Purdue University

Purdue e-Pubs

Department of Computer Science Technical Reports

Department of Computer Science

1986

Parallel Methods for PDES

John R. Rice

Purdue University, jrr@cs.purdue.edu

Report Number:

86-587

Rice, John R., "Parallel Methods for PDES" (1986). *Department of Computer Science Technical Reports.* Paper 506.

https://docs.lib.purdue.edu/cstech/506

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.

PARALLEL METHODS FOR PDES

John R. Rice

CSD-TR 587
April 1986

PARALLEL METHODS FOR PDES*

John R. Rice**
Department of Computer Science
Purdue University

CSD-TR 587

April 1, 1986

ABSTRACT

This paper examines the potential of parallel computation methods for partial differential equations (PDEs). We start by observing that linear algebra is not the right model for PDE methods, that data structures should be based on the physical geometry. We observe that there is a naturally high level of parallelism in the physical world to be exploited. An analysis is made showing there is a natural level of granularity or degree of parallelism which depends on the accuracy needed and the complexity of the PDE problem. It is noted that the granularity leads to the use of superelements and that computational efficiency suggests that these should be of higher accuracy. 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. The paper ends with a brief taxonomy of methods for PDEs, the classification is based on the method's use of three basic procedures: Partitioning, Discretization and Iteration.

* To appear as chapter in Taxonomy of Parallel Algorithms, Gannon and Jamieson, MIT press, 1987.
** This work supported in part by Air Force Office of Scientific Research grant AFOSR-84-0385

PARALLEL METHODS FOR PARTIAL DIFFERENTIAL EQUATIONS

John R. Rice
Department of Computer Science
Purdue University
West Lafayette, IN 47906

I. INTRODUCTION AND SUMMARY

This paper examines the potential for the use of parallelism in the solution of partial differential equations (PDEs). There are eight 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 granularity 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. Partitioning the computation into pieces is equivalent to using superelements in the discretization.
6. More accuracy in the superelements can enormously reduce the computational task.
7. Parallel machines are very messy and it is essential for most users that one have very high level PDE systems to hide this mess.
8. There is much to be gained to using regularity in parallel methods, but one should not carry this to extremes.

The final section presents a taxonomy of PDE methods. The classification is based on how

the methods use three procedures: Partitioning, Discretization and Iteration. It is conjectured that the most promising methods for parallelism are those created in the order: iterate, partition, then discretize.

II. PARALLEL METHODS FOR PDEs IS NOT ABOUT LINEAR ALGEBRA

In recent years there have been numerous papers written about linear algebra on parallel/vector machines (see [Hockney and Jesshope, 1981], [Dongarra and Sorensen, 1986], [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, 1984] and [Hwang and Briggs] for surveys and further references). Most of this work is motivated or justified in some part by applications to solving PDEs. Everyone sees that 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 are only tangentially relevant to solving PDEs and, in fact, they are often misleading.

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, 1973] 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 one is almost sure not to find efficient methods to solve the PDE. This is because the inherent structure

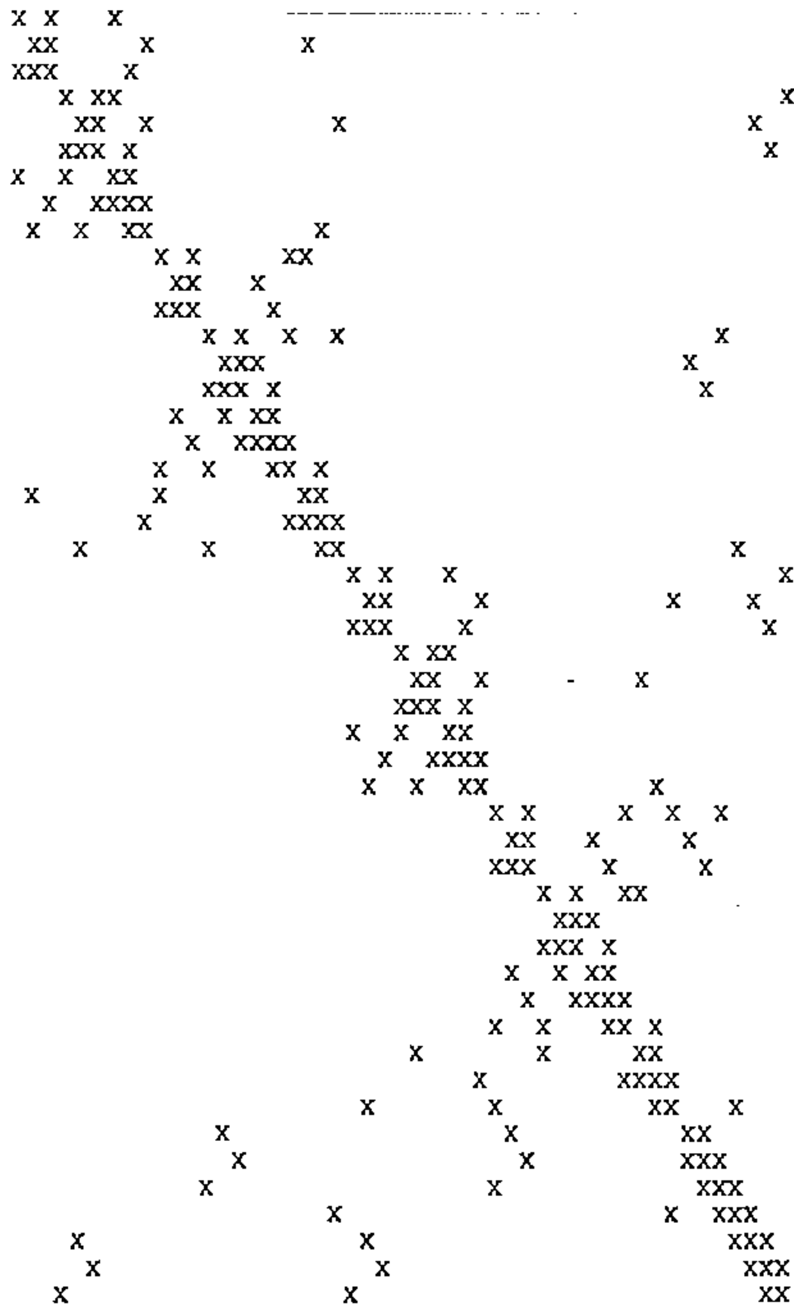


Figure 1. The pattern of non-zeros that occurs in solving Laplace's equation using the nested dissection ordering of the conventional matrix formulation with finite differences.

of the PDE problem is so distorted by conventional matrix/vector representations that it is infeasible to uncover the natural problem structure. This is further illustrated in Figure 3 which shows the conventional matrix structure obtained by discretizing a second order PDE using a 9 point star on the domain shown in Figure 2. It is a computational tour-de-force to recover from Figure 3 the information that is superficially apparent in Figure 2.

