Lawrence Berkeley Laboratory

UNIVERSITY OF CALIFORNIA

Information and Computing Sciences Division

Parallel Computation of Automatic Differentiation Applied to Magnetic Field Calculations

R.L. Hinkins
(M.S. Project Report)

September 1994

RECEIVED
DEC 29 1994
OSTI

Prepared for the U.S. Department of Energy under Contract Number DE-AC03-76SF00098

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

DISCLAIMER

This document was prepared as an account of work sponsored by the United States Government. While this document is believed to contain correct information, neither the United States Government nor any agency thereof, nor The Regents of the University of California, nor any of their employees, makes any warranty, express or implied, or assumes any legal responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by its trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof, or The Regents of the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof, or The Regents of the University of California.

Lawrence Berkeley Laboratory is an equal opportunity employer.

Parallel Computation of Automatic Differentiation Applied to Magnetic Field Calculations

Ruth L. Hinkins

M.S. Project Report

Computer Science Department

University of California

and

Information and Computing Sciences Division

Lawrence Berkeley Laboratory

University of California

Berkeley CA 94720

September 1994

This work was supported by the Director of the Scientific Computing Staff, Office of Energy Research, of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098. Some of the equipment used was provided by NSF grant No. CDA-8722788.

~~DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED~~

JR

Abstract

We present a parallelization of an accelerator physics application to simulate magnetic fields in three dimensions. The problem involves the evaluation of high order derivatives with respect to two variables of a multivariate function. Automatic differentiation software had been used with some success, but the computation time was prohibitive. Our implementation runs on several platforms, including a network of workstations using PVM, a MasPar using MPFortran, and a CM-5 using CMFortran. A careful examination of the code led to several optimizations that improved its serial performance by a factor of 8.7. The parallelization produced further improvements, especially on the MasPar with a speedup factor of 620. As a result a problem that took six days on a SPARC 10/41 now runs in minutes on the MasPar, making it feasible for physicists at Lawrence Berkeley Laboratory to simulate larger magnets.

1 Introduction

Magnet designers have been trying to develop a new form for specifying the 3D fields produced by electromagnetic lens elements in particle accelerators that will be convenient for the dynamical computations required to evaluate the quality of the field. The behavior of beam-transport lines depends upon the characteristics of the magnetic fields used to guide and focus the beam. Several Fortran77 programs had been written to do the necessary computations. Using automatic differentiation in the calculation was favored because of the accuracy, but the running time of the code using this method was prohibitive. Lawrence Berkeley Laboratory (LBL) recently acquired a MasPar MP-2,^[MasP1] a SIMD machine, which could take advantage of all the data parallelism in the field calculation. Also the use of automatic differentiation, while an old technique, is currently experiencing a surge of new interest. The code was ported to several architectures to see what performance could be obtained by using distributed and parallel techniques and how much effort the port required.

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

2 Differentiation

The technique of differentiation is fundamental in scientific applications. Many programs that perform differentiation require that the user provide code to compute the derivatives of a function as well as the function itself. Basic requisites for differentiation [Iri91] are that the algorithm must be fast, free from truncation error, carried out automatically, and able to estimate rounding errors. To be fast, the complexity of the differentiation algorithm should not be high compared with the complexity of what is being differentiated. For differentiation to be *automatic*, it should not require much more work than writing a program to evaluate the function.

2.1 Differentiation Techniques

When the user is unable or unwilling to supply code for the derivatives, the application may have to rely on numerical differentiation which is a divided difference calculation. In addition to eliminating the need for user code to calculate derivatives, numerical techniques do not require that the function be expressible symbolically. However approximations made by divided differences suffer from truncation errors, which are especially problematic for higher order derivatives. The magnetic field calculation in our application requires derivatives of order 17 or higher.

If a function can be represented as a formula, symbolic differentiation can be used. Symbolic algebra systems are available to do this. The expression for the derivatives may be much more complicated than the expression for the original function, and may require much more time and space than numerical differentiation. These symbolic representations for the derivatives have to be coded in an appropriate language for the program that needs them. Symbolic differentiation needs to optimize to prevent repeated evaluation of common subexpressions and must generate the required code to compete with automatic differentiation.

