01 May 1994

Designing a New Programming Methodology for Optimizing Array Accesses in Complex Scientific Problems

Larry Coffin

Designing a New Programming Methodology for Optimizing Array Accesses in Complex Scientific Problems

Larry Coffin
Computer Science Department

Many problems of interest to scientists and engineers, such as fluid flow and stress analysis, require the solution of complex PDEs. Often the solution requires discretizing the physical domain into a mesh or grid then stepping from a given initial state towards some final state using small incremental time steps. Grid sizes can reach several thousand to hundreds of thousands of elements and the number of time steps required to solve the problem can vary from several thousand to tens of thousands or more. Traditional programming techniques are unable to take advantage of the non-random grid access patterns and generally result in array accesses that are computationally expensive. Our research has produced a method of programming based on the method of Psi Calculus (Mullin, 1988) that results in faster access times -- producing programs that run significantly faster than programs written in a traditional style.

INTRODUCTION

Many real world problems on which scientists and engineers spend much time and computational resources often require the solution of complex partial differential equations, or PDEs. Many problems, such as air flow around an airplane wing or the stress analysis of a bridge strut, are modeled on a computer in order to try to understand them in more detail than is possible by studying a real system. Other problems, such as air and fuel mixture and combustion in a hypersonic jet engine can not be studied in the real world and must be modeled on a computer. The solution to these problems often requires discretizing the physical domain by use of a mesh or grid then stepping through time from some initial state in order to reach some final state or just to study the changes in the system over time. Grid sizes can easily exceed several hundred thousand elements -- the larger the grid size, the more accurate the solution. The number of time steps can vary from several thousand to tens of thousands or more -- smaller

time steps can also increase the accuracy of the solution over a given time span or can provide a better simulation of a dynamic system. Both grid size and the number of time steps are limited by the amount of memory and CPU time available. A typical problem with grid sizes of over 300,000 elements and 20,000 time steps can take from 15 to 20 hours of CPU time on a Cray YMP (Riggins, 1992). These problems are extremely important and their solution can greatly benefit the advancement of science. However, these problems are also so restricted by lack of memory and time limitations that any improvement in computational efficiency can greatly extend the capability of computers to solve larger an more accurate problems.

GOALS

Our work was based on several goals: first, we wanted to familiarize ourselves with PDEs. Second, we wanted to develop a programming technique which would speed up programs which rely on the solution of complex PDEs by means of an explicit iterative solution. And thirdly, we wanted to create a technique by which programs based on an arbitrary number of dimensions can be solved.

We decided to focus on the PDE for heat flow or heat transport because it is a simple problem involving one component, heat. The PDE can be solved with an iterative technique similar to more complex PDEs and can also be solved directly, allowing us to compare answers and check the validity of our technique.

BASIC C PROGRAMMING LANGUAGE SYNTAX

First, for those unfamiliar with the C programming language, some of the syntax as used in the paper:

• Arrays of size N elements are accessed with an index from 0 to N-1.

• Multi-dimensional arrays are stored in row-major format rather than in column-major format (as in Fortran). See Figure 1.

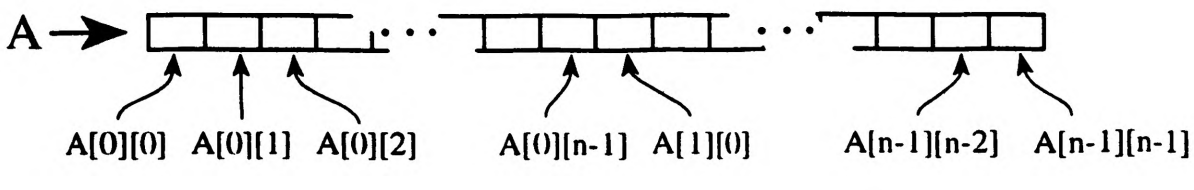

Figure 1: Storage in memory of array elements in a row-major format.

