Challenges in the Automatic Parallelization of Large-scale Computational Applications*†

Brian Armstrong

Rudolf Eigenmann

School of Electrical and Computer Engineering
Purdue University, West Lafayette, IN 47907-1285

ABSTRACT

Application test suites used in the development of parallelizing compilers typically include single-file programs and algorithm kernels. The challenges posed by full-scale commercial applications are rarely addressed. It is often assumed that automatic parallelization is not feasible in the presence of large, realistic programs. In this paper, we reveal some of the hurdles that must be crossed in order to enable these compilers to apply parallelization techniques to large-scale codes. We use a benchmark suite that has been specifically designed to exhibit the computing needs found in industry. The benchmarks are provided by the High Performance Group of the Standard Performance Evaluation Corporation (SPEC). They consist of a seismic processing application and a quantum level molecular simulation. Both applications exist in a serial and a parallel variant. The parallel variants are hand-parallelized with shared-memory directives either at the largest level of granularity or in a hybrid manner where MPI is used at the largest level of granularity and OpenMP directives are used at a lower level. In our studies we compare the parallel variants with the automatically parallelized, serial codes. We use the Polaris parallelizing compiler, which takes Fortran codes and inserts OpenMP directives around loops determined to be dependence-free. Polaris also reports the reasons why it assumes that a loop is parallel. We have found five challenges faced by an automatic parallelizing compiler when dealing with full applications: modularity, legacy optimizations, symbolic analysis, array reshaping, and issues arising from input/output operations. The results of this work will be used to equip parallelizing compilers with the necessary capabilities for handling commercially relevant science and engineering applications.

Keywords: Large-scale Computational Applications, Automatic Parallelization, Compiler Techniques

1. INTRODUCTION

Any programming language, compiler, operating system, and computer architecture will ultimately have to show that it can improve functionality and performance of applications that have commercial value. Such applications are typically large in terms of programming lines and data sets, are widely-used, and are usually not freely available. Most programs that are being used to drive and evaluate the design of new computer systems technology do not fit this definition of commercial applications. Systems research typically uses benchmarks that have reasonably short execution times and that are publicly available. Short runtimes are important because it is not unusual that in the course of a research project a test program is run 100 times. If architecture simulators are used, these programs run two to three orders of magnitude slower than on an ordinary computer. Public availability of test programs is essential for all scientific research, because research results are of small value if they cannot be reproduced by other research groups.

The long-term goal of the research project described in this paper is to advance automatic parallelization technology for high-performance computers. Test applications for such research typically includes suites such as the

*This work was supported in part by NSF grant #9703180-CCR. The findings do not necessarily reflect the views of the supporting agencies.

SPEC CPU, Perfect, or Linpack benchmarks. In this paper we study two programs that come close to our definition of commercial applications. We use two applications from the SPEChpc benchmark suite, called SPECseis and SPECchem. Both codes are large-scale computational applications that reflect problems faced in commercial settings. In using some of the largest possible computational applications, but still using codes that are publicly available, we find a tradeoff between the two demands stated initially. That is, we are using applications that are commercially relevant while still obtaining results that can be reproduced and shared publicly.

SPECseis[5] was developed by ARCO beginning in 1993 to gain an accurate measure of the performance of computing systems as it relates to the seismic processing industry for procurement of new computing resources. It consists of a modeling phase which generates synthetic seismic traces for any size of data set, with a flexibility in the geometry of shots and receivers, ground structures, varying lateral velocity, and many other options. A subsequent phase stacks the traces into common midpoint stacks. There are two imaging phases which produce the valuable output seismologists use to locate resources of oil. The first of the two imaging phases is a 3D Fourier method that is very efficient but that does not take into account variations in the velocity profile. Yet, it is widely used and remains the basis of many methods for acoustic imaging. The second imaging technique is a much slower finite-difference method that can handle variations in the lateral velocity. This technique is likely the basis of most seismic processing migration today. The current SPECseis is missing Kirkoff and pre-stack migration techniques.

Our other application package, SPECchem,[4] is used to simulate molecules ab initio, at the quantum level. It is a current research effort under the name of GAMESS at the Gordon Research Group of Iowa State University and is of interest to the pharmaceutical industry. Like SPECseis, SPECchem is often used to exhibit performance of high-performance systems among the computer vendors. Portions of SPECchem codes date back to 1984. It comes with many built-in functionalities, such as various field molecular wave-functions, certain energy corrections for some of the wave-functions, and simulation of several different phenomena. Depending on what wave-functions one chooses, SPECchem has the option to output the energy gradients of these functions, find saddle points of the potential energy, compute the vibrational frequencies and IR intensities, and more.

