1997

NASA/ASEE SUMMER FACULTY FELLOWSHIP PROGRAM

MARSHALL SPACE FLIGHT CENTER
THE UNIVERSITY OF ALABAMA IN HUNTSVILLE

PERFORMANCE OF A BOUNCE-AVERAGED GLOBAL MODEL OF SUPER-THERMAL ELECTRON TRANSPORT IN THE EARTH'S MAGNETIC FIELD

Prepared by: Tim McGuire, Ph.D.

Academic Rank: Assistant Professor

Institution and Department: West Texas A&M University
Department of Computer Information Systems

NASA/MSFC:

Office: Space Sciences Laboratory
Division: Science Systems

MSFC Colleague: Mike Liemohn, Ph.D.

Introduction

In this paper, we report the results of our recent research on the application of a multiprocessor Cray T916 supercomputer in modeling super-thermal electron transport in the earth's magnetic field. In general, this mathematical model requires numerical solution of a system of partial differential equations.

The code we use for this model is moderately vectorized. By using Amdahl's Law for vector processors (Fosdick, 1996), it can be verified that the code is about 60% vectorized on a Cray computer. Speedup factors on the order of 2.5 were obtained compared to the unvectorized code. In the following sections, we discuss the methodology of improving the code.

In addition to our goal of optimizing the code for solution on the Cray computer, we had the goal of *scalability* in mind. Scalability combines the concepts of *portabilty* with *near-linear speedup* (Sabot, 1995). Specifically, a scalable program is one whose performance is portable across many different architectures with differing numbers of processors for many different problem sizes. Though we have access to a Cray at this time, the goal was to also have code which would run well on a variety of architectures.

Additional Vectorization of the Code

The first step attempted was to examine the code carefully to see if the degree of vectorization could be increased. Typical consideration for improving vectorized codes include:

- Most vector-register architectures are designed to handle vectors of a given length; for instance, the vector registers of the Cray T90 series hold 128 elements of a vector. Using vectors of that approach that length is more efficient than using shorter vectors. In a nested loop, the inner loop is usually the one the compiler attempts to vectorize. Thus, if it is computationally equivalent to do so, the code should be arranged so that the longer vector operation occurs in the innermost loop.
- When dealing with multi-dimensioned arrays, the storage order can have an impact on the efficiency. Since FORTRAN stores arrays in column-major order, it is most efficient to access those arrays in that order. This allows us to process the array with a stride of one. It also reduces the memory and caching overhead, since the elements are being read and stored in a sequential manner.
- Compiler dependence analysis is of necessity conservative. Thus, the compiler might not detect all instances where an apparent dependence can be ignored. For example,

```
do i=2,n
   a(i-1) = a(i) + b(i)
end do
```

  can be vectorized and still generate the correct value. A more subtle instance is

```
do i=1,n
   a(i+1,j) = a(i,k) + b(i)
end do
```

which can be vectorized if is known that $j \neq k$. In most cases, however, the compiler is unable to make such an assumption and does not vectorize the loop. In these cases, most compilers have a "no dependence" directive that allows the code to be vectorized.

In practice, it was discovered that the Cray `f90` compiler performed quite well at exploiting the vectorization and as a result, the amount of performance increase on this code by hand vectorizing was not large ($\approx$ 5%.) When possible, loops containing recurrences (which are inherently scalar) were split so the portions which could be vectorized were separate.

Parallelization of the Code

Cray FORTRAN90 supports various levels of parallel processing through *autotasking*. In addition, it is possible to use *macrotasking*, since Cray provides a library of synchronization routines. This approach requires a major restructuring of the code to take advantage of parallelization. It has value when there is a large granularity to the code, but has considerable overhead cost. Because the resulting code is not portable, this approach was not used.

Autotasking directs the compiler to exploit parallelism in the code, typically by distributing loop iterations to multiple processors. The user may, if desired, insert directives in the form of comment lines (leading to transportable code) in some or all of the routines to enhance parallelization in the regions of the code where the preprocessor cannot determine, that parallelization is safe.

Utilizing the fully automatic method of parallelization only yielded a minimal improvement in the performance of the code ($\approx$ 5%.) This led us to pursue the insertion of directives into the code in an attempt to increase both the granularity and the average degree of multiprocessing.