• C provides address storage and manipulation via pointers and pointer arithmetic. If "ptr" is a

pointer to (address of) an element in memory, "ptr + 3" is a pointer to the third element beyond "ptr". Note that "ptr + 3" involves an implicit multiplication by the size of an element so that if the value of "ptr" is 1000 and the size of an element is six bytes, then the value of "ptr + 3" is 1000 + 6*3 = 1018, not 1003.

• The value of the element that a pointer points to can be found by the dereferencing operator. In C this is a prefixed asterisk -- "*ptr" is the value of the element pointed to by "ptr". Likewise "*(ptr + 3)" is the value of the third element beyond "ptr".

• In C, arrays can be accessed in two ways -- by index or by offset via the use of pointer arithmetic. Writing "A[5]" is the same as "*(A + 5)" -- the name of an array is taken to be a pointer to the beginning (the first element) of the array in memory.

• For loops (DO loops in Fortran) are written as:

```
for (Initial; Test; Modify) {
      Body
}
```

which can be thought of as:

```
do Initial statement
while Test is true
      do Body statement(s)
      do Modify statement
end of while loop
```

• "=" is the assignment operator.

GENERAL PROBLEM

We initially decided to focus on PDEs because the solution to these problems often requires the use of extremely large multi-dimensional arrays. In a computer system, memory exists only in the form of a linear array of bytes. Therefore, in order to handle multi-dimensional arrays, the compiler must supply some sort of function that maps the multi-dimensional array to the linear array of physical memory. For arrays of a single dimension, the mapping is straightforward -- the address of an element can be computed by

$$\text{address} = \text{start} + \text{index} * \text{size of an element in bytes.} \qquad (1)$$

For example, given an array of N elements whose size is four bytes and a starting address of 1000, to find the 63rd element (index = 62 if index goes from 0 to N-1) we use:

$$\text{address} = 1000 + 62*4 = 1248. \qquad (2)$$

For a two dimensional M by N array and elements of size S bytes, the formula for finding an element indexed by i and j becomes:

$$\text{address} = \text{start} + S*(i*N + j). \qquad (3)$$

This can be extended to an N dimensional array with dimensions $D_0 x D_1 x D_2 x ... x D_{n-1}$ with elements of size S. For an element indexed by $I_0, I_1, I_2, ..., I_{n-1}$ the formula becomes:

$$\begin{aligned}\text{address} = \text{start} + S*(I_0*(D_1*D_2*...*D_{n-1}) + I_1*(D_2*D_3*...*D_{n-1}) \\ + ... + I_{n-3}*(D_{n-2}*D_{n-1}) + I_{n-2}*D_{n-1} + I_{n-1}).\end{aligned} \qquad (4)$$

Even if the product terms $P_0 = (D_1*D_2*...*D_{n-1}), P_1 = (D_2*D_3*...*D_{n-1}), ... P_{n-2} = D_{n-1}, P_{n-1} = 1$ are precalculated, the formula will be:

$$\text{address} = \text{start} + S*(I_0*P_0 + I_1*P_1 + ... + I_{n-2}*P_{n-2} + I_{n-1}). \qquad (5)$$

This still involves N multiplications and N additions to find a single element! So as the number of dimensions increases, the cost of randomly accessing any element becomes large.

Often though, elements are not accessed randomly. Consider for example the initialization of a linear array:

```
for (i = 0; i < N; i = i+1){
      array[i] = 1;
}
```

All elements are accessed in an increasing fashion. Rather than calculating the address of each element from the starting address and its index, we know that the next element that we want to access is located S bytes beyond the current element. The formula thus becomes:

$$\text{address of next element} = \text{current address} + S. \qquad (6)$$

This formula works for arrays of any dimension as long as the elements are accessed in the order in which

they are stored (i.e. in a row-major order language, the "column" index increases faster than the "row" index). Most optimizing compilers will automatically convert:

```
for (i = 0; i < M; i = i+1) {
      for (j = 0; j < N; j = j+1){
            array[i][j] = 1;
      }
}
```

to:

```
end = array + M*N;        (The address of the first byte beyond the end of the array)
for (ptr = array; ptr < end; ptr = ptr+1){
      *ptr = 1;
}
```

This involves only two multiplications, M*N+1 additions and M*N comparisons vs. 2*M*N multiplications, 3*M*N + M additions and M*N + M comparisons.

