-axis Velocity */
   w = 5*(1.0-x**2)*(1.0-y**2) ,      /* z-axis Velocity */
   S = exp(-x**2-x**2-(1.0-z)**2) ;
                          /* Source Distribution */

cvect    V = (u, v, w) ;       /* Velocity Vector */
region
   In = (*, *, 0) ,          /* In */
   0 = (*, *, 2) ,           /* Out */
   X0 = (0, *, *) ,          /* Left */
   X1 = (1, *, *) ,          /* Right */
   Y0 = (*, 0, *) ,          /* Bottom */
   Y1 = (*, 1, *) ,          /* Top */
   R = ([0:1], [0:1], [0:2]) ;  /* Whole Region */

equ      rho*c*(dt(T)+V..grad(T)) = k*lapl(T)+S ;

bound    T = 0      at   In+X1+Y1 ,
          dz(T) = 0   at  0  ,
          dx(T) = 0   at  X0 ,
          dy(T) = 0   at  Y0 ;
init     T = 0 at R ;

ctr      NT ;    /* Iteration Counter */

scheme ;
  iter NT until NT gt 200;
    T<+1> = T+dlt*((k*lapl(T)+S)/(rho*c)-V..grad(T)) ;
    print T at Y0 ;
    disp T at Y0 every 100 times ;
  end iter ;
end scheme ;
end ;
```

Figure 8. A DEQSOL program for applying Newton's method to solve the Plateau problem.

Our final point in the software and programming area concerns the role *regularity* in data structures, in algorithms and in programs. Clever programmers and hardware designers can do a lot of special things to exploit special situations. This exploitation is usually achieved at the cost of more complex software and hardware. Thus there must be a balance between the execution time costs and the design costs of software and hardware. While it is hard to defend general statements on the matter, we believe that the optimum lies nearer to regularity and its attendant simplicity than it does to irregularity and its attendant complexity. However, we feel *extreme* simplicity is not the best approach either.

This view is illustrated by an example in discretizing a domain. Figure 12 shows a physical domain that has been partitioned in six ways for a problem with difficulties near the right boundary:

(A) A fine triangulation of a common type
(B) A fine, uniform, rectangular overlay grid
(C) Mapping the domain to a rectangle and inducing a logically rectangular partition
(D) Triangulation adapted to the difficulty
(E) Rectangular overlay grid adapted to the difficulty
(F) Logically rectangular partion adapted to the difficulty

We believe that the irregular triangulations do not provide any execution time advantage over the more regular partitions (one can do a regular triangulation if one wants). On the other hand, we also believe that the uniformly spaced partitions are too simple and have too large an execution time penalty. We believe the adaption will pay off. The logically rectangular discretization is the simplest to program but the relative execution efficiencies resulting from (E) and (F) are not clear. Thus we believe that the search for the "best" method should be concentrated on partitions like (E) and (F) but there are still many undetermined degrees of freedom.

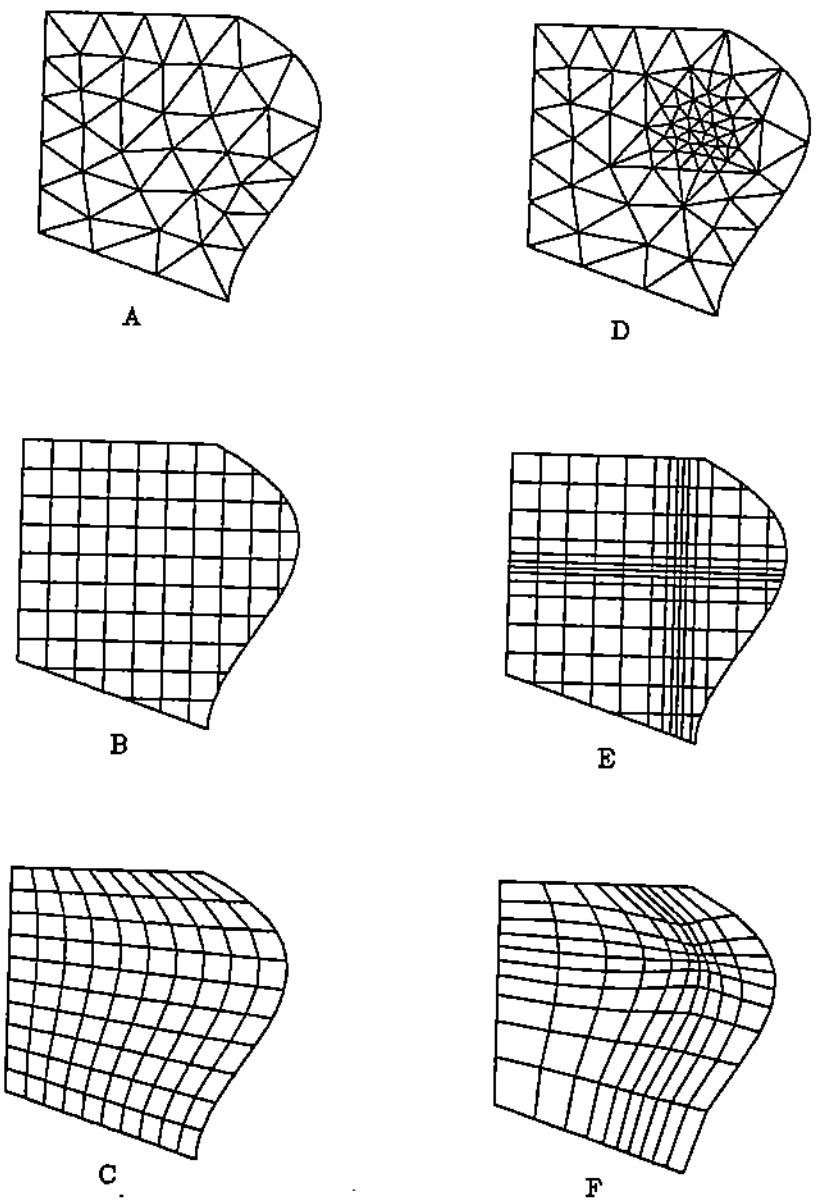

Figure 10. Six ways to partition a domain showing ways to achieve regularity and to adapt to a difficulty. The letters A through F refer to the discussion in the text.

V. REFERENCES

K. Hwang and F. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, New York, 1984.

K. Hwang, *Supercomputers: Design and Applications*, IEEE EH0219-6, Silver Spring, 1984.

J. Ortega and R. Voight, Solution of Partial differential equations on vector and parallel computers, SIAM Review, 27 (1985), 149–240.

J. Rice and R. Boisvert, *Solving Elliptic Problems Using ELLPACK*, Springer-Verlog, New York, 1985.

A. Sameh, An overview of parallel algorithms for numerical linear algebra, *First Int. Colloquium on Vector and Parallel Computing in Scientific Applications*, Paris, 1983.