Computational Results

The supercomputer used in this study was a Cray T916, a shared memory multiprocessor vector machine with up to 16 processors and a vector length of 128 64-bit words. The particular machine used was a 4-processor model with 256 megawords of main memory. Each processor has a peak performance of approximately 1.8 billion floating point operations per second (1.8 Gflops.) The operation system utilized was UNICOS 9.0.2.4 and the FORTRAN compiler `f90` version 3.0.0.

The problem considered for this study was a collisional three-dimensional interhemispheric flux tube model for photoelectrons (PE) (Khazanov *et al*, 1996). Using this model, initial calculations of the high energy PE distribution as a function of time, energy, pitch angle, and spatial location in the equatorial plane and along the field lines, are reported for different conditions of geomagnetic activity. To explore both the dynamic and steady behaviors of the model, the simulation starts with the abrupt onset of PE excitation, and is followed to steady state conditions. The results illustrate several features of the interaction of PE with typical magnetospheric plasmas and fields, including collisional diffusion of PE in pitch angle with flux tube filling, diurnal intensity and pitch angle asymmetries introduced by directional sunlight, and energization of the PE distribution in the evening sector.

This code had previously been run on an HP 9000 workstation with a run time greater than one month. The first port to the Cray ran longer than one week. Exact runtime data is not available, but it is obvious that improving the code is desirable.

For the sake of this project, we chose to use the original input data with a shorter runtime (i.e., 40 time units rather than 172800) so that many more runs could be done. On a non-optimized code, this lead to total wallclock time of 89.6 seconds. Utilizing the compiler's scalar optimization, we achieved a wallclock time of 35.7 seconds. Using the default level of optimization (which includes moderate scalar, vector, and tasking optimization) for the `f90` compiler led to a wallclock time of 17.5 seconds. Using aggressive scalar and vector optimization gave a wallclock time of 15.4 seconds. The addition of aggressive autotasking led to 14.7 seconds of wallclock time.

At this point is was determined that hand optimization would be necessary to achieve substantial performance gains. Hand vectorization of the code was attempted first. This was mainly done by splitting recurrences out of loops in which other operations could be vectorized. This gave a runtime of 14.1 seconds.

We concluded the work with an examination of how we may be able to help the compiler with directives. Aggressive autotasking found 20 loops in the code which could be parallelized. Because several loops contained subroutine calls, these were not parallelized by the compiler. Careful examination of the code revealed that no side effects resulted from many subroutine calls, so a compiler directive was inserted to allow those loops to be parallelized. In addition, it was discovered that on one occasion the autotasking preprocessor was unable to determine that an outer loop was safe to parallelize, and hence it parallelized the inner loops. We were able to reduce the overhead and increase the granularity by inserting compiler directives. By doing this we were able to double the amount of loop parallelized and reduce the wallclock time to 5.3 seconds.

It should be noted that each case was run three times and the median of the wallclock times were taken. This is help eliminate the machine load factor since we did not have a dedicated machine.

Concluding Remarks

The above results are summarized in the following table:

Non-vectorized code	89.6 seconds
Non-vectorized code, scalar optimization	35.7 seconds
Normal code (moderate scalar and vector optimization)	17.5 seconds
Aggressive vectorization	15.4 seconds
Aggressive vectorization and autotasking	14.7 seconds
Manual vectorization and autotasking	14.1 seconds
Manual tasking	5.3 seconds

We have shown that the Cray T916 can achieve substantial performance improvement in a "real-world" problem. The ability of the compiler to exploit vectorization is impressive. Code that is not designed specifically with parallelization in mind may be restructured to take advantage of parallel processing. Parallelization by hand is necessary in these cases.

The speed-up factor of 2.8 on a four-processor machine is reasonable for a problem of which is not inherently parallel. It is reasonable to expect that correspondingly faster performance would arise using additional processors, up to the number of grid points.

References

L. D. Fosdick, E. R. Jessup, C. J. C. Schauble, and G. Domik (1996), *An Introduction to High-Performance Scientific Computing*, Cambridge, MA: MIT Press.

G. V. Khazanov, T. E. Moore, M. W. Liemohn, V. K. Jordanova, M. C. Fok (1996), "Global, collisional model of high-energy photoelectrons," *Geophysical Research Letters* 23:331-334.

G. W. Sabot (1995), *High Performance Computing: Problem Solving with Parallel and Vector Architectures*, Reading, MA: Addison-Wesley.