However, as loops become more complex, compilers are less able to able extract the non-randomness and reduce them to more efficient loops. For example, consider the main loop as found in the solution to the 3D heat transfer PDE:

```
for (i = 1; i < L-1; i = i+1){
   for (j = 1; j < M-1; j = j+1){
      for (k = 1; k < N-1; k = k+1){
         uprime[i][j][k] = u[i][j][k]
            + λ*C*(u[i-1][j][k] + u[i+1][j][k] + u[i][j-1][k] + u[i][j+1][k]
                        + u[i][j][k-1] + u[i][j][k+1] - 6*u[i][j][k]);
      }
   }
}
```

There are two problems in trying to optimize this, first all the elements are not accessed. All elements whose i, j, or k index equals zero or where i = L-1, j = M-1, or k = N-1 are skipped. These are the boundary elements -- the elements that lie on the boundary of the physical domain. The second problem is that at each iteration multiple elements are accessed whose addresses are based on increments or decrements to the indexes i, j, and k (i±1, j±1, k±1). These extra elements are based on what is termed a "mask". These elements are the "neighbors" of the central element (index = i, j, k). Figure 2 shows masks for 1, 2 and

3D problems.

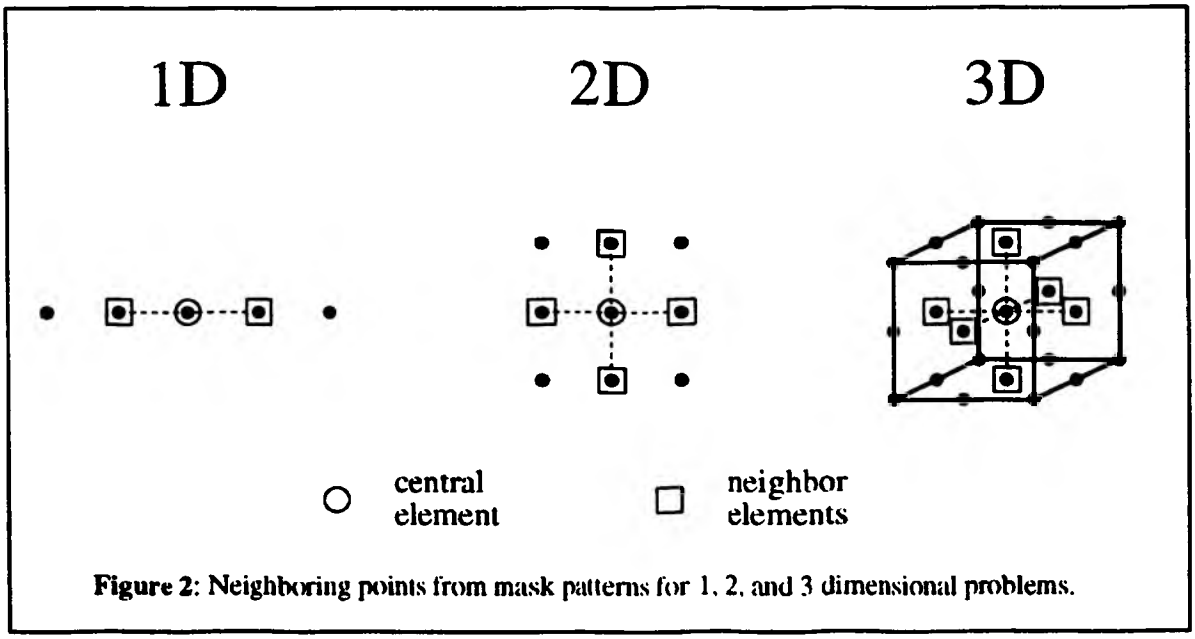

Figure 2: Neighboring points from mask patterns for 1, 2, and 3 dimensional problems.