Automatic differentiation is the process of producing the values of a function's derivatives from a computer language representation of the function. Frequently the function to be differentiated is not a single expression but an algorithm with conditional branches. A computer program that evaluates the function is a complete description of the function; automatic differentiation can use this representation to give derivatives that are free from the truncation

error produced by numerical calculation of derivatives. No additional input is required to describe the derivatives, as in the case of symbolic differentiation. It can be more efficient in both computer resources and human effort than symbolic or numerical differentiation.

Almost as soon as programmable computers were available, it was observed that they could be made to automatically produce derivatives by application of the chain rule.^[Grie91] The chain rule may be applied bottom-up or top-down. The bottom-up (forward, covariant) method of automatic differentiation was proposed by R.E. Wengert in 1964. In the forward mode, derivative values propagate from the independent variables to the dependent variables. The top-down (reverse, backward, covariant) method propagates from the dependent to the independent variables. Which method is better depends upon the application. Automatic differentiation software has been developed which use either or both techniques.

2.2 Software for Automatic Differentiation

Automatic differentiation tools have been classified by David Juedes^[Jued91] based on the level of integration of automatic differentiation provided by the source language. He has classified twenty-nine tools as Elemental, Extensional, Integral, Operational or Symbolic.

Elemental tools assume every function can be decomposed into a sequence of elementary operations. The operations are implemented as procedures which, given the input arguments and their derivatives, return the result and its derivatives. Transformation from the source language is performed manually by generating the appropriate procedure calls.

Extensional tools automate the process of decomposing complicated right-hand sides of equations into sequences of elementary operations. Precompilers are typically used to transform the language extensions back to the original language. The largest number of tools were classified in this category; the software originally used by our application was this type.

Integral tools can automatically differentiate functions that have been integrated within the language. Since this is a new language rather than an extension, the transformation is done transparently by the compiler or interpreter.

Operational tools require a programming language that provides operator overloading. A new data type is defined for which differentiation can be easily performed. The programming language interprets the new data type with standard arithmetic operators and functions overloaded.

Symbolic tools use a symbolic representation of the function to produce a representation of the derivatives. The tool may either process a symbolic representation of the function at run time or use a computer algebra system to produce the differentiated function that is translated to a standard programming language.

Juedes classified 29 tools with this taxonomy. For each tool he also noted what degree of derivatives can be computed and whether the mode of derivation is forward or backward.

The software used by the original version of our application was written by Martin Berz who has used automatic differentiation in various accelerator physics applications^[Berz89]. His software is classified as an Extensional tool using the forward method of calculating derivatives. The precompiler DAFOR^[Berz90] converts the extended language program, which allows a new data type, to standard Fortran code.

In this system, space is allocated and function calls generated to convert the extended language representation of the function value computation to include computation of the derivatives. The DAFOR reference manual states that the elementary operations are highly optimized, a sophisticated algorithm for multiplication keeps the bookkeeping overhead to 30% of the computation time, and vanishing entries in vectors do not contribute to computation time. The number of variables and order of derivation is limited only by time, space, and rounding errors.

Our new code to compute derivatives also uses the forward method, but is still at the Elemental tool stage where we manually make the subroutine calls to perform arithmetic operations or evaluated library functions, having no precompiler. It is our intentions to make it an Operational tool with operator and function overloading.

2.3 Differentiation Arithmetic

Given the code to evaluate a function, we want to produce the function's derivatives. The following is a short explanation of why evaluation of the function in this arithmetic produces derivative values as well as the function value.

For a power series $a(\xi) = \sum_{k=0}^{\infty} a_k \xi^k$ addition, subtraction, multiplication and derivation can be defined on the infinite vector of coefficients $a = (a_0, a_1, a_2, \dots, a_n, \dots)$. Division is defined in terms of multiplication, and $c = a/b$ is uniquely determined only if $b_0 \neq 0$.

If for some function $f(\xi)$ the formal power series $f(\xi) = \sum_{k=0}^{\infty} a_k \xi^k$

has been computed using power series arithmetic and f is differentiable $k+1$ times, then by Taylor's theorem,

$$a_k = \frac{1}{k!} f^{(k)}(0), k = 0, 1, \dots, n.$$

Since the derivatives of $f(\xi)$ have been obtained at $\xi = 0$ by evaluating f using the formal power series arithmetic, Louis Rall calls this a differentiation arithmetic [Rall91].