The contribution of this paper is to show program patterns of commercially relevant HPC applications that pose significant problems for automatic parallelization. Both SPECseis and SPECchem are parallelized using OpenMP. For this study, we commented out the OpenMP directives and used the manual parallelization as our standard for evaluating how well our parallelizing compiler performs. Then, we have analyzed the reasons why a parallelizing compiler could not detect the same level of parallelism. The compiler used is the Polaris translator,[1] one of the most advanced parallelizing compilers to date. In Section 2 we will describe five categories of challenges faced by parallelizing compilers. Section 2.1 describes issues arising from the fact that large applications naturally have a very modular structure. Section 2.2 shows examples of "legacy optimizations" that compilers must recognize. Section 2.3 discusses the need for advanced symbolic analysis. Section 2.4 deals with the issue of array reshaping at subroutine boundaries; and Section 2.5 describes problems in the presence of input/output operations. Section 3 concludes the paper.

2. HURDLES FACED BY AUTOMATIC PARALLELIZING COMPILERS

In today's mid-scale benchmarks of numerical applications, parallelizing compilers are successful in about half of all applications.[1,3] Using compiler tools on large-scale applications we may find that this success rate is significantly less. In the following sections we present code examples that illustrate these challenges and we discuss possible improvements to compiler technology. Generally, we find that once the regular access patterns typical of computational codes can be extracted from a full application suite of codes, automatic parallelizing compilers and parallelization techniques can perform well.

2.1. Modularity

Large-scale applications naturally tend to be structured into many modules. This has several reasons. Modularity is a general software engineering tool. We have also found that large programs tend to be a result of many software engineers adding code modifications over many years. Furthermore, library modules may be included that perform some of the desired functionality. It is no surprise that full applications have deep levels of hierarchy. They include abstractions with interfaces to the different computational routines. Some of the code may no longer reflect modern engineering practices, which is often referred to as legacy code. Modular programs generally raise the compiler issue of interprocedural analysis. In our work, this issue has become crucially important. To complicate this issue, it is

```
DO jproc = jtop, nproc
   ...
   IF (name .EQ. 'DCON') THEN
      IF (x .EQ. 'A') THEN
         CALL DCONA(ldim,maxtrc,otr,nra,ra,nsa,sa,abort,ipr)
      ELSE IF (x .EQ. 'B') THEN
         CALL DCONB(ldim,maxtrc,otr,nra,ra,nsa,sa,abort,ipr)
      ...
   ELSE IF (name .EQ. 'DGEN') THEN
      ...
   ELSE IF (name .EQ. 'DMOC') THEN
      ...
ENDDO
```

Figure 1. Driver Routine, `SEISPROC`, from SPECseis. `SEISPROC` is the driver routine of SPECseis which loops through the list of seismic processes that are applied to each set of seismic traces.

not always known at compile time which of the functions will be called during a specific execution. In addition, a full application package often consists of code written in several different programming languages, posing a significant challenge to the compiler.

2.1.1. Dynamic Application Functionality

Both of our applications include a large body of functionality. Only a small part of the code is used in any specific execution. For example, SPECseis typically runs in four "phases", called *data generation, data stacking, depth migration,* and *time migration*. The specific routines invoked are determined by the input data. Because of this, our compiler could not determine at compile time what routines would be called. It had to conservatively assume that all possible calls can happen and in all orders.

The code example in Figure 1 shows how a driver routine is used to implement this form of dynamic subroutine invocation in SPECseis. The driver invokes the specific routines of the application. The driver also happens to be within the parallel region of SPECseis. Ideally, Polaris would be able to determine this outer level of parallelism. Unfortunately, this is not the case. Seismic traces are read from disk, transformed through a series of seismic processes, and then written to disk. Each high-level routine, such as `DCONA` or `DCONB` above, uses the same list of parameters. The variable `name` is derived from input data. Therefore, the compiler cannot determine which routines are called or in which order they are called. It concludes that there are cross-iteration dependencies within this region. We have found similar code patterns in the SPECchem application.

To overcome the problem of not knowing which routines will be called at compile time and still determining data independence would require program knowledge. With SPECseis, this would mean that the compiler must know that the seismic routines are only applied to the seismic traces in certain orders. The compiler would also have to understand that the data originates from only a few locations in the code. With this knowledge and with extensive expression propagation an automated compiler may be able to find parallelism encompassing a driver routine.