In trying to design a programming methodology for optimizing these types of array accesses, we decided that we could break the problem down into two parts: finding the address of the central element, then once we have found this, finding the addresses of its neighbors.

To reduce the computations necessary to access the non-boundary elements, a second array was created that contained the offsets from the start of the array of the non-boundary elements which are calculated once at the start of the program. Thus the algorithm for accessing all the central elements becomes:

```
for (i = 0; i < (L-2)*(M-2)*(N-2); i = i+1){
      *(uprime + offset[i]) = *(u + offset[i]) ...;
}
```

In order to access the central element's neighbors we looked at the extra elements we wanted to access. If we take the six elements from the previous 3D example: u[i±1][j][k], u[i][j±1][k], u[i][j][k±1] and the central element: u[i][j][k] and convert these indexed representations to the corresponding addresses using offsets we get:

$$
\begin{aligned}
&\text{u[i][j][k]} && = u + i*M*N + j*N + k && = c \\
&\text{u[i±1][j][k]} && = u + (i\pm1)*M*N + j*N + k = u + i*M*N + j*N + k \pm M*N && = c \pm M*N \\
&\text{u[i][j±1][k]} && = u + i*M*N + (j\pm1)*N + k = u + i*M*N + j*N + k \pm N && = c \pm N \\
&\text{u[i][j][k±1]} && = u + i*M*N + j*N + (k\pm1) = u + i*M*N + j*N + k \pm 1 && = c \pm 1
\end{aligned}
\qquad (7)
$$

These differences are equal to the number of elements in the sub-array for the dimension whose index

is modified. So for the neighbor in the yth dimension, i.e. $I_y \pm 1$, the two elements can be found by $c \pm P_y$. So once we have the address of the central element, it only requires a single addition or subtraction to get any of its neighbors. The previous example can be rewritten as:

```
for (i = 0; i < (L-2)*(M-2)*(N-2); i = i+1){
      new = uprime + offset[i];
      old = u + offset[i];
      *new = *old + λ*C*( *(old-P[0]) + *(old+P[0]) + *(old-P[1]) + *(old+P[1])
                                + *(old-P[2]) + *(old+P[2]) - 6*(*old));
}
```

The final aspect of the problem that we looked at was the problem of handling an arbitrary number of dimensions. By writing programs that can handle an arbitrary number of dimensions, time can be saved by eliminating the need to modify the program if the number of needed dimensions changes. We looked at the solution equations to the heat transfer PDE for 1, 2 and 3 dimensions (see equations number eight) and from them we designed and derived a generic program for a dimension independent equation using the Psi Calculus (Mullin, 1988):

$$\begin{aligned}
&1D \rightarrow u'_i = u_i + \lambda C(u_{i-1} + u_{i+1} - 2u_i) \\
&2D \rightarrow u'_{i,j} = u_{i,j} + \lambda C(u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - 4u_{i,j}) \\
&3D \rightarrow u'_{i,j,k} = u_{i,j,k} + \lambda C(u_{i-1,j,k} + u_{i+1,j,k} + u_{i,j-1,k} + u_{i,j+1,k} + u_{i,j,k-1} + u_{i,j,k+1} - 6u_{i,j,k})
\end{aligned} \tag{8}$$

If we use n to mean the number of dimensions and d_1, d_2, and d_3 to indicate subscripts for dimensions 1, 2, and 3 such that, in 3D, $u_{i+1,j,k} \equiv u_{d_1+1}$ these equations can be rewritten as:

$$\begin{aligned}
&1D \rightarrow n = 1,\ u' = u + \lambda C(u_{d_1-1} + u_{d_1+1} \\
&\qquad\qquad - 2nu) \\
&2D \rightarrow n = 2,\ u' = u + \lambda C(u_{d_1-1} + u_{d_1+1} \\
&\qquad\qquad + u_{d_2-1} + u_{d_2+1} \\
&\qquad\qquad - 2nu) \\
&3D \rightarrow n = 3,\ u' = u + \lambda C(u_{d_1-1} + u_{d_1+1} \\
&\qquad\qquad + u_{d_2-1} + u_{d_2+1} \\
&\qquad\qquad + u_{d_3-1} + u_{d_3+1} \\
&\qquad\qquad - 2nu)
\end{aligned} \tag{9}$$

In general we get:

$$u' = u + \lambda C(\sum^{n}_{q=1}(u_{d_q-1} + u_{d_q+1}) - 2nu) \quad (10)$$

Note that the address of $u_{d_q \pm 1}$ = address of $u \pm P_{q-1}$.