This method is valid for multivariate functions and at arbitrary expansion points. Computations are made with finite vectors corresponding to the Taylor polynomials. Viewing automatic differentiation as the evaluation of the function in a differentiation arithmetic suggests implementation of the forward mode by operator and function overloading in some appropriate language, such as C++, Ada, Pascal-XSC or Fortran 90.

2.4 Example of Forward Automatic Differentiation Arithmetic

To compute second order derivatives for a function of 2 variables, requires 6 components. Note that the computations increase rapidly as the order of derivation increases: order 5 has 21 components; order 17 has 171 components.

The components are

$$(b_{0,0}, b_{1,0}, b_{0,1}, b_{2,0}, b_{1,1}, b_{0,2}) \text{ where } b_{n,m} = \frac{1}{n!m!} \frac{\partial^{n+m}}{\partial x^n \partial y^m} f(x,y) \Big|_{x=x_0, y=y_0} .$$

For $f(x,y) = \frac{3xy}{\sqrt{x^2+y}}$ evaluate at $x=2, y=5$

$x = (2,1,0,0,0,0)$ Taylor representation of x at $x=2$

$y = (5,0,1,0,0,0)$ Taylor representation of y at $y=5$

$xy = (10,5,2,0,1,0)$ multiply

$3xy = (30,15,6,0,3,0)$ multiply

$x^2 = (4,4,0,1,0,0)$ multiply

$x^2 + y = (9,4,1,1,0,0)$ add

$\sqrt{x^2 + y} = (3, \frac{2}{3}, \frac{1}{6}, \frac{5}{54}, -\frac{1}{27}, -\frac{1}{216})$ sqrt

$\frac{3xy}{\sqrt{x^2 + y}} = (10, \frac{25}{9}, \frac{13}{9}, -\frac{50}{54}, \frac{35}{54}, -\frac{7}{108})$ divide

2.4 Work by Others in Parallel Automatic Differentiation

While our application of forward automatic differentiation had enough parallelism just in the data, there are obvious parallel opportunities within the forward calculation, vector arithmetic and independent subexpressions. Dixon has done a forward calculation of gradients in parallel.[Dix87a] Rall notes that, although applying the chain rule in a backward direction is often more efficient for calculating partial derivatives in serial mode, forward evaluation could be better if given enough parallelism.[Rall91] However, from the literature it seems that for most problems, the reverse mode is better. Since the reverse mode is frequently implemented by overloading the function evaluation to produce a trace of the computation and then generating a computation graph, the computation graph is already available and can be used to exploit parallelism.

Bischof exploits parallelism in the reverse mode by independent passes through the computation graph and by splitting the

computational graph.[Bisc91a] Parallelization of the computations associated with each graph node is too fine grained to be effective. Bischof and other researchers have shown that parallelism can be exploited in automatic differentiation in a way that is transparent to the user.

There is a tradeoff between making automatic differentiation efficient and making it completely automatic, requiring nothing from the user but code to evaluate the function. Efficient code utilizes special user knowledge about a particular application, and it may require that certain derivatives are provided explicitly. Bischof[Bisc91b] says we must exploit user intuition about parallelism, that the key to efficient automatic differentiation implementation on high-performance computers is to find better ways to incorporate user and/or compile-time information about the behavior of the program.

3 Application Description

Accelerator physicists need to model 3D fields produced by electromagnetic lens elements in the region interior to coil windings. In figure 1, which presents the cross section of a typical quadrupole, are shown the field B from the winding currents I and the force F that comes from the motion of a particle of charge q , velocity v through the field. Our focus here is to calculate the magnetic field B . It has been shown[Casp92] that in the region near the z axis the 3 dimensional magnetic field components of a multipole magnet can be expressed in terms of "harmonic components" proportional to $\sin(n\theta)$ or $\cos(n\theta)$ of the azimuthal angle.

The r, z dependence of any such component can then be expressed in powers of r times functions $A_n(z)$ and their derivatives. Physicists believe that this form for expressing the field will be an efficient way to transfer field information. Once the functions and their derivatives have been determined, they can be used to calculate the magnetic field anywhere inside the curl-free divergence-free region near $r=0$.

Figure 1, X Y Cross Section of a Typical Quadrupole, 1st Quadrant

When the magnetic field cannot be expressed in terms of analytical functions of Z , the $A_n(z)$ values and their derivatives can be computed using the law of Biot-Savart (or Ampere). The Biot-Savart formula gives the contribution to the field at a point by the current in an individual straight wire segment. The coil windings then are considered as many short wire segments, and the contributions of each short segment are added to get the total field at any point.