2.1.2. Language Barrier

Another result of code modularity is multi-lingual applications. SPECseis has a Fortran77 main program, which calls a C routine to allocate memory. The C routine, in turn, calls a Fortran77 routine, which performs the main data processing. These routines may then call low-level C routines to perform disk IO or communicate with other processors. The code sections in Figure 2 illustrate these situations.

As new languages become widely used and compilation techniques for higher-level languages of the object-oriented flavor are developed to produce efficient code, we expect to see the instances where the optimizing compiler must cross language barriers within a single application to grow with time. To overcome this hurdle, the compiler must perform interprocedural analysis across languages. In the codes that we examined, each language was used to perform tasks on specific data. For example, the C routines in SPECseis are used to read seismic trace data from disk files or to copy trace data to a message buffer, but they do not access the data stored in the Fortran common blocks. Yet, Polaris currently assumes that either all or none of the global data is modified when it comes across a subroutine or function call to a routine it cannot access (such as a C routine.)

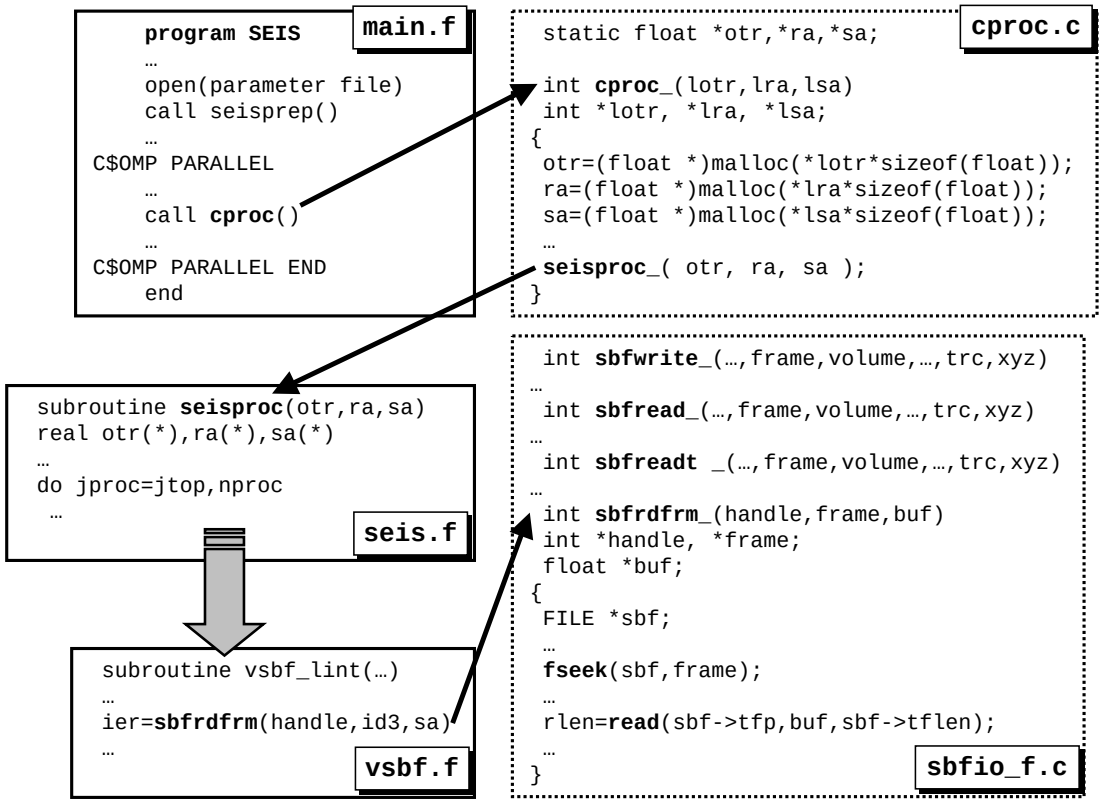

Figure 2. Multi-Lingual Characteristics of SPECseis. SPECseis includes a mix of Fortran77 and C, requiring a compiler to perform global analysis and optimizations across both languages.