Figure 4 gives a geometric visualization of nested dissection. First the domain is divided in four parts (by the open bars), each of these parts is then, in turn, divided into four parts (by the solid bars). Then each of these is divided into four parts (by the open circles). This leaves the solid circles completely isolated from one another. The discretization equation (using a 5-point or 9-point star) at one of the solid circles does not involve any variable at the other points. Thus the solid circle unknowns can be eliminated independently (and simultaneously). After that the unknowns at the small connected groups of open circles can be eliminated independently. The order within the groups of five is immaterial. Then the solid bar unknowns can be eliminated followed by the open bar unknowns. This is the geometric pattern behind the matrix structure seen in Figure 1.

The shortcomming of the conventional linear algebra approach is that the right data structure is not used, one should base the data structure on the underlying physical geometry. Figure 2 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 could be incorporated in a similar way.

It is not just parallelism that needs information such as seen in Figure 2, 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

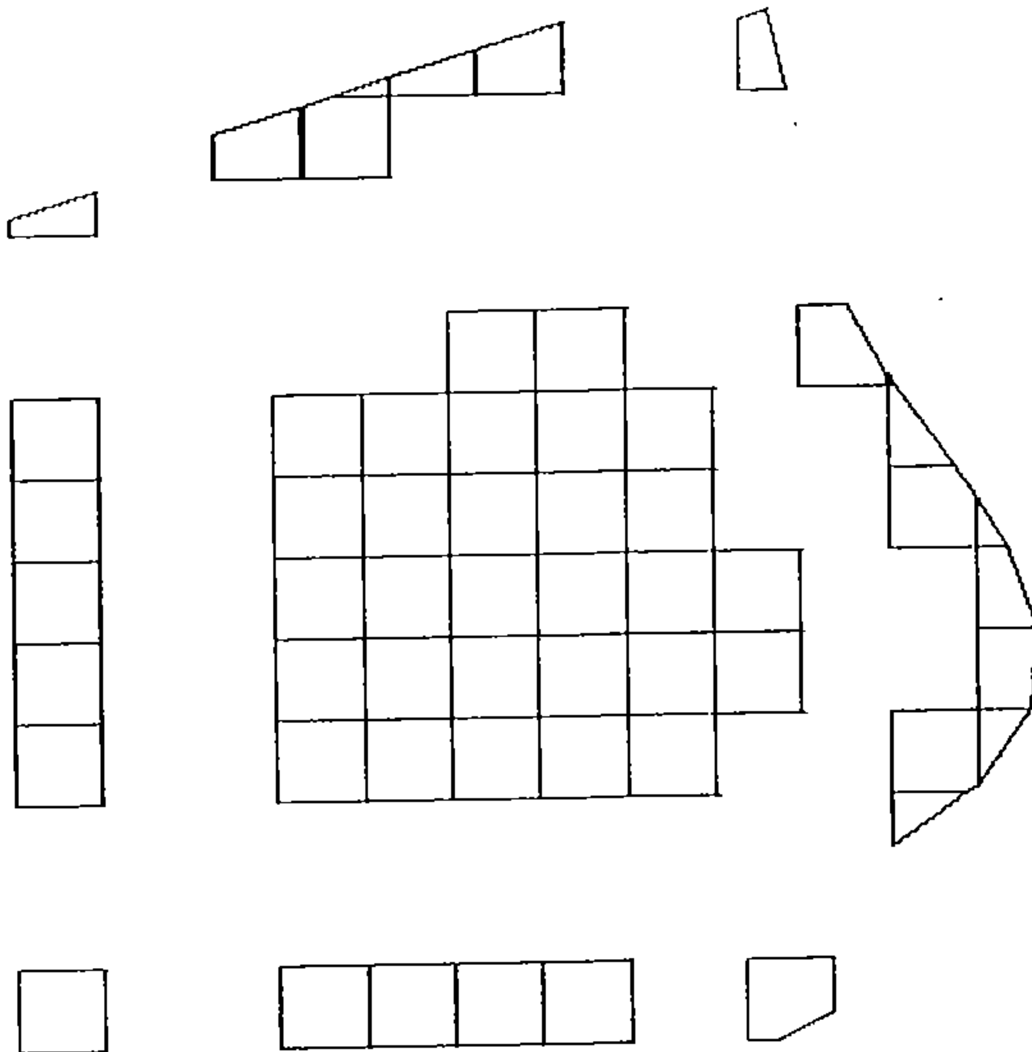


Figure 2. An exploded view of a physical domain which shows the elements of a ''like'' nature grouped together. The groupings are the first step in determining an appropriate structure in the problem of an efficient parallel method.

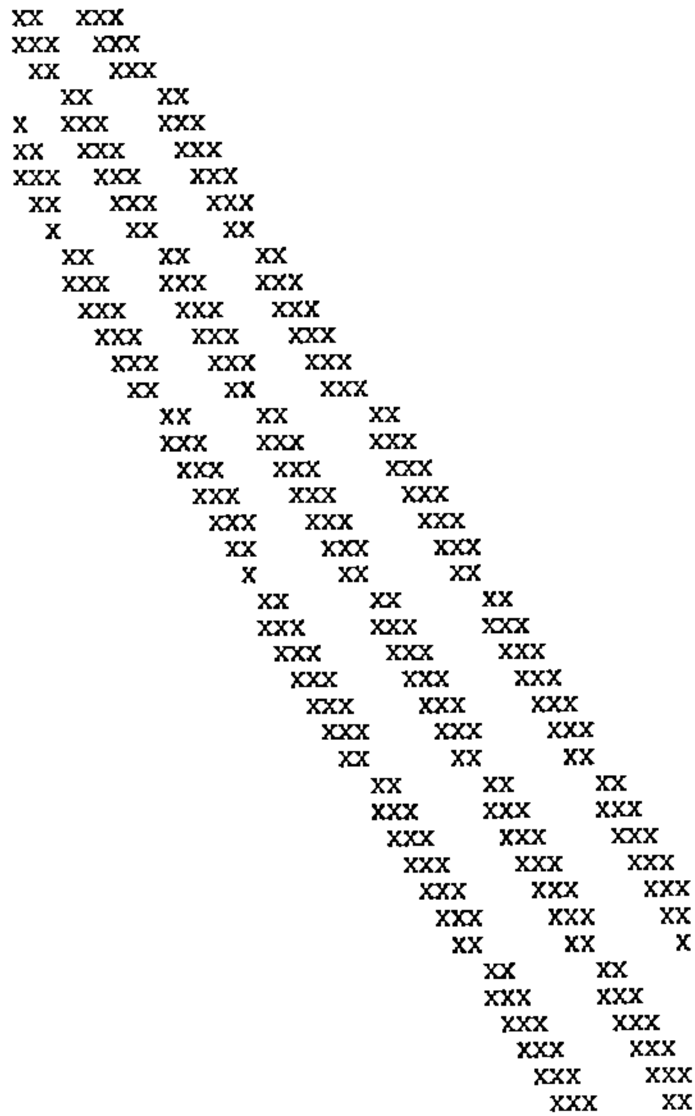


Figure 3. The conventional matrix structure obtained from a 9-point finite difference discretization on the domain seen in Figure 2.