The physicists have tried two methods of computing $A_n(z)$. [Caspi91] The first used Biot-Savart to compute field values on an arc chosen so that a Fourier analysis could be conveniently performed. The harmonic components were then numerically differentiated with respect to z . This method suffers from truncation errors for high orders of derivatives. The second method using automatic differentiation will now be described.

3.1 Biot & Savart field calculation

The magnetic field produced by current in the cables of a magnet element can be computed in the case of complete circuits using the law of Biot & Savart which is

$$\vec{B} = \frac{\mu_0}{4\pi} \oint \frac{J d\vec{l} \times \vec{r}_{c \rightarrow p}}{r^3}$$

where

J is the magnitude of the current density in the direction $d\vec{l}$ tangent to the wire;

$\vec{r}$ is a vector from the wire to the field point;

$\mu_0 = \frac{4\pi}{10}$ for r in centimeters, current in amperes and field in gauss.

To accommodate curved and twisted cables, individual cables are broken into short blocks with quadrilateral planar end faces. The block current is represented by a small number of straight wires extending between the end faces. Sometimes the wires used in the calculation are consistent with the actual wires in the cable. Sometimes they are approximated by assuming wires at the locations appropriate for 2D Gaussian integration, with wire currents proportional to their corresponding Gaussian weighting factors.

Now let us compute the contribution to the field at point p from current I flowing in a short segment of a wire from a to b .

Let

$$p = (p_x, p_y, p_z)$$

be a point where the field functions are required and

$$a = (a_x, a_y, a_z)$$

$$b = (b_x, b_y, b_z)$$

be the endpoints of a wire

Then $m = (m_x, m_y, m_z) = \frac{a+b}{2}$ is the midpoint of the wire

$f = (f_x, f_y, f_z) = b - m$ is a vector from the wire end to the middle

$k = p - m$ is a vector from the wire midpoint to the field point

$g = a - p$ is a vector from the field point to the wire endpoint a

$h = b - p$ is a vector from the field point to the wire endpoint b

$t = k \times f$ is the cross product, $t_y = (p_z - m_z)d_x - (p_x - m_x)d_z$

To evaluate B_y (where the B_x and B_z components are similar with cyclic permutation) we have

$$B_y = -t_y \frac{\mu_0}{4\pi} I \int_{-1}^1 \frac{ds}{(k - sf)^3} = \frac{\mu_0}{4\pi} I \frac{t_y}{t \cdot t} \left(\frac{h \cdot f}{|h|} - \frac{g \cdot f}{|g|} \right)$$

This calculation is done for all wires for each field point, using differentiation arithmetic. So in addition to the value of B_y at each point we have the partial derivative with respect to x and z for some order, say 17.

3.2 Solving for $A_n^{(k)}$

Evaluation of the function B_y at a point p where $x=0$, $y=0$, for specific z using differentiation arithmetic and the Biot-Savart law, gives us the Taylor coefficients.

$$b_{n,m} = \frac{1}{n!m!} \frac{\partial^{n+m}}{\partial x^n \partial z^m} B_y|_{y=0} \quad \text{for all } n+m = \text{order of differentiation}$$

Then the desired functions A_n and their derivatives $A_n^{(k)}$ are obtained by equating corresponding terms of this power series representation of B_y with another representation obtained by expressing the field components as $\vec{B} = -\nabla V$ where V is a scalar potential function for which $\nabla^2 V = 0$. From this is obtained[Caspi91]

$$B_{y|y=0} = \sum_{n=n_0} \sum_{k=0} c_{n,k} A_n^{(2k)}(z) r^{2k+n-1}$$

$$\text{where } c_{n,k} = \frac{(-1)^{k+1} n}{2^{2k} k! (n+k)! / n!} \quad \text{and, for a quadrupole magnet, } n_0 = 2.$$

4 Analysis of Computation

Experience with conversion software had convinced us we would not get good results without actually understanding the computation. It was essential to know what was in all those common blocks to determine which data should be distributed to the processor nodes. We did the same computation as the original code except we did not implement one time-saving feature which used fewer wires in computing the small field contribution from cable blocks far from the field point. Also in doing the differentiation arithmetic we did not try to exploit sparseness by using a zero tolerance and eliminating computations involving computed zeros as Bertz did. We did use the order of the operands to avoid unnecessary computation. The following steps describe the calculation done.