2.1.3. Flexible Libraries

Our applications make use of software libraries. One characteristic of these library routines is that they tend to have many options and parameters. Such routines can be called in many different contexts. However, not all of the functionality in the routines is used. Figure 3 shows an example from SPECseis. The library routine, `SCOPY`, is used to copy one vector into another with any stride for either of the two vectors. If a negative stride is given for one of the vectors then that vector is traversed in reverse. However, throughout SPECseis `SCOPY` is consistently called with the strides of both vectors being 1. Similar examples could be given from SPECchem, such as the `DDOT` routine shown later in Figure 4. `DDOT` is almost always called with the strides of the two vectors (`incx` and `incy`) equal to 1.

Our compiler is capable of recognizing parallelism in the presence of these additional parameters. For example, the compiler will pass constant parameters into the subroutine via *constant propagation*. As a result, it can simplify or even remove some of the unnecessary code. In general, enabling such flexible designs of library routines comes with the cost of requiring more advanced compiler capabilities for symbolic analysis, interprocedural propagation, and the recognition of program patterns. Figure 4 (right) also shows a variant of the `DDOT` routine, discussed next.

2.2. Legacy Optimizations

Both SPECseis and SPECchem use legacy code for low-level mathematical functionality. SPECseis includes 35 IEEE library routines to perform Fast Fourier Transformations. SPECchem includes 63 matrix routines, some of which were derived from Linpack code of 1978. These codes tend to be optimized for performance, but may hinder additional compiler optimizations. For example, the `DDOT` routine in SPECchem simply produces the dot product of two vectors. The simple code for the general case is shown on the left side of Figure 4. On the right, a form of `DDOT` is shown that was transformed for improved locality of data references.

```
      SUBROUTINE SCOPY( n, a, inca, b, incb )
C  Copy a vector with stride, BLAS version
      INTEGER n, inca, incb, i, ia, ib
      REAL a(*), b(*)
C  If stride is negative, start from end of vector
      IF (inca .LT. 0) THEN
              ia = 1 + inca*(1-n)
      ELSE
              ia = 1
      ENDIF
      IF (incb .LT. 0) THEN
              ib = 1 + incb*(1-n)
      ELSE
              ib = 1
      ENDIF
C  Loop and copy from a to b
      DO i = 1, n
              b(ib) = a(ia)
              ia = ia + inca
              ib = ib + incb
      ENDDO
      RETURN
      END
```

Figure 3. Library Subroutine `SCOPY` of SPECseis. `SCOPY` comes from the BLAS routines. It copies vector `a` into vector `b` with a possible stride. It even allows the vector to be traversed in reverse with a negative stride. `inca` and `incb` are the strides for vectors `a` and `b` respectively. `n` is the number of elements to copy. Throughout SPECseis, `SCOPY` is called with $inca = incb = 1$.

```
      DOUBLE PRECISION FUNCTION DDOT(n,dx,
     *                        incx,dy,incy)
      DOUBLE PRECISION dx(*),dy(*)
      DDOT = 0.0D+00
      dtemp = 0.0D+00
      ix = 1
      iy = 1
      IF(incx .LT. 0) ix = (-n+1)*incx + 1
      IF(incy .LT. 0) iy = (-n+1)*incy + 1
      DO 10 i = 1,n
        dtemp = dtemp + dx(ix)*dy(iy)
        ix = ix + incx
   10   iy = iy + incy
      DDOT = dtemp
      RETURN
      END
```

```
C USES UNROLLED LOOPS FOR INCREMENTS EQUAL TO ONE.
C JACK DONGARRA, LINPACK, 3/11/78.
   20 m = MOD(n,4)
      IF( m .EQ. 0 ) GO TO 40
      DO 30 i = 1,m
   30   dtemp = dtemp + dx(I)*dy(I)
      IF( n .LT. 4 ) GO TO 60
   40 mp1 = m + 1
      dtloc = 0.0D+00
      DO 50 i = mp1,n,4
   50   dtloc = dtloc + dx(i)*dy(i) + dx(i + 1)*dy(i + 1) +
     *          dx(i + 2)*dy(i + 2) + dx(i + 3)*dy(i + 3)
      dtemp = dtemp + dtloc
   60 DDOT = dtemp
      RETURN
      END
```

Figure 4. Library Subroutine `DDOT` of SPECchem. `DDOT` simply performs the dot product of two vectors, `dx` and `dy`. On the left is the generic form of the code that allows for variable strides through the arrays. The code on the right shows a manually strip-mined version of `DDOT`, optimized for memory locality when both strides (`incx` and `incy`) are 1. The optimized code poses extra complexity for the compiler. Polaris can easily inline the code on the left into parallel loops.

```
DO 310 J = 1,m,mxrows
   jjmax = MIN(m,j+mxrows-1)
   ij = j*(j-1)/2
   ...
   DO 300 jj=j,jjmax
      ...
      DO 200 i = 1,jj
         ij = ij+1
         ...
         h(ij)=hij
```

Figure 5. A Need for Including Intrinsics in the Symbolic Language. To determine some loops to be independent, such as the outer-most loop, the compiler's symbolic analysis must be able to handle intrinsic functions, such as `MIN`.