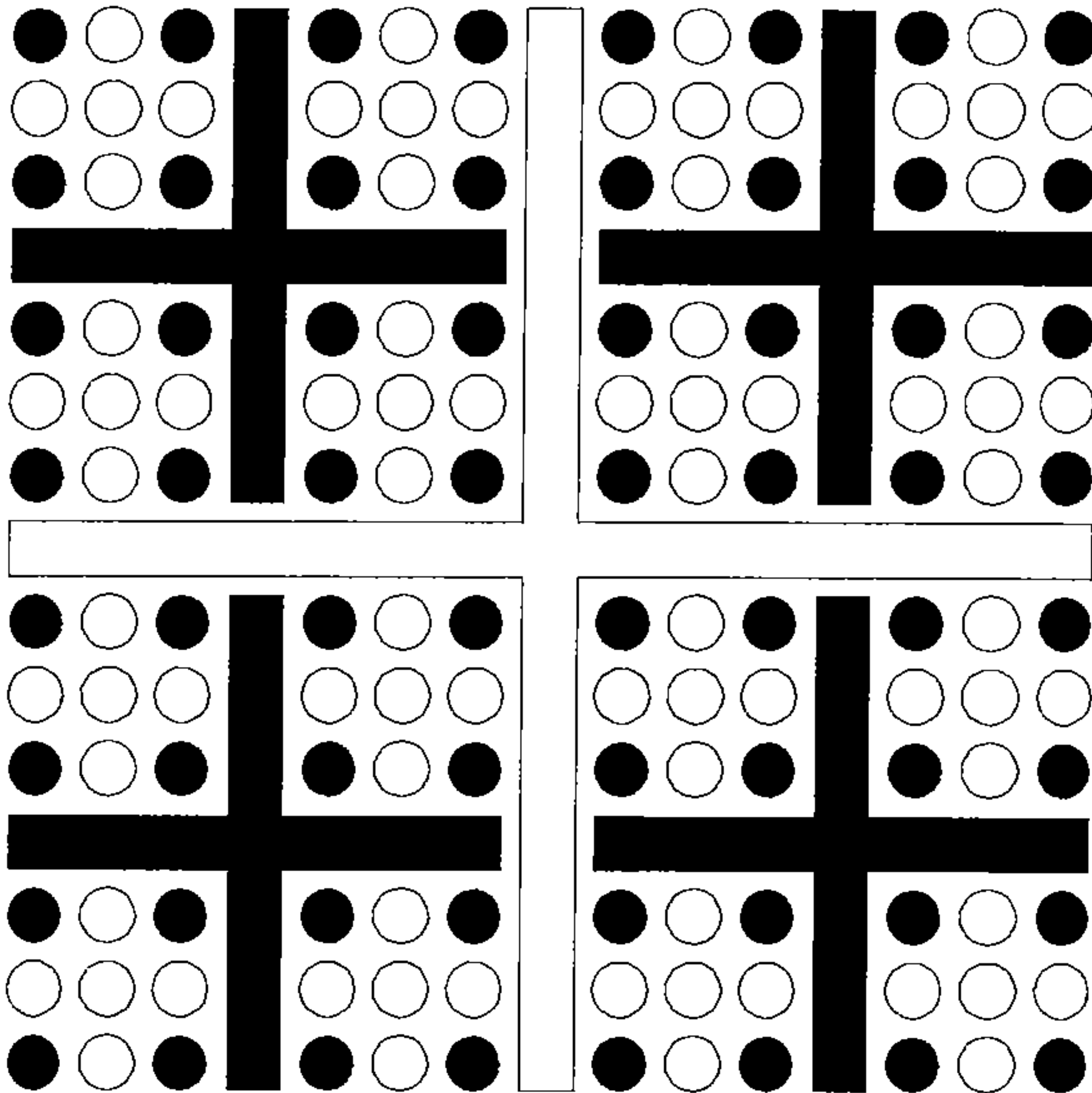


Figure 4. A visualization of the nested dissection ordering shown on a two dimension grid. The solid circles unknowns are eliminated first, followed by the open circles, then the solid bars and finally the open bars. The order of elimination within groups is immaterial.

iterative methods is strongly influenced by this information. To underscore this view of parallel methods for PDEs, we claim (conjecture?) that:

> *The really good parallel methods for PDEs do not require a global numbering of the unknowns in the computation.*

A global indexing is often useful to create a specific, efficient computational representation, but it should not be an essential ingredient in the method. We note that the work to obtain a global numbering can become the dominant component of the computation if it is done carelessly. A good parallel method for a PDE problem with K unknowns and N processors ($K >> N$) asymptotically should use time which is $O(K/N + \log N)$ or so, a numbering that requires $O(K)$ time must be avoided.