Read cable blocks.

The typical input has about five thousand blocks or sections of the cable windings. These blocks are specified by giving the 3D coordinates of the faces. For some magnets, say a quadrupole, only one octant of the windings are input. The others are generated by symmetry when needed. In the case of the automatic differentiation calculation, only one octant, plus computer generated-wires needed to complete the circuit, is used. Then knowledge about the symmetry and the harmonics of the particular magnet type (dipole, quadrupole, etc.) is used to compute the A_n function values and derivatives. Other magnets, a wiggler for example, cannot take advantage of symmetry.

Generate wires.

The number and location of the wires generated depends upon the cable properties. Typically, each small chunk of cable might be represented by 24 wires, giving us several hundred thousand wire segments in an octant of windings. Wire data generated are the coordinates of the wire endpoints.

Distribute wires to processors.

On the MasPar and CM-5 the wire data are divided between the processors. The PVM^[PVM] version puts all the wires on each workstation.

Optionally save the wire data for future runs.

Although the time to read the ASCII file of windings and convert to wires is an insignificant part of the total computation time, waiting for it may take several minutes. For the MasPar this data is saved on the disk array the first time a new magnet is used. Subsequent runs do a parallel read directly into the processor elements. This makes the input and distribute step very fast. Even without a disk array, it is efficient to save the wires as a binary file.

Read locations where field functions are required.

The number of points at which values for the A_n 's and derivatives are needed to specify the field in a region of interest are typically a few hundred. Values at other points are obtained by interpolation.

Do the Biot-Savart calculation.

This is the calculation done in parallel. Although there is possible parallelism in the automatic differentiation within just one wire-point calculation, there was so much parallelism in the data that it was not necessary to exploit any other parallelism. Hundreds of thousands of wires contribute to the field at a given point.

Solve for the A_n functions & derivatives.

This is done by equating the coefficients of two power series representations of the field. This was not parallelized for the MasPar. It uses little time since the order of the system of equations does not exceed the order of the derivatives computed.

Output the A_n 's and their derivatives.

This is done by the front-end for the MasPar and by the master in the PVM version.

5 MasPar Data Parallel Implementation

The MasPar MP-2 computer with 4096 processors was acquired by LBL knowing that it would do well with image processing. Because it was the only parallel machine in the laboratory, we were interested in knowing how well it would do other scientific applications. We will describe a few difficulties that were encountered in most of the f77 codes offered as candidates to run on the MasPar.

Each processor element (PE) on the MP-2 has only 64K Bytes of memory. LBL computer users are accustomed to virtual memory, and are sometimes very extravagant in memory use. Since there is a tendency to use double precision everywhere, rather than determining where it is needed, the small amount of memory prevented many of the parallelizations attempted. Incidentally, double precision computations on the MasPar take almost three times longer than single precision, a greater factor than on most other machines.

One of the most common difficulties is the extensive use of the Fortran common statement. The organization of data in some applications is incomprehensible, and is explained as "historic." Sometimes dynamic memory management is done by the users, reallocating space within common. Many physics codes use the CERN library and one of the more popular utilities does memory management. Sometimes parallelization was abandoned solely because of common blocks.

While conversion tools are provided, and they do reorganize common blocks to satisfy the restrictions placed on them in MPFortran, automatic conversion of codes was, in general, not satisfactory. The converted program required too much space, made inefficient use of the processors, and did lots of "sloshing," the MasPar term for moving data between the front end and the PEs.

A minimal-effort conversion began with profiling the existing code. If the profile showed a large percentage of time being spent in a small part of the code, there was a good chance of a successful conversion. Most of the code was left untouched and ran on the front-end. The part that was converted to data-parallel[DataP] was small and, if you were lucky, involved a standard computation, like

the solutions of systems of equations or Fourier analysis, for which code was available in the MasPar Mathematics Library.^[Maspb]

The magnetic field calculation code spent almost all of its time in subroutines called to do automatic differentiation. Should we look for another differentiation software package? What would run on the MP-2? Or perhaps the calculation just was inherently time consuming. It was decided to write a new program. Although not ready to invest resources in a new general-purpose software package, we did want to evaluate automatic differentiation for this particular problem.

