Porting LCPFCT to the CRAY T3E Using HPF

Jay Boisseau, Bob Sinkovits, and Ken Steube
San Diego Supercomputer Center
San Diego, CA USA

ABSTRACT: *LCPFCT is an accurate, robust, easy-to-use, publicly-available library of routines for solving nonlinear, time-dependent continuity equations using the Flux-Corrected Transport (FCT) algorithm. LCPFCT has been used in a variety of applications, including fluid dynamics, plasma dynamics, magnetohydrodynamics, and reactive flows. LCPFCT has been optimized for many platforms, including CRAY PVPs. We present the first version of LCPFCT converted to HPF (LCPFCT-HPF) and optimized for the CRAY T3E. We discuss the issues involved in porting and optimizing LCPFCT and present performance data for up to 32 processors on the T3E.*

Introduction

In order to better understand the process of taking an application from a CRAY PVP system and porting it to the T3E, we have parallelized a fluid dynamics code using High Performance Fortran (HPF). Several steps were involved:

- choose a code that seems well-suited for HPF on an MPP system
- choose a porting strategy
- port the code
- evaluate performance
- enhance performance

While this work evaluates only one of the available HPF compilers, it will help scientists and consultants set their expectations for such an endeavor. The strengths and weaknesses of the different HPF implementations will of course vary.

Our ease of success at parallelizing a code using the Portland Group's HPF compiler speaks well for this compiler implementation. Certainly some effort was required to find the best method of porting and to avoid some problems with the compiler, but considered as a whole, our experience with this compiler on this code is positive.

Choosing a Code for Porting to HPF

As has been noted elsewhere, a class of codes that works reasonably well in current implementations of HPF compilers includes many finite difference and image processing codes. These types of problems require the exchange of data on the boundaries between blocks owned by different processors and can be parallelized using overlap shift operations. For example, in a problem with a 2-D array that is distributed in vertical strips, it might be necessary to have processors exchange one or more boundary columns, depending on the type of differencing or image processing technique used. HPF compilers are optimized to recognize the need for this type of communication pattern.

Some of the current HPF compilers, such as Portland Group's, provide an option that allows the programmer to specify the amount of overlap to be exchanged. This is an aide to the compiler in cases where it may not be able to determine the optimal overlap. Continuing the 2-D example, an algorithm that involves updating U(I,J) based on the values of four near neighboring values would require an overlap of 1. The programmer may be able to assist the compiler by specifying this value.

HPF is not currently designed to easily accommodate irregular grid problems, although enhancements are under consideration.

The Code: LCPFCT

LCPFCT solves the nonlinear, time-dependent continuity equations using the Flux-Corrected Transport (FCT) algorithm. It offers 4th order phase accuracy with minimal residual diffusion, and is monotone, conservative, and positivity-preserving. FCT was developed to solve problems which are characterized by very steep spatial gradients. When applied to the numerical simulation of shock wave propagation, FCT preserves the sharp density and pressure profiles while maintaining the positivity of physical variables and without introducing new maxima or minima into the solution.

Algorithm

LCPFCT can be used for both 1-D and multidimensional problems. In the later case, a technique known as direction splitting is used where LCPFCT is called repeatedly over each of the spatial dimensions. Although fully 2-D and 3-D versions of LCPFCT have recently been developed, many codes still use the 1-D version.

Code Characteristics

Our first task was to run the code on a single processor of the T3E using **apprentice** to determine which loops were important to parallelize. There were five loops in the LCPFCT subroutine which consumed nearly all the compute time, and so we focused our efforts on those loops.

Viability for Porting

LCPFCT is a fairly strong candidate for parallelization even though it solves only a 1-D problem. In many instances, such as the numerical simulation of reactive flow problems (i.e. flames, detonations, etc.), convection is only one of several physical processes (conduction, diffusion, chemistry) that must be treated. It is important that the convection routine be well-parallelized so that it does not become the bottleneck in the simulation.