Our compiler can find that the `DO 50` loop is parallel. However, subroutine `DDOT` is called within a triply-nested loop (not shown here) which is also parallel. The parallelism of the outermost loop can be recognized in the situation of the generic `DDOT` code (the code on the left,) but, the compiler is unable to recognize this fact with the transformed code.

A related, more severe problem is that hand transformations in legacy codes may have been designed for previous generations of high-performance computer systems. For today's machines the transformation may no longer be beneficial, or may even degrade performance. SPECseis includes many lower-level FFT routines that date back to an IEEE Press book of 1979. These routines are optimized to perform Fourier transforms with minimal memory requirements by writing the output to the supplied input array. Such optimizations introduce memory-related dependences, limiting the performance a parallelizing compiler can obtain. Given the application's modest memory requirement and today's machine resources, such optimizations may no longer be adequate.

If the compiler is enabled to recognize specific legacy optimizations then the previous optimizations could be undone and the compiler could perform its own. Another approach would be to empower the compiler with the ability to handle all the functions and complexities added by legacy optimizations. In the above example with `DDOT`, this would mean enabling the compiler to handle the `MOD` function, evaluate the conditional expressions, and merge the `DO 30` and `DO 50` loops into a single, simple loop.

2.3. Symbolic Analysis

At the core of a parallelizing compiler is its capability to detect data accesses that do or do not access the same memory location. This capability involves the analysis of array subscript expressions. Some compilers in current use on high-performance systems can only analyze such expressions if they are *affine*. Affine subscript expressions contain linear combinations of the iteration variables of enclosing loops, where all coefficients are known, compile-time constants. We have found that symbolic and sometimes non-linear analysis is needed in order to deal with the expressions found in realistic codes. If subroutines are inlined, the complexity of these expressions tend to increase further. Also, the *induction variable substitution* technique, which is an important parallelization technique, tends to create non-linear expressions in situations of *triangular loops* or when induction variables are *coupled*. Similar patterns have been discovered in the Perfect Benchmarks.[2]

An example from subroutine `TFTRI` of SPECchem shows the propagated expression used to index an array in loop `DO 200`:

```
x(14+i1+lhc+(jj0**2+(-55)*j1+(-11)*jj0+25*j1**2)/2+5*j1*jj0) = hij0
```

TFTRI deals with a triangular matrix where the subsequent $j(j-1)/2$ elements of the work array are accessed in the next iteration. Variables `j, jj0`, and `i1` are loop indices of a triply nested loop. The Polaris parallelizer could propagate and analyze data dependences in the presence of the above polynomial expressions. Therefore, it was able to find the outermost loop of `TFTRI`, `DO 310`, to be parallel. However, we have found no other compiler with this capability. Our codes emphasize the importance of such symbolic analysis techniques.

The code in Figure 5 shows another example. Polaris was unable to analyze expressions that contain `MIN` or `MAX` intrinsic functions. Originally, the `DO 310` loop of `TFTRI` contained the code in Figure 5. Polaris indicated that a possible cross-iteration dependence for loop `DO 310` may exist in array `h`. Index `ij` iterates from the initial value of

```
      n2 = 1
 10   IF (n2 .GE. n  .OR.  n .EQ. 65536) RETURN
          n2 = 2*n2
          mag = mag+1
          GOTO 10
      END
      ...
      n2 = 2**mag
      ...
        DO 30 i=1,n2
        ...
```

Figure 6. A Need for Recognition of Power-of-2 Loops. The variable n2 is the power of two greater than or equal to n. We can know the value of n2 in terms of n at compile time but Polaris is not capable of recognizing the pattern of the goto loop at line 10.

$j*(j-1)/2$ to $(j+1)*j/2-1$. The `MIN` function was used to ensure that `ij` would not exceed the total number of elements, `m`. If we assume that `mxrows` divides `m` exactly, then we can remove the `MIN` function and Polaris can find `DO 310` to be parallel. `MIN` and `MAX` functions require our symbolic analysis to incorporate inequality relations. Since the size of the data (such as `m`) is not ensured to be a multiple of `mxrows`, we need to include such functions as `MIN`, `MAX`, `MOD`, `FLOOR`, and `CEIL` in our symbolic analysis.