If MasPar had provided overloading functionality in either MPL, their variant of the C language with parallel extensions, or in MPFortran, their variant of Fortran90, we would have used that method. Since this was likely to be available soon, we went the elemental approach, making our own subroutine calls to do elementary operations. The code was rewritten in MPFortran, using the forward method of automatic differentiation.

Getting efficient code was not hard. Using the tools provided for analysis of the parallel code allowed us to find any statement which had generated inappropriate router communication or data sloshing. The cause was usually obvious and easy to correct.

As frequently happens, writing new Fortran90 code requires some analysis of the computation which means improvements are made which really have nothing to do with parallelization. When the program done for the MasPar was converted to f77, it ran almost nine times faster than the original code.

A coding decision, insignificant to MasPar performance, was made which severely affected performance when the code was later run on the CM-5. The LBL MasPar had 4096 processors: we could hope for more, but would never have as many processors as we had wire-point calculations. We could just let the compiler take care of distributing the job across the processors as many times as required. But we chose to specify the actual number of processors and then do the required number of wire-point calculations on each processor, summing locally before summing once across the processors for each field point. This worked efficiently on the MasPar. The job runs at 268 Mflops; the machine is rated at 600 Mflops for double precision operations.

6 CM-5

We used the CM-5 at UC Berkeley consisting of 32 SPARC2 processor nodes, each with high-performance arithmetic hardware called vector units. The CMFortran compiler can be directed to compile code to use or not use the vector units.

The effort to compile the code written for the MasPar with CMFortran was minimal since both languages are Fortran90 variants. The memory mapping directives, the timers, and file specification parameters were changed

The first performance results were disappointing. Using the vector units we got 9 Mflops while the machine is rated at 4096 Mflops. Without the vector units we got 2 Mflops from a possible rated 134 Mflops. Also the timers reported that the busy time was frequently less than half of the elapsed time.

The Timers used are SIMD type timers which refer to the processor nodes as a collective unit giving you a busy time and an elapsed time. Busy time is the time during which the user program is executing user code. Elapsed time is the sum of busy time and idle time when the user's program is looping in the operating systems dispatch loop. Since system load under timesharing can affect the program timings, and exclusive use of the partition is usually not possible, some comparisons were made, timing with various loads.

It seems that the load effect is usually not very great for busy time, but it makes a significant difference in idle time. We checked the reliability of the timers by timing other examples, timing nothing, and timing the timers. We made multiple runs, trying to make some when the machine was lightly loaded. We concluded that probably the idle time for this application was less than 10 percent. For consistency, we used busy time in our timing comparisons.

The real problem for CMFortran was our having told the compiler to do N wire-point calculations by using the actual number of processors, 128 vector units for our CM-5, and doing $N/128$ calculations on each processor. This worked well for the MasPar but was a disaster here. This compiler needed to be told it had N virtual processors to do the N calculations. This could be easily done just by changing a parameter in the code.

To verify this effect we used a tiny code computing dot products rather than wire-points. The same number of dot products were done in each case. The 32 nodes were used for each calculation. The only variable was the number of virtual processors specified to the compiler which is the matrix dimension spread in the parallel (NEWS) direction. Since each of the 32 processors had 4 vector units, the smallest number of processors specified was 128.

This is the data layout, with nproc, the number of virtual processors, varied.

```
parameter (nproc=4096)
parameter (lvec=64)
parameter (nvectot=65536)
parameter (nvec=nvectot/nproc)
CMF$ LAYOUT a(:SERIAL,:SERIAL,:NEWS)
CMF$ b(:SERIAL,:SERIAL,:NEWS)
CMF$ LAYOUT c(:SERIAL,:NEWS)
```

This loop was timed.

```
do j=1,nvec
  do i=1,lvec
    c(j,:)=c(j,:) + a(i,j,:)*b(i,j,:)
  enddo
enddo
```

The simple dot loop compiled with CMFortran without the vector units got 5.68 Mflops, using the vector units got 542 Mflops. Replacing the Fortran loops with one call to the routine from the hand-optimized CMSS library for multiple dot products, GEN_INNER_PRODUCT, our best run got 1342 Mflops, a respectable but not spectacular fraction of the rated 4096 Mflops for 32 processors with 4 vector units. Unfortunately, since we just used the dot as a sample calculation, all this suggests for the wire-point calculation is that it shouldn't have been done in CMFortran.