III. PARALLELISM IS (ALMOST) UNLIMITED IN SOLVING PDEs

We claim that the physical phenomena that PDEs model are inherently local and asynchronous. Locality means that they 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 size, 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. We assume for 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

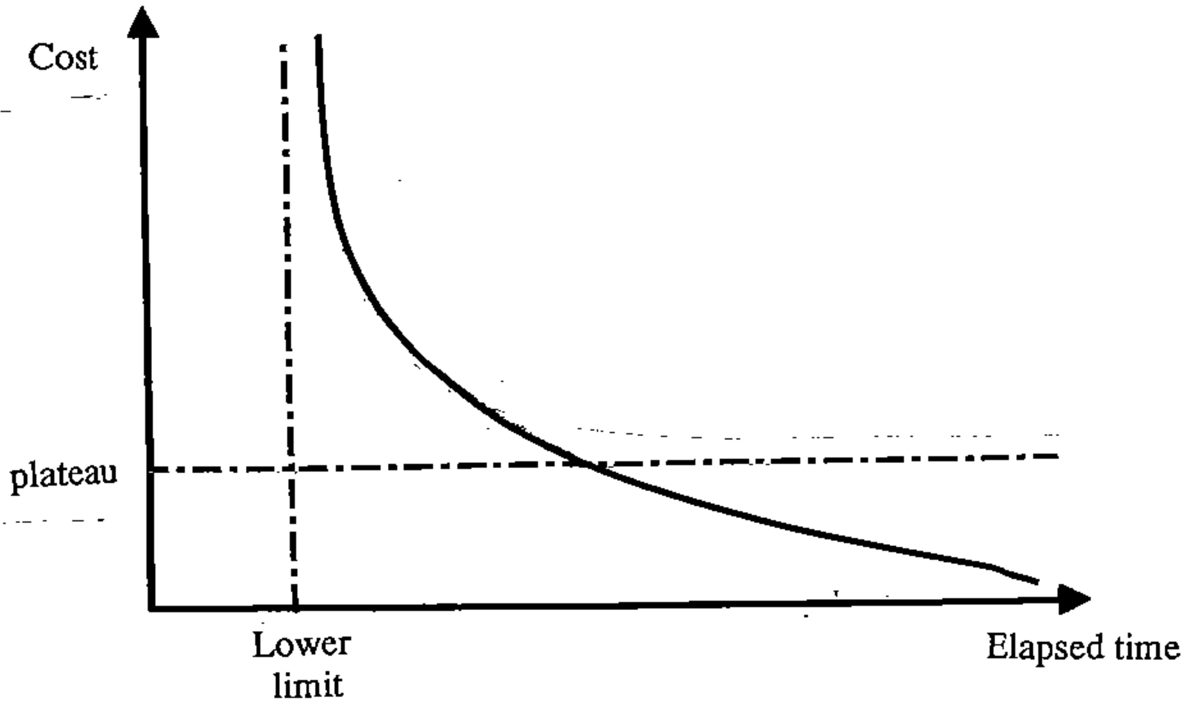


Figure 5. Cost versus elapsed time to solve a particular problem using N processors.

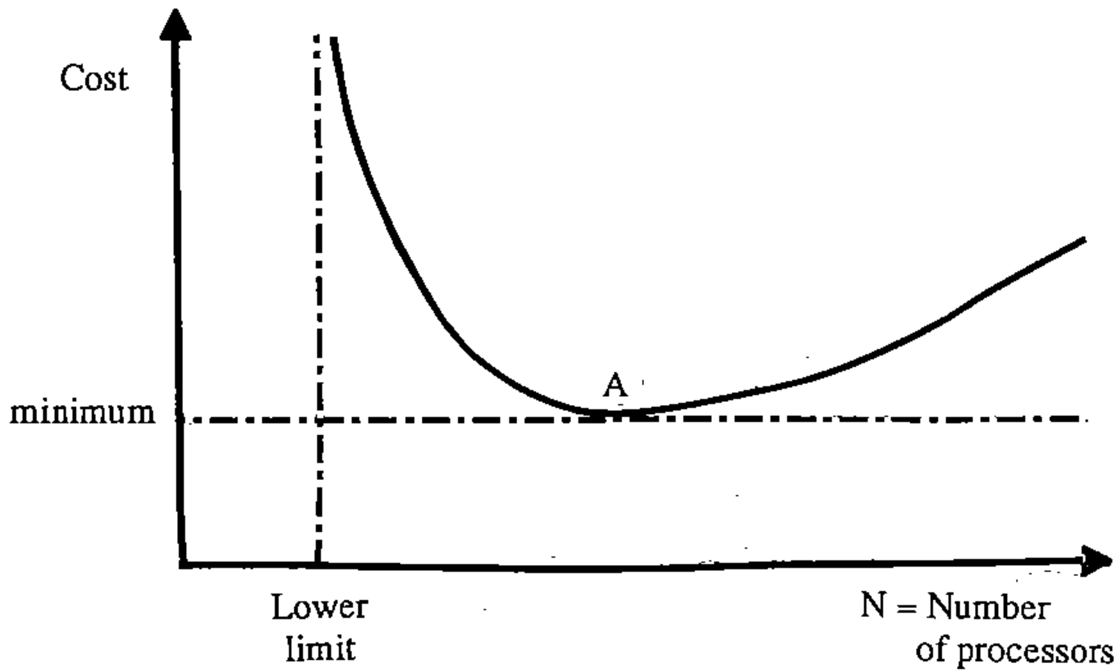


Figure 6. Cost versus the number of processors N used to solve a particular problem in a fixed elapsed time. The point A gives the minimum cost using an optimal number of processors.

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.

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.