SPECseis poses another challenge to symbolic analysis. Since SPECseis relies heavily on fast Fourier transforms, we find loops that access up to the power of two greater than a dimension of the data. The result of this is that some loops access up to `n2` elements of an array, where `n2` is $2^{\lceil log_2 n \rceil}$ of the data size, `n`; but Polaris gives up with symbolic analysis when dealing with logarithmic and exponential expressions. An example of this from SPECseis is where `seiftm` sets `n2` to be the power of two greater than `n`, and `seicft` includes one of many loops that iterates `n2` iterations, as is shown in the code excerpt in Figure 6. In order for the compiler to know the range of the `DO 30` loop, the compiler would have to understand the goto loop at the top or transform it to use a logarithm function (such as $2^{\lceil log_2 n \rceil}$,) and be able to handle exponentials and logarithm functions in its symbolic analysis.

2.4. Array Reshaping and Type Change

We have described interprocedural analysis to be a very important technique for dealing with modular programs. The Polaris compiler uses subroutine inline expansion to achieve the same effect. A problem that both of these techniques face is that arrays may assume different shapes and have different types in a subroutine and its caller.

2.4.1. Array Reshaping

An example of array reshaping is given in Figure 7. The caller routine shapes the array as a 2D array and the callee shapes it as 1D. Fortran compilers assume that no out-of-bounds indexing occurs by default. With this assumption, `v(1,i)` and `v(1,j)` in the following example will never overlap as long as $i \neq j$. These two portions of the array `v` are passed as two separate vectors into the `DAXPY` subroutine, which sees the two parameters as two single-dimension arrays.

The problem occurs when subroutine `DAXPY` is inlined into `SCHMD`. When Polaris inlines `DAXPY` into `SCHMD`, it linearizes the array index to the 2D array and accesses `v` as a one dimensional array. Then, Polaris cannot determine that the access to `v(1,i)` (which is now `v(i5+(i3-1)*ndim)`) does not overlap with `v(1,j)` (which is now `v(i5+(i3-1+j1)*ndim)`.)

```
      DO i5 = 1, num0, 1
        v(i5+(i3-1+j1)*ndim) = v(i5+(i3-1+j1)*ndim) +v(i5+(i3-1)*ndim)*dum1
      ENDDO
```

Another example of reshaping in Figure 8 shows a situation where portions of a large array, declared in the main program of SPECseis, are passed into several subroutines. Work array, `sa`, is passed into `SEICTRI3D`. If we inline `SEICTRI3D` into the caller routine, then the implicit non-alias assumption of Fortran (the assumption that none of the parameters to a subroutine are aliased) is lost. Only with the non-aliasing assumption of Fortran77 do we know that the accesses to the segments of `sa` do not overlap.

Caller Routine:

```
      SUBROUTINE SCHMD(v,m,n,ldv)
      DIMENSION v(ldv,n)
      ...
      CALL DAXPY(n,dum,v(1,i),1,v(1,j),1)
      ...
```

Callee Routine:

```
      SUBROUTINE DAXPY(n,da,dx,incx,dy,incy)
      DIMENSION dx(*),dy(*)
      ...
      DO 10 i = 1,n
         dy(iy) = dy(iy) + da*dx(ix)
         ix = ix + incx
   10    iy = iy + incy
      ...
```

Figure 7. Example of Array Reshaping. The two columns of the 2D array, `v`, are viewed as two flat arrays within subroutine `DAXPY`. When `DAXPY` is inlined, `v` is accessed as a flat array and Polaris cannot determine that the two accesses to `v` never overlap within the `DO 10` loop.

```
      CALL SEICTRI3D( q0(1,1,k), sa(ka), sa(kb), sa(kaa), sa(kbb),
     &      sa(kbnx1), sa(kbnxn), sa(kbbnx1), sa(kbbnxn),
     &      sa(kbny1), sa(kbnyn), sa(kbbny1), sa(kbbnyn),
     &      sa(kze), sa(kzf), nx, ny )
```

Figure 8. Array Carving. SPECseis allocates a large array, `sa`, as a work space, depending on the needs of the seismic routines chosen to process the seismic traces. This large work array is carved up into smaller segments and used as a series of arrays.