The best performance on the vector units required that the number of virtual processors equal the number of calculations. Also the CMFortran compilation should be done with -noaxisreorder (since we had ordered our axes the way we wanted them) and without the -O flag when compiling for vector units, but with -O for SPARC. This was true for the dot calculation and for the wire-point calculation.

Using CMFortran to spread our data-parallel wire point calculation across the processors, the best time was 132 Mflops with the vector units and 2.3 Mflops without vector units.

Host/node code which didn't use the vector units could be produced using CMMD[CMMD] for communication and compiling the code with Fortran77 to give a performance of 32 Mflops, 14 times better than the CMFortran code performance. The performance of host/node code using CMMD and CMFORTRAN to compile node code for the vector units was about the same as the the data-parallel version for vector units, 127 Mflops vs 132 Mflops. These performance measurements are shown in the following graph.

Performance

7 PVM

At LBL, as at any large research facility, there are many workstations with unused compute cycles. There is a continuing interest in using these free cycles. To distribute this application across workstations, PVM was used. PVM (Parallel Virtual Machine) is a software system that enables a collection of heterogeneous computers interconnected by a network to function as a loosely coupled concurrent supercomputer. PVM^[PVM] is readily available without cost, an important portability consideration. Although other software, including Express^[ParaS], was available at LBL, it might be unavailable at other sites. PVM is easy to install and use, is supported, and has a large user's group.

In addition to the unused cycles of colleagues' machines, the central computing facility at LBL provides a selection of other machines, including ten SPARC 10/41 workstations on their own CDDI subnetwork intended for distributed computing. Scratch disk space is available on the local server to avoid heavy traffic outside the subnet.

The most important performance consideration for distributed computing in this environment is the fact that the network of machines is being used in a multiuser, multitasking environment. There is no use getting caught up in computing task granularity when you have no control over machine load or network traffic, whatever the rated performance of the various machines or the network bandwidth.

At LBL there have been some acrimonious mail messages between users sharing the ten SPARC facility. One Express user, who divides his tasks equally between the processors, was enraged when another scientist totally saturated one of the processors. The Express job was barely running, although there was still lots of compute power available within the subnet. The response to his complaint was easy: all users have equal access; you have to do better load balancing. Static load balancing does not work very well in a multiuser environment.

When trying to get free cycles from colleagues' machines, a more difficult problem is encountered. Some people try to run only at night so that they do not interfere with the workstation owner. Others monitor the load on the processor, and kill the task if a load

is detected. Sometimes it is hard to do this fast enough. Occasionally the owner will show up in the off hours, say preparing for a presentation the next day. He is tired, desperate, and with a very low tolerance for even a fraction of a second response delay that the distributed job may cause. With colleagues in computing there is no problem--they kill the distributed process. Some of the physicists cannot do this on their own workstation. The general solution to the problem is being worked on by others. If the distributed tasks are killed by someone or a machine goes down, we need to deal with fault tolerance.

The first PVM implementation used the existing code and PVM2. Dealing with the ubiquitous common blocks was the hardest problem in getting it to run. The worst performance problem was caused by faults. If a workstation crashed or a daemon mysteriously disappeared, the whole job crashed. When these disasters did not occur, performance was fine. This application has a lot of computing and little communication.

PVM3 is more robust than PVM2. Now if a workstation dies or never returns any messages, the whole job does not crash. However, it is up to the application to see that whatever work was expected from the defunct machine is done somewhere else. The PVM3 implementation used the new optimized serial code for the Biot-Savart calculation, thus it was fast as well as stable.

Fortunately with this application it is easy to handle load balancing as well as fault tolerance. The job runs in master/slave mode with the master just doing I/O and distributing data. The computation differs from the data-parallel version only in that each workstation is assumed to have enough memory for all the wire data. Therefore, a complete field point calculation is done by one processor. This was a reasonable assumption, since a typical workstation had enough memory, and made keeping track of the completed work easier.

After broadcasting wire data to the slaves, the master sends points to the slaves, who return the field and derivatives at a point. The master begins by sending a set of points to each slave, so the slave won't be waiting for additional points. The slave returns the field results one point at a time. When the master sees that a slave is working on its last point, the master sends another set of points. The size of the set of points sent decreases as termination nears, with single points being sent near the end. The master checks the

points that have not been returned and reassigns the one that has been out longest to the next available processor. This continues until all have been returned. This simple pool-of-tasks scheme solves the dynamic load balancing problem and takes care of fault tolerance.