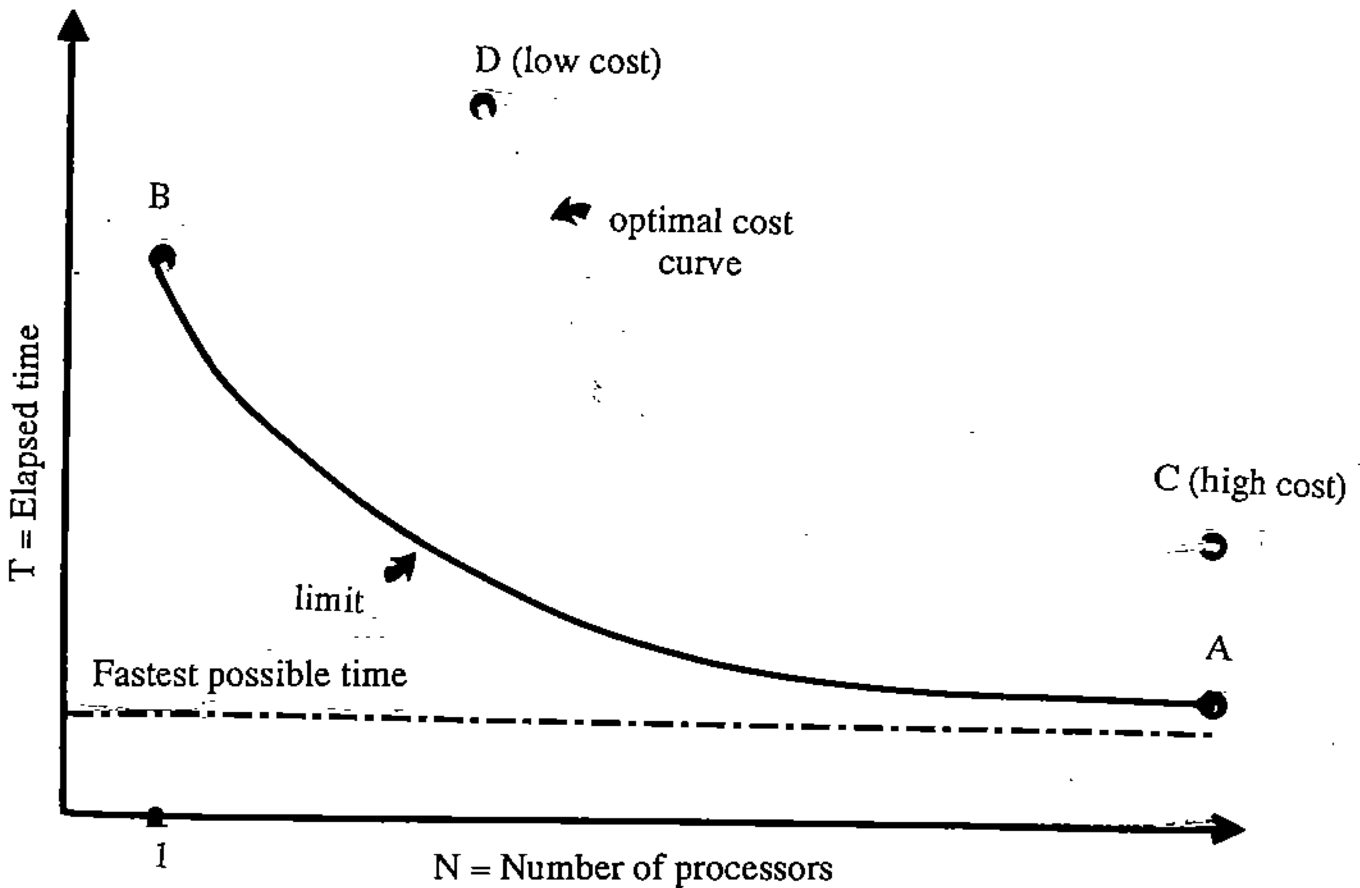


Figure 7. The (N, T) plane showing the limiting curve (A to B) of what is possible and the locus (C to D) of optimal cost combination of N and T.

Thus we see that while in principle there is 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 applications. 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 quite 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 CREATE SUPERELEMENTS

We have argued that parallel methods will tend toward partitioning the computations into elements or pieces of "moderate" size. There will usually correspond to dividing physical space into elements and, to give a concrete example, if the numerical model requires 10,000 grid point in a 100 by 100 space, you can expect N to be perhaps 25 to 500 so that each processor "handles" something like a 4 by 4 grid (16 points) to a 20 by 20 grid (400 points).

Given that this argument is correct, let us recall that we are striving for a computationally optimal method. So, within the smallish subgrid handled by each processor, we want the numerical model that achieves the accuracy requirements with minimal computations. We might rephrase this to say that each processor is to handle 16 to 400 degrees of freedom and this is to be optimal (or at least reasonably good) in efficiency. That is to say, each processor will handle a *superelement*, one with a fairly large number of degrees of freedom. We will, of course, never know the best numerical model. However, there is abundant evidence that the standard approach of using a large number of simple finite difference formulas or a large number of simple finite elements is *very far* from optimal. The really good parallel methods will use complex superelements with high accuracy.

The question of what kind of elements to use is not one intrinsic to parallelism because the same is true for sequential methods. Since the essence of the question is outside the topic of this paper, it is not pursued further here. See [Rice and Boisvert, 1985] for some of the evidence of the value of higher accuracy elements. We do believe that the inherent use of superelements in

parallel methods will make it more natural for one to introduce a little more complexity into the elements in order to gain considerably in accuracy and enormously in efficiency.

V. PARALLEL METHODS REQUIRE NEW SOFTWARE SYSTEMS

Parallel machines are already rather complex, much more so than previous operations of 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 should aim 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, 1984]. 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. There 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

```
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. The DEQSOL program to solve a time dependent, three dimensional diffusion problem.

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 speed up 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:

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

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.

Newton's method is a natural candidate to try to solve a nonlinear problem by iterating on a sequence of linear problems. In this case one differentiates (1) with respect to u and rewrites the standard iteration (F represents the PDE operator in (1))

$$u^{(N+1)} = u^{(N)} + (dF/du)^{-1}F(u^{(N)})$$

to obtain the linear PDE exhibited in the ELLPACK program of Figure 9. This differentiation is somewhat tedious and a better system would also do that, a MACSYMA program for this is given in [Rice and Boisvert, 1985]. 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 solution is displayed in Figure 10 and Figure 11 shows a contour plot of the difference between the third and fourth iterates.

```
EQUATION.
     (1.+uy(x,y)**2) uxx + (1.+ux(x,y)**2) uyy                  &
          - 2.*ux(x,y)*uy(x,y)  uxy                             &
     + 2.*(ux(x,y)*uyy(x,y) - uy(x,y)*uxy(x,y))  ux             &
     + 2.*(uy(x,y)*uxx(x,y) - ux(x,y)*uxy(x,y))  uy             &
  =    2.*(ux(x,y)*uyy(x,y) - uy(x,y)*uxy(x,y))*ux(x,y)        &
     + 2.*(uy(x,y)*uxx(x,y) - ux(x,y)*uxy(x,y))*uy(x,y)
BOUNDARY.
     u = bound(x,y) on x = 1.0,                           &
               y = 0.5 + p       for p = 0.0 to 3.5
     u = bound(x,y) on x = 1.0 + p,                       &
               y = 4.0           for p = 0.0 to 3.0
     u = bound(x,y) on x = 4.0 + .1*p*(p-4.5)**2,         &
               y = 4.0 - p       for p = 0.0 to 4.5
     u = bound(x,y) on x = 4.0 - p,                       &
               y = -0.5 + p/3.   for p = 0.0 to 3.0

GRID.
     9 x points 1.0 to 5.5  $  9 y points -0.5 to 4.0

TRIPLE.  set ( u = gessu )

FORTRAN.
     do 100 it = 1, 5
          call save(r1unkn,i1neqn)
discretization.   collocation
solution.         band ge
output.           max(diffu)