2.4.2. Array Type Change

A similar problem arises when the type of an array changes between the caller and callee subroutine. Figure 9 gives an example from SPECseis where some arrays are declared real and used as complex within the callee subroutines. This is because it is a large work array. One set of routines use a portion of the work array as a smaller real array and another portion as a smaller complex array.

Certain assumptions of Fortran77 programs, such as the non-aliasing of subroutine parameters and no out-of-bounds array indexing, allow the compiler to be less conservative. These assumptions are no longer possible after the routine that typed `sa` as complex was inlined into the routine that typed it as real. For compiler developers, this means that the compiler should incorporate information it could easily determine before inlining in the data dependence tests after the routine is inlined.

2.5. IO Statements and Loop Exits

One reason Polaris could not determine that one of the main loops of SPECchem (`TWHEIP_do#3`) is parallel was because of an `abort` statement. Of course, the `abort` statement is executed only in rare cases, but the compiler simply sees a conditional and an exit from the loop. In this case, the `abort` statement was hidden deep in a nest of subroutine calls, loop nests, and conditionals, illustrated in Figure 10. The compiler should ignore program exits when searching for data dependencies since the exits occur only in cases of errors, cases where correct program execution is not applicable.

Figure 11 gives another example of conditionals that section off portions of code which are rarely executed but which hinder parallelism. This example is a section of SPECseis showing code that is executed only once per set of seismic traces. The variable `kdepth` is incremented as the wavefield is propagated down in depth one depth step. The first time through, an extra synchronization and transpose is performed. The last time through (`kdepth = nz`), a synopsis is printed to the screen and the routine is exited. Without knowing that these two conditional sections

Caller Routine:

```
        SUBROUTINE MG3D_ZSTEP(...,sa,...)
C       This routine propagates wave-field 1 depth step.
        REAL sa(*)
        ...
C          Extrapolate in x and y directions
           CALL MG3D_XTRAP(...,sa,sa(ksa))
        ...
```

Callee Routine:

```
        SUBROUTINE MG3D_XTRAP(...,vel,sa)
        REAL vel(nx,ny)
        COMPLEX sa(*)
```

Figure 9. Array Type Changing. In SPECseis, the array `sa` is allocated as a large work array. Depending on the seismic routines being performed, the array is carved up into segments which are used as independent, smaller arrays. When Polaris attempts to inline subroutine `MG3D_XTRAP` into subroutine `MG3D_ZSTEP`, it must deal with the array, `sa`, as both a real and a complex array.

```
      SUBROUTINE TWHEIP
        DO iit = 1, npar
          IF (ijkl .EQ. 1) THEN
            CALL GENRAL
              DO 480 kg = 1, ngc
                DO 460 lg = 1, lgmax
                  DO 440 n = 1, nij
                    IF (nroots .GE. 6) CALL ROOT6P
                      DO k = 3,n
                        CALL RYSNOD
                          CALL ABRT
                            CALL abort()

      SUBROUTINE RYSNOD
        ...
        IF (prod .GE. zero) THEN
          IF (maswrk) WRITE(6,15) m, k
          CALL ABRT
          STOP
        endif
        ...
```

Figure 10. Program Exit within a Loop. This example comes from SPECchem. The loop in subroutine `TWHEIP` is parallel but Polaris is forced to be conservative because of the call to `abort` deep within the loop.

```
   IF (kdepth .EQ. 0) THEN
C     Transpose data if first time through
      CALL JSYNC()
C     Stored data volume is ( x, f, y ), with y spread across nodes.
C     Use transpose operations to spread frequencies across nodes.
C     ( x, f, y ) -> ( x, y, f )
      CALL DTRAN132C( nx, nfp, nyp, ra, sa )
C     If up to number of lines, quit
   ELSE IF (kdepth .GE. nz) THEN
      ntro = 0
      IF (node .NE. master) return
      CALL SYSOHDR('MG3D')
      WRITE (ipr,9010) tload, tcomp, tcorr, tcomm, flopsm,
     *       ratee, ratec, flopsm/(tcomp + tcorr + tcomm)
     return
   endif
   ...
```

Figure 11. Rarely Executed Code Sections Cause Conservative Restrictions. The compiler is unable to see that the conditional statements seldom evaluate to "true". `kdepth` is incremented with each propagation of the wave field down in depth. Between the time that `kdepth = 0` and `kdepth = nz`, none of the I/O operations or dependencies that the `DTRAN132C` routine introduces would be relevant.