In a typical finite difference calculation, element `X(I)` of an array is updated based on the values of neighboring elements in index space. This leads to a communication pattern in which a single floating-point value needs to be sent to the neighboring processor for each distributed array. A sample loop from the LCPFCT subroutine is shown below:

```
      DO I = 1, N
         FLXH(I+1) = FSGN(I+1) *
     &      AMAX1(0.0, AMIN1(TERM(I+1),FABS(I+1),TERP(I+1)))
         RHON(I) = RLN(I) * (LNRHOT(I) + (FLXH(I)-FLXH(I+1)))
         SOURCE(I) = 0.0
      END DO
```

Three criteria are commonly applied to a program to determine if it is a candidate for parallelization. **Frequency of use**: LCPFCT is the basic program used by an entire group at the Naval Research Laboratory, in addition to a large number of researchers around the world. **Execution time**: runs last from several minutes to several hours on the CRAY PVPs. **Resolution needs**: LCPFCT is commonly used with high temporal resolution, and occasionally used for high spatial resolution. It is these runs with high spatial resolution that our work targets. High temporal resolution does not indicate a need for parallelization in the case of this program because the time steps must be done sequentially.

Additionally, vector-style calculations on stride-one loops means that we should be able to get reasonable performance on this code. However, one normally has to resort to extremes to get good performance from the hierarchical memory systems on modern microprocessors, and we will chronicle our difficulties in this part of the endeavor below.

Choosing a Porting Strategy

HPF Overview

HPF is a data parallel language intended to simplify the process of porting serial Fortran programs to parallel computers. In theory, you can take an existing serial program and add compiler directives to distribute large arrays among the processors on your system. Special attention must be given to the loops in the program to specify how they should be parallelized. The programmer can apply compiler directives to parallelize a loop or rewrite **DO** loops using data parallel constructs such as the **FORALL** statement. In addition, many loop operations can be replaced by intrinsic functions which are generally well-optimized for the target architecture.

PGHPF Overview

The Portland Group's **pghpf** compiler was used to port the code. This compiler is currently capable of compiling most Fortran 90 and HPF syntax, and a review of the release notes turned up no serious problems for our effort.

pghpf compiles HPF programs by translating them to Fortran 77 syntax, and then by calling the native Fortran 77 compiler (**f90** on the T3E) and linking with HPF libraries. Communication is done via HPF library calls, which ultimately use CRAY's fast SHMEM calls.

Initial Plan

Since LCPFCT is a structured grid computation, we were able to decompose arrays using the **DISTRIBUTE** and **ALIGN** directives**.** All computation is done in approximately 30 conformable 1-D arrays, which are distributed in a block fashion such that corresponding elements of different arrays all reside on the same processor.

The easiest technique for parallelization in HPF is to apply **INDEPENDENT** directives to loops where there are no data dependencies. Care must be taken in these cases to be certain that no data dependencies actually exist within the loops.

As always in parallel programming, the simplest technique is not necessarily the best. We anticipated using **FORALL** and array syntax to parallelize some or all of the loops to get the level of performance we desired. We drew the line at creating extrinsic functions since our purpose was to study HPF rather than message passing.

Porting Experiences

A number of the loops in the LCPFCT routine did contain data dependencies. These were eliminated by splitting the loops into two or more loops. Our first attempt at parallelization of the **DO** loops was to precede them with the **INDEPENDENT** directive. While this gave correct results, it was immediately obvious that we had achieved no scaling whatsoever. In an attempt to determine the reason for the poor performance we examined the intermediate Fortran 77 source code generated by the compiler. What we discovered was that the compiler had to insert numerous PGI library calls to ensure data coherence in loops that would have required only one or two message passing calls. The **DO INDEPENDENT** loops could be made to work much more efficiently by using temporary arrays to hold shifted copies of arrays that are used within the loops. By doing this the loops could be written so that the same array index is referenced on both side of assignment statements.