FORTRAN.
     100 continue
OUTPUT.  table(u) $ plot(u)
```

Figure 9. An ELLPACK program for applying Newton's method to solve the Plateau problem.

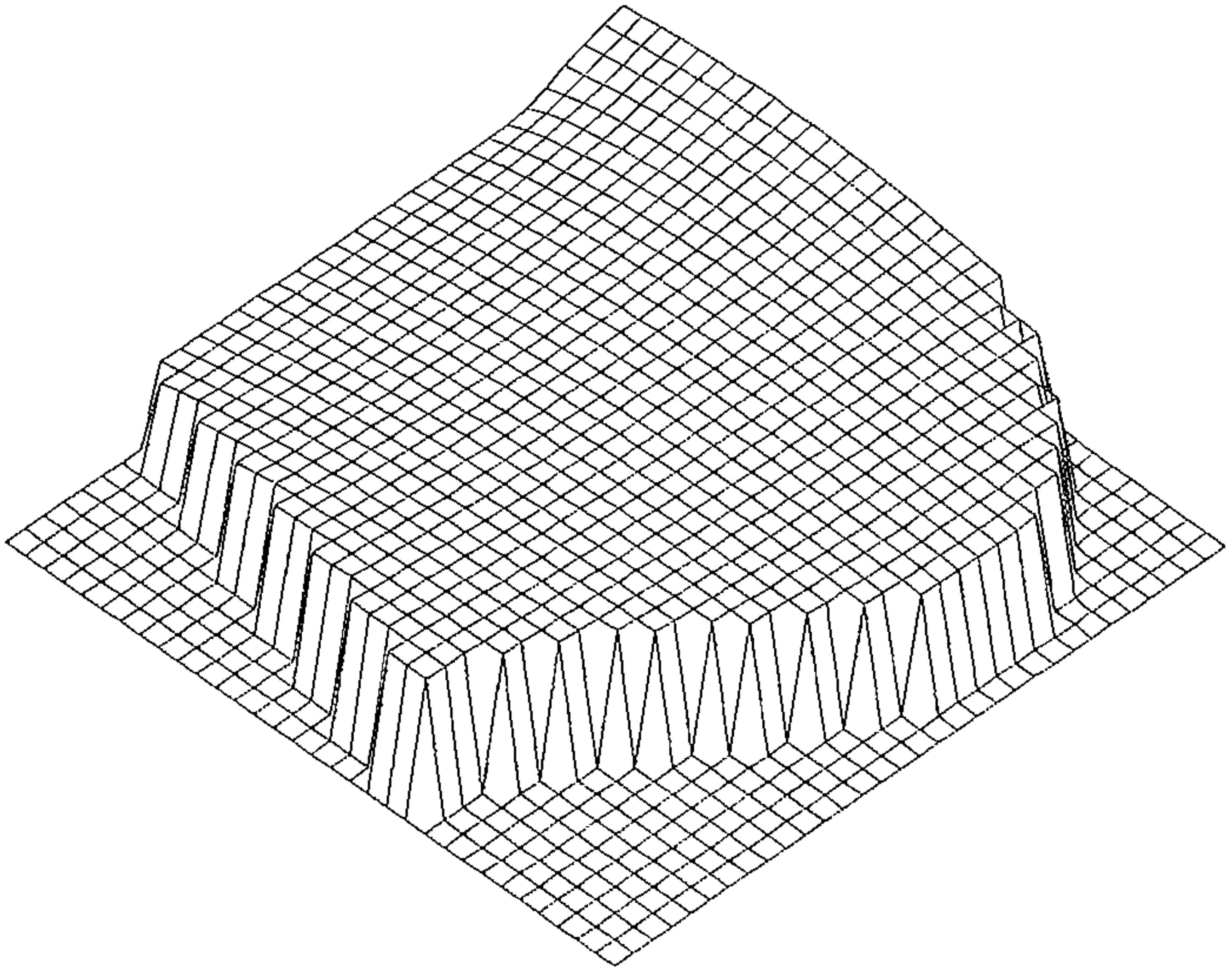


Figure 10. A view of the surface defined by the solution computed by the program in Figure 9.

ELLPACK OUTPUT

diffu

Contour	Value
1	-0.24E-05
2	-0.20E-05
3	-0.17E-05
4	-0.13E-05
5	-0.98E-06
6	-0.63E-06
7	-0.28E-06
8	0.73E-07
9	0.42E-06
10	0.77E-06

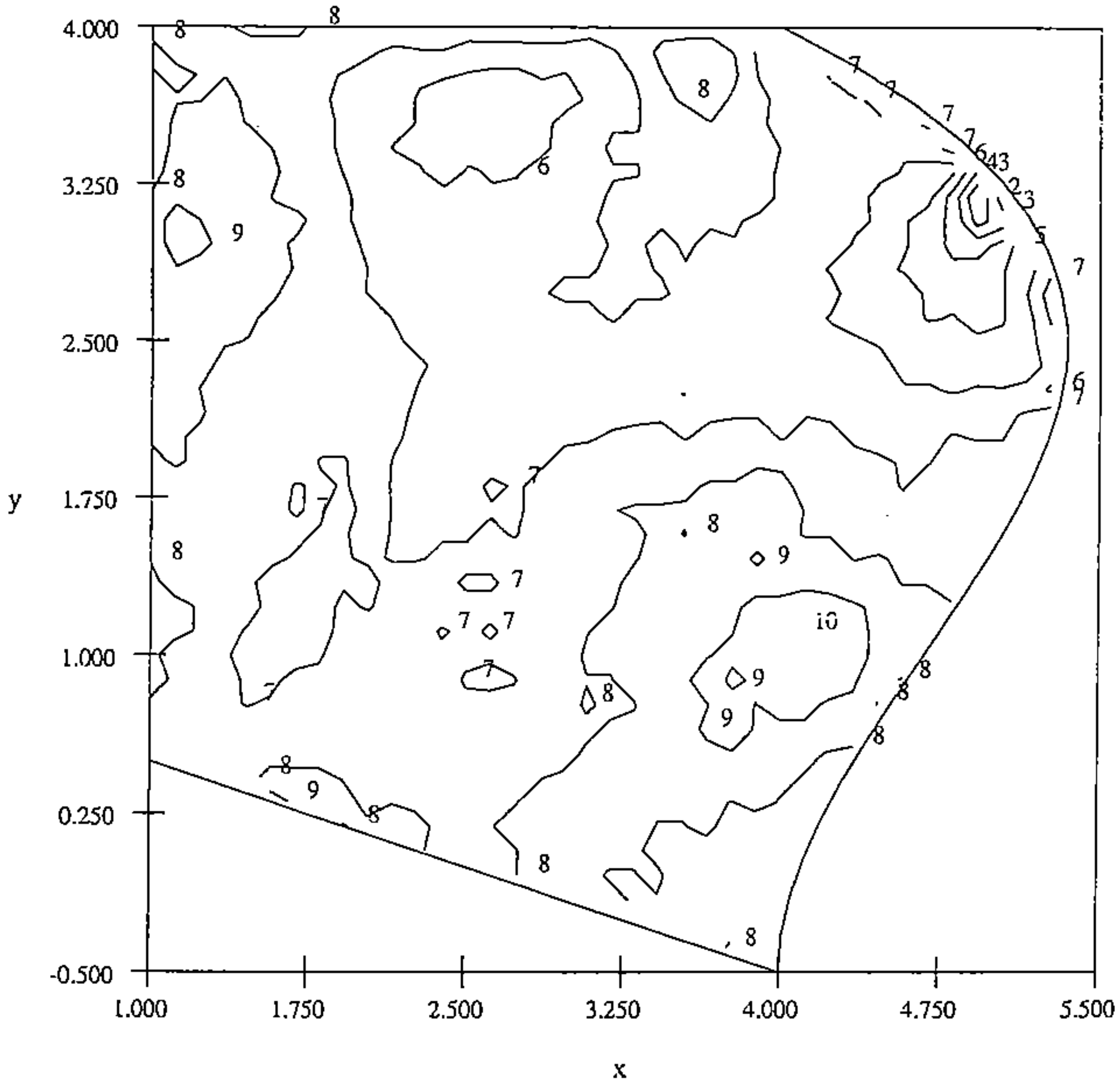


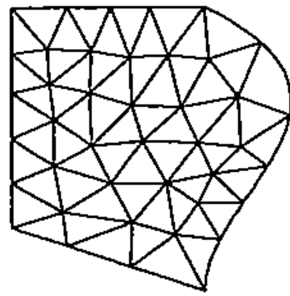
Figure 11. Contour plot of the difference between the third and fourth iterates in Newton's method. The maximum difference here is 2.4×10^{-5} and that between the fourth and fifth iterates is 5×10^{-7}.

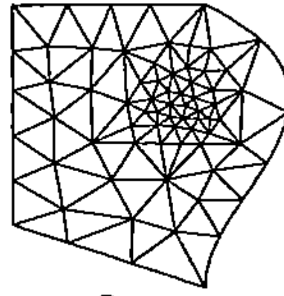
Our final point in the software and programming area concerns the role *regularity* in data structures, in algorithms and in programs. Clever programmers and hardware designers can do a lot of special things to exploit special situations. This exploitation is usually achieved at the cost of more complex 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.

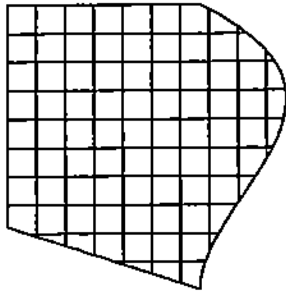
This view is illustrated by an example in discretizing a domain. Figure 12 shows a physical domain that has been partitioned in six ways for a problem with difficulties near the right boundary:

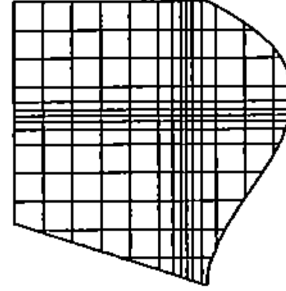
(A) A fine triangulation of a common type
(B) A fine, uniform, rectangular overlay grid
(C) Mapping the domain to a rectangle and inducing a logically rectangular partition
(D) Triangulation adapted to the difficulty
(E) Rectangular overlay grid adapted to the difficulty
(F) Logically rectangular partion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

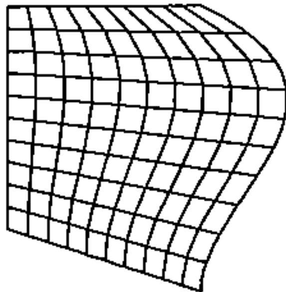
We believe that the irregular triangulations do not provide any execution time advantage over the more regular partitions (note that one can do regular triangulation if one wants). On the other hand, we also believe that the uniformly spaced partitions are too simple and have too large an execution time penalty. We believe the adaption will pay off. The logically rectangular one 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.











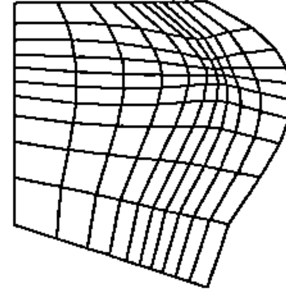


Figure 12. Six ways to partition a domain showing ways to achieve regularity and to adapt to a difficulty. The letters A through F refer to the discussion in the text.

VI. BRIEF TAXONOMY OF PARALLEL METHODS FOR PDEs