are executed only once per program run, Polaris assumes that they could be executed each time this code section is invoked. As a result, Polaris sees a possible call to jsync (synchronize MPI processors), `DTRAN132C` (transpose the distributed dimension), a call to `SYSOHDR` (prints out info to the screen), and a premature return within every invocation of this code section. Also, the value of `ntro`, which is important for the following seismic routines, is unknown at the end of this code section. (`ntro` is the number of traces out of this seismic routine which the next seismic routine in the pipeline will process – see the discussion on the driver routine of SPECseis in Section 2.1.1.) A valuable enhancement of the compiler would be to enable it to unroll these special cases from the series of computations, and then analyze the data dependencies.

2.6. Issues Not Specific to Large-scale Applications

We have presented examples and compiler issues for which the context of large, commercially relevant applications makes a difference. In addition, we have discovered a number of problems faced by our compiler that we believe are equally important in smaller applications. One example is the presence of `GOTO` and `WHILE` loops. Parallelizing compilers are well-capable of analyzing Fortran `DO` loops for parallelism. Loops that are formed by `GOTO` statements or `WHILE` constructs can often be turned into `DO` loops. Although such control-flow normalization techniques are well understood, we have found that they may not be part of a compiler's repertoire.

Another important, general issue is the need for data dependence analysis in the presence of subscripted subscripts and pointers. Most parallelizing compilers will not recognize parallel loops that contain such data accesses. We have found that subscript arrays are often only written during the initialization of a program or program phase and from then on are constant. This knowledge could be taken advantage of by a parallelizing compiler.

3. CONCLUSIONS

It is apparent that parallelization is not yet at the level of being fully automatic. With commercial codes we can see little success of automatic parallelization. However, there are clear steps that may be taken to empower automatic parallelizing compilers to produce efficient parallel code. In this paper we have pointed out several key areas that the compiler community should focus on to enable automatic parallelization to become beneficial to developers of large-scale applications. The first aspect of commercial applications that applies to automatic parallelization is the growing amount of modularity. Large applications have modular structure due to their size, the use of software components, and the use of multiple programming languages. We expect that these effects will continue to grow in commercial codes.

The use of legacy libraries is another characteristic of the codes we analyzed. We expect that reuse of library codes will become even more prolific in the future. It is important that compilers be able to undo and revise optimizations within legacy codes that no longer apply to modern architectures.

Symbolic analysis is an important capability of automatic parallelizing compilers, which becomes increasingly complex with larger application codes. In such applications, most program variables are derived from input data and are therefore unknown at compile time. Furthermore, expressions contain intrinsic functions, such as logarithms and modula terms. To respond to such trends, symbolic analysis and manipulation capabilities need continued improvement.

Array reshaping, type changing, and carving add complexity to interprocedural analysis techniques and, in this way, hinder compiler optimizations. Although such practices may be considered bad software engineering, large, commercially-relevant codes often combine a variety of legacy code modules and coding styles. The compiler's ability to deal with this variety of issues is critical for successfully optimizing large-scale applications.

Input/output operations are another impediment to successful parallelization. We have found several program patterns that are parallel, but the compiler could not recognize this fact due to I/O statements. The compiler would have to recognize that certain I/O statements are executed rarely or only in error situations.

The study presented in this paper is only a small step in the direction of understanding the characteristics of large-scale applications. Analyzing such applications takes significant effort. Many more, similar studies will be necessary to help the current generation of compilers to become truly useful tools for the user of real-world, commercial applications.

REFERENCES

1. W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. Parallel programming with Polaris. *IEEE Computer*, 29(12):78–82, December 1996.
2. William Blume and Rudolf Eigenmann. An Overview of Symbolic Analysis Techniques Needed for the Effective Parallelization of the Perfect Benchmarks. *Proceedings of the 1994 International Conference on Parallel Processing*, pages II233 – II238, August, 1994.
3. Rudolf Eigenmann, Insung Park, and Michael J. Voss. Are parallel workstations the right target for parallelizing compilers? In *Lecture Notes in Computer Science, No. 1239: Languages and Compilers for Parallel Computing*, pages 300–314, March 97.
4. Michael W. Schmidt et. al. General atomic and molecular electronic structure system. *Journal of Computational Chemistry*, 14(11):1347–1363, 1993.
5. C. C. Mosher and S. Hassanzadeh. ARCO seismic processing performance evaluation suite, user's guide. Technical report, ARCO, Plano, TX., 1993.