As the calculation concludes, only a few nodes are busy. For a small problem this significantly affects the computed performance. The calculation for the last field points could be subdivided among processors to reduce this effect, but for real problems it is unimportant. The job is usually run using a heterogeneous mix of workstations that are being shared. However, if we have exclusive use of the SPARC10 subnet we see a near linear speedup; we do lots of computing and very little communicating.

Conclusion

We have parallelized a magnetic field simulation based on automatic differentiation using two different kinds of parallelism: data parallelism and process parallelism. We chose to use data parallel Fortran90 variants on the MasPar and the CM-5. This application is well suited to a data parallel model and the MasPar version of Fortran90 gave good results, but performance was disappointing on the CM-5. The test problem originally took 6 days on a SPARC10. The new serial code is 8.7 times faster. Distributed using PVM, it takes full advantage of whatever computing resources are available on the network, achieving near linear speedups. The MasPar code runs 620 times faster than the original code, the CM-5 is 305 times faster. This application now runs fast enough to be practical, demonstrating that automatic differentiation is a good technique and enabling researchers to run larger problems.

[MasPa] MasPar Architecture Specification, Document Part Number 9300-5001, MasPar Computer Corporation, 749 North Mary Avenue, Sunnyvale, California 94086

[Iri91] Masao Iri, *History of Automatic Differentiation and Rounding Error Estimates*, Automatic Differentiation of Algorithms, SIAM 1991

[Grie91] Andreas Griewank and George F. Corliss, preface, *Automatic Differentiation of Algorithms*, proceedings of SIAM Workshop in Automatic Differentiation 1991, SIAM 1991

-
- [Jued91] D. Juedes, *A Taxonomy of Automatic Differentiation Tools*, in Automatic Differentiation of Algorithms: Theory, Implementation, and Application, A.Griewank and G.F.Corliss,eds., SIAM, Philadelphia, PA, 1991
- [Berz89] *Differential Algebraic Description of Beam Dynamics to Very High Orders*, Particle Accelerators, 1989, vol.24 pp109-124
- [Berz90] M. Berz, *Differential Algebra Precompiler Version 3--reference manual*, Technical Report MSUCL-771, National Superconducting Cyclotron Laboratory, East Lansing, MI 48824, 1990
- [Rall91] Louis B. Rall, *Point and Interval Differentiation Arithmetics*, Automatic Differentiation of Algorithms, SIAM 1991
- [Dix87a] L.C.W. Dixon, *Automatic differentiation and parallel processing in optimisation*, Technical Report No. 180, The Numerical Optimisation Center, Hatfield Polytechnic, Hatfield, UK, 1987
- [Bisc91a] C.Bischof, *Issues in parallel automatic differentiation*, in Automatic Differentiation of Algorithms, A. Griewank and G.F. Corliss, eds., SIAM, Philadelphia, PA, 1991
- [Bisc91b] C. Bischof, A. Griewank, and D. Juedes, *Exploiting parallelism in automatic differentiation*, in Proceedings of the 1991 International Conference on Supercomputing, E. Houstis and Y. Muraoka, eds., Baltimore, MD., 1991, ACM-Press, pp. 146-153.
- [Casp92] S. Caspi, M.Helm, L.J. Laslett, and V.O. Brady, *An Approach to 3D Magnetic Field Calculation*, LBL-32624, Lawrence Berkeley Laboratory, U. of California, Berkeley, CA 94720
- [PVM] PVM 3 User's Guide and Reference Manual, ORNL/TM-1287, Oak Ridge National Laboratory, Oak Ridge, , Tennessee 37831
- [DataP] Data -Parallel Programming Guide, MasPar Computer Corporation, 749 North Mary Avenue, Sunnyvale, California 94086
- [Maspb] MasPar Mathematics Library, Document Part Number 9302-5001, MasPar Computer Corporation, 749 North Mary Avenue, Sunnyvale, California 94086
- [CMMD] CMMD Reference Manual, Thinking Machines Corporation, 245 First Street,Cambridge,Massachusetts 02142-1264
- [ParaS] ParaSoft Corporation, 2500 E. Foothill Blvd, Pasadena, CA 91107