This taxonomy assumes that the reader is familiar with the multitude of methods for solving PDEs (see [Birkhoff and Lynch, 1984], [Gladwell and Wait, 1979] [McBryan and Van de Velde, 1986] and [Ortega and Voight, 1985] for recent, systematic treatises). Specific methods are mentioned as examples from larger classes and we do not provide descriptions of these methods here.

We classify methods by the way three basic procedures are used for defining them:

1. **Partitioning.** The unknowns in the problem are divided into groups. The criterion for grouping may be geometric (e.g., substructuring), or a matrix property (e.g., columns of an array) or physical (e.g., neighborhoods of singularities, boundary layers or shock waves). Different partitions may be used at different times (e.g., ADI methods) and partitions may be further partitioned (e.g., nested dissection or multigrid). Partitioning is almost always conducive to the use of parallelism.

2. **Discretization.** The continuous PDE problem is replaced by a finite problem with a set of real numbers as unknowns. The two most common techniques are finite differences and basis functions (e.g., finite elements, polynomials or series expansions). Finite difference methods and basis functions with local support (e.g., piecewise polynomials) are very good for parallel methods in the discretization phase as these computations are highly independent of one another. Discretization are also divisible into high order and low order methods. High order methods are advantageous for parallelism because more of the work can be shifted to the discretization phase of the computation.

3. **Iteration.** Iteration is an all-purpose technique to handle difficulties (e.g., non-linearities, very large problems or time dependence). Iteration is inherently sequential and thus not very suitable for exploiting parallelism. Yet iteration is essential to solving many, if not most, PDEs, one should concentrate on introducing iteration so that the number of iterations is very small and the work in each iteration is large and easily divisible into parallel components.

There is a fourth basic procedure that is more limited in its applicability and thus usually appears at the "lower levels" of the method:

4. **Solve Directly.** One applies a procedure that exactly solves a problem. By far the most common example in PDEs is some form of elimination to solve a linear system of equations, other examples are FFT methods and symbolic integration.

Figure 13 shows a schematic of our taxonomy of PDE methods. The letters P, D, I and S are used to denote the four procedures, Partitioning, Discretization, Iteration, and Solve, respectively. The classes are defined in terms of the order in which the method is defined. There are some special methods for special problems (e.g., FFT for Laplace's equation) and some methods for linear, steady state PDEs do not use iteration. But, as Figure 13 indicates, most general methods involve all three of the basic procedures. By far the most common methods in current use are those that start with Discretization (corresponding to boxes 3 and 4).