The next attempt at loop parallelization involved the use of the **FORALL** construct. This resulted in a significantly faster code which achieved reasonable scaling and avoided the overhead and additional memory requirements associated with copying shifted versions of arrays into temporary arrays. Finally, we tried replacing the **FORALL** constructs by Fortran 90 array syntax statements. The timings for these two versions of the code were virtually identical as we had expected.

Evaluating Performance

Timings

We tabulated timings and performance data for runs with our 1-D arrays of lengths 10000 and 100000 on various numbers of T3E processors and compared these to the times on a single vector processor of the CRAY C90:

PEs	N=10000		N=100000	
	CPU(s)	Mflops	CPU(s)	MFlops
1	1.1790	16.5	10.6875	18.5
2	0.7502	26.0	5.5221	35.9
4	0.3768	51.8	2.8627	69.2
8	0.2515	77.6	1.5354	129.1
16	0.1973	98.9	0.8694	228.0
32	0.2046	95.4	0.5663	350.1
C90	**0.0507**	**385.0**	**0.5071**	**391.0**

The CPU time decreased as the number of processors was increased to 16 or 32, respectively, for problems of size N=10000 and N=100000. Since the Portland Group's profiling facility **pgprof** is not yet supported for the

CRAY MPP systems, performance data was determined by comparison with CPU times obtained from runs on the CRAY C90 using the **jumpview** utility. For both problem sizes, a single processor of the C90 was faster than 32 PEs of the T3E. This result is not too surprising though since the ratio of computations to communications tends to be relatively low for reasonably sized 1-D problems. In addition, the LCPFCT code had already been carefully optimized for vector machines, achieving almost 40% of peak performance on the C90 and utilizing average vector lengths very close to 128.

Enhancing Performance

The types of performance enhancements we were able to do in this work were severely limited due to the constraints placed upon us by HPF. The main enhancement involved using **FORALL** constructs or array syntax operations. After this our code used optimized library calls provided by the Portland Group to perform much of the calculation. Of course, this gave us no degree of control over the use of the DEC Alpha 21164 cache memory.

The way to optimize for cache memory use is to control data layout so that large chunks of each of the arrays can co-exist in cache. Normally, this would be done by re-organizing the common blocks in which arrays are stored, and by inserting pad variables to space the arrays. This way, one can control the mapping of the arrays into cache memory. Since arrays in Fortran 77 common blocks are guaranteed to be stored contiguously in memory, you can control where they map into cache relative to each other by changing the sizes of the pad variables.

In HPF, though, a guarantee of contiguousness inhibits automatic distribution of the arrays, and so no such guarantee is available. You simply cannot get contiguous storage of arrays while distributing those arrays. A future release of CRAY's **f90** compiler will have compiler options to assist with cache storage control, and it may be possible to better optimize for cache at that time.

Another issue in optimizing this code is that since there are so many work arrays requiring the same communication pattern, we would like to combine the communication calls into a single call to reduce latency. This is a global optimization that is easy for the programmer to detect given his knowledge of the calculation. We were unfortunately subject to the limitations of the compiler and its ability to do this type of global optimization.

Summary

The overhead generated by the implicit synchronization and communication calls between dependent loops and array syntax operations makes it necessary to construct loops and array operations with *many* operations per processor to overcome the latency and bandwidth limitations of the hardware. This constraint is much more severe than on CRAY PVP systems, which have vector processing hardware (much less overhead to start loops) and even compilers that automatically perform loop fusing where possible. Thus, efficient vector code does not necessarily produce good parallel code just by using **DO INDEPENDENT** assertions, converting to FORALL statements, or rewriting using array syntax operations. Porting Fortran codes from CRAY PVPs to a CRAY T3E requires an extremely careful analysis of the main loops for sufficient computational content between the loops (that probably must be reorganized to remove parallel dependencies), where implicit overhead occurs. Our work is shows that the current PGI HPF compiler can produce scalable code, but demonstrates the difficulties one might have in developing efficient parallel code using an HPF compiler at this level of maturity.