The final algorithm thus becomes:

```
num_elem = 1; (The number of non-boundry elements)
for (q = 0; q < num_dim; q = q+1){
      num_elem = num_elem * (D[q] - 2);
}
m2n = -2*num_elem;
lambda_C = lambda*C;
(time step loop) {
      for (i = 0; i < num_elem; i = i+1){
            off = offset[i]
            old = u + off;
            new = uprime + off;
            *new = m2n*(*old);
            for (q = 0; q < num_dim; q = q+1){
                  p = P[q];
                  *new = *new + ( *(old-p) + *(old+p));
            }
            *new = (*new)*lambda_C + *old;
      }
}
```

Since num_elem, m2n, and lambda_C are calculated only once, at each time step we do (3*num_dim+2)*num_elem fetches, (6*num_dim+5)*num_elem additions, (num_dim+1)*num_elem compares, (3*num_dim+5)*num_elem multiplies, and num_elem stores.

RESULTS

Several programs were written: three with the traditional method of array indexing for 1, 2, and 3 dimensional problems and one with the new method. For the program written with the new technique, the number of dimensions, the size and of the grid, as well as the number of time steps to iterate over were read in from an input file. For the other programs, these values were hard coded into the program. The

programs were run on NeXT workstations and the programs reported the CPU time they took to run. The resulting times were (in seconds):

1D

Size	Traditional	New
40	3.69	7.04
80	5.50	9.84
120	7.24	14.03
160	9.27	22.78

2D

Size	Traditional	New
40x40	23.27	25.02
80x80	128.69	107.09
120x120	425.81	243.45
160x160	535.80	447.55

3D

Size	Traditional	New
20x20x20	61.64	25.20
30x30x30	283.54	96.38
40x40x40	575.24	239.69
50x50x50	1417.76	475.00

As can be seen from the times, the traditional method was better for 1D problems, but the new method was, for the larger arrays, significantly faster for 2D problems and for all size arrays, much faster for 3D problems.

Of course, keeping an array of offset values uses up memory that could be used elsewhere. Assuming that we use long integers (0 to 2^{32}-1) to store offset values we will be using four bytes for each element of the main array (not counting boundary elements, but for an upper bound we can assume the 1 to 1 ratio of offset elements to grid elements). The solution to the problem requires that we have two main arrays u and u' to hold the current values and the values for the next time step. So, as in the case of the heat transfer problem that we worked with, if we are storing a single floating point value (also four bytes) in each element, the addition of the offset array will increase the memory usage by approximately 50% of the original, necessary memory usage. If, however, we store more information in each element, and typical PDEs (as in combustion/reaction problems) often store from three to nine or more values, requiring 12 to 36 bytes for single precision values or 24 to 72 bytes for double precision values. The extra memory required for the offset array will be from $4/_{24}$ = 17% (3 values, single precision) to $4/_{144}$ =

3% (9 values, double precision) or less of the necessary memory. For complex problems, the memory increase will be very small and more than offset by the increase in speed of the overall program.

ACKNOWLEDGMENTS

I would like to thank Dr. Lenore Mullin and Dr. Michael Hilgers for their work on this project and their advice, guidance and support for my participation in this and other research, and OURE for the opportunity to work on this research.

REFERENCES

Mullin, L. R., "A Mathematics of Arrays", Ph.D. dissertation, Syracuse University, Syracuse, NY, 1988.

Riggins, D., et. al., "Modeling of Supersonic Combustor Flows Using Parallel Computing", *Computing Systems in Engineering*, Vol. 3, Nos 1-4, pp. 217-229, 1992.