We suggest that there are real advantages to doing the partition or iteration procedures first. It is much easier to use information about the physical domain data structure as this stage of developing the method. It is much easier to see how to apply Newton's method to a PDE than to the thousands of equations generated by a discretization. If Newton's method is applied directly to the PDE it is still easy to see how the physical domain data structure interacts with the linear PDE generated by Newton's methods.

The effective use of parallelism depends in many cases on an astute global organization of the computations. We believe that the most favorable approach to discover such methods is to follow paths 2 or 5, those where the discretization is delayed to the last.

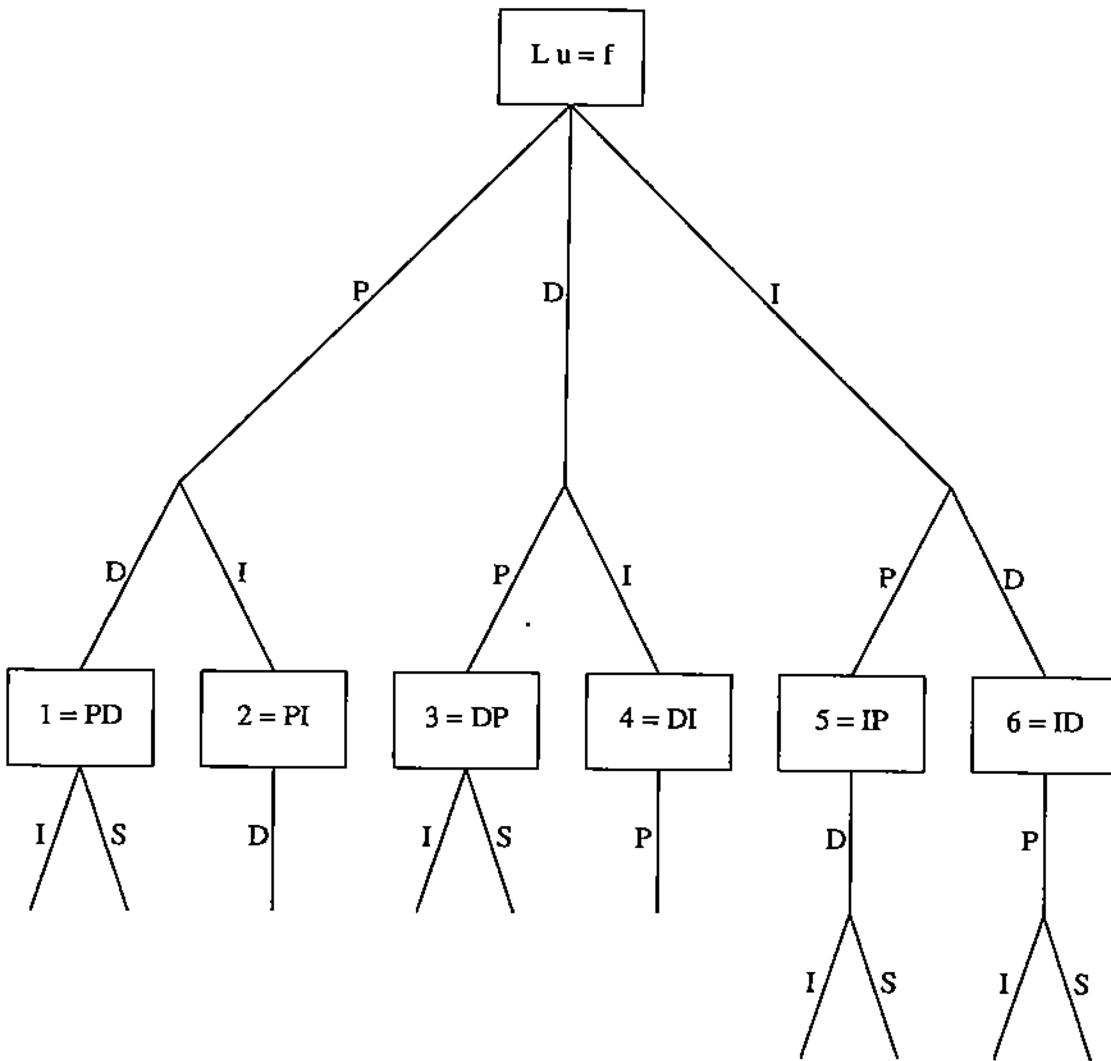


Figure 13. Taxonomy of methods for PDEs. The notation P = Partition, D = Discretization, I = Iterate and S = Solve Directly is used.

VII. REFERENCES

G. Birkhoff and R. Lynch, *Numerical Solution of Elliptic Problems*, SIAM, Philadelphia, 1984.

J.J. Dongarra and D.C. Sorensen, Performance and library issues for mathematical software on high performance computers. In *New Computing Environments: Parallel, Vector and Systolic* (A. Wouk, ed.) SIAM Publications, Philadelphia, 1986, pp. 110–133.

J.A. George, Nested dissection of a regular finite element mesh. SIAM J. Numer. Anal., 10(1973) pp. 345–363.

I. Gladwell and R. Wait, *A Survey of Numerical Methods for Partial Differential Equations*, Clarendon Press, Oxford, 1979.

R.W. Hockney and C.R. Jesshope, *Parallel Computers*, Chapter 5. Adam Hilger, Bristol, 1981.

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.

O.A. McBryan and E.F. Van de Velde, Parallel algorithms for elliptic equations. In *New Computing Environments: Parallel, Vector and Systolic* (A. Wouk, ed.) SIAM Publications, Philadelphia, 1986, pp. 236–270.

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-Verlag, New York, 1985.

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

A.H. Sameh, Numerical parallel algorithms - A survey. In *High Speed Computer and Algorithm Organizations*, Academic Press, New York, 1983, pp. 207–228.

Y. Umetami, M. Tsuji, K. Iwasawa and H. Hirayama, DEQSOL: A numerical simulation language for vector/parallel processors, Technical Report, Hitachi Ltd., Tokyo, 1984.