Application of Automatic Parallelization to Modern Challenges of Scientific Computing Industries

Brian Armstrong
Rudolf Eigenmann
School of Electrical and Computer Engineering
Purdue University, West Lafayette, IN 47907-1285

Abstract

Characteristics of full applications found in scientific computing industries today lead to challenges that are not addressed by state-of-the-art approaches to automatic parallelization. These characteristics are not present in CPU kernel codes nor linear algebra libraries, requiring a fresh look at how to make automatic parallelization apply to today's computational industries using full applications.

The challenges to automatic parallelization result from software engineering patterns that implement multifunctionality, reusable execution frameworks, data structures shared across abstract programming interfaces, a multilingual code base for a single application, and the observation that full applications demand more from compile-time analysis than CPU kernel codes do. Each of these challenges has a detrimental impact on compile-time analysis required for automatic parallelization.

Then, focusing on a set of target loops that are parallelizable by hand and that result in speedups on par with the distributed parallel version of the full applications, we determine the prevalence of a number of issues that hinder automatic parallelization. These issues point to enabling techniques that are missing from the state-of-the-art.

In order for automatic parallelization to become utilized in today's scientific computing industries, the challenges described in this paper must be addressed.

1. Failure to Address Today's Computational Needs

There is a peaked interest in automatic parallelization due to the impact that multicore computer systems have on today's market, broadening access to parallel systems, and the growth of data, requiring parallelization for efficient execution. Consequently, there is current incentive to apply automatic parallelization to real applications used in industry (referred to as full, industrial-grade applications) and achieve effective speedups.

Automatic parallelization has proven successful at discovering loop-based parallelism in CPU kernel codes and libraries of vector utilities. Previous work achieved speedups of four on average when CPU kernel codes and linear algebra libraries were executed on eight processor systems.[6], [4], [2]

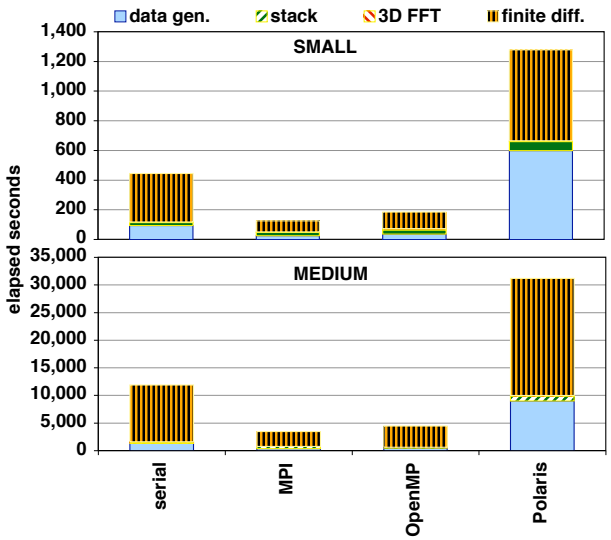

Figure 1. Measured Performance Achieved by Automatic Parallelization of SEISMIC.

However, applying automatic parallelization to full applications typical of industrial-grade codes reveals characteristics of full applications that pose additional challenges not addressed in traditional automatic parallelization techniques and not evident in common CPU kernels or libraries of linear algebra utilities. Using Polaris, a state-of-the-art automatic parallelizing compiler, speedups of four on

0190-3918/08 $25.00 © 2008 IEEE
DOI 10.1109/ICPP.2008.65

eight-processor machines can be achieved with CPU kernel codes and small applications, such as several of the PERFECT BENCHMARKS, yet, Polaris is not effective in finding significant parallelism in SEISMIC, a seismic processing application suite that mimics applications found in the seismic processing industry.

Figure 1 shows the performance of SEISMIC on a four-processor machine, comparing speedups from automatic parallelization by Polaris with speedups from manual parallelization using MPI, which can be considered the ideal in this case. The performance due to manual parallelization of outer loops using OpenMP represents what is feasible to achieve with loop parallelization, showing that loop parallelization can achieve speedups on par with the ideal case using MPI. Each of the two charts compares the performance of four compiled versions of a seismic processing application. The serial version is the base case, run on one processor of a four-processor machine. The "MPI" label identifies the manually parallelized version of the code, which was run on four processors using MPI. The "OpenMP" version includes manual parallelization of the outermost parallel loops using OpenMP directives, which represents a target for automatic parallelization. The "Polaris" version represents the state of the art in automatic parallelization. Data generation, stacking, 3D FFT, and finite difference are four components of the application suite, representing different phases of seismic processing.

Only simple loops were parallelized by Polaris, which is evident from the fact that parallelization overhead is greater than any speedups obtained from running SEISMIC on four processors. The MEDIUM dataset is an order of magnitude larger in terms of the memory required than SMALL, revealing that the poor performance of automatic parallelization is not an artifact of the data size being small. Also, the performance trends among the four versions of parallelization are consistent across each of the four components of the seismic processing application suite, each of which involves different computations, algorithms, and code, and each could be run as a separate executable, implying that the challenges faced by automatic parallelization applies to a variety of cases. In brief, the figure indicates that automatic parallelization is not able to achieve speedups comparable to manual parallelization even when only using the best loop-based parallelism as a basis or when using larger datasets.

The contribution of this paper is to answer the question of how automatic parallelization applies to industrial-grade applications. Even though automatic parallelization is a mature field, most work has been evaluated with small applications and CPU kernel codes. Our work identifies challenges and characteristics of full applications that CPU kernel codes do not exhibit but that must be addressed in order for automatic parallelization to apply to today's computational industries.

The following section uses applications representative of codes in use by scientific computing industries today to identify characteristics of full applications that lead to challenges to automatic parallelization. The codes we consider are SEISMIC, GAMESS, and SANDER[1], representing computational challenges of scientific computing industries in seismic processing, quantum dynamics simulation, and molecular dynamics simulation respectively. We identify challenges that are not present in CPU kernel codes nor linear algebra libraries.

Each of the identified challenges has a detrimental impact on compile-time analysis required for automatic parallelization. Section 3 provides further insight from observing specific target loops in full applications, revealing enabling techniques that are missing from state-of-the-art, automatic parallelizing compilers, and how significant each issue is in terms of the number of target loops that would benefit.

2. Unique Challenges in Today's Industry

Codes used in the scientific computing industry have characteristics that are different from CPU kernel codes and libraries of linear algebra utilities commonly used as benchmarks. Industrial-grade application suites typically include utilities to access file systems, multiple versions of a code for addressing various types of distributed and parallel machines, elaborate input datasets, and many configuration parameters that enable user selection of various functionalities.

The impacts that the key differences between industrial-grade applications and CPU kernel codes have on automatic parallelization are described in the following sections.

2.1. Multifunctionality

Industrial-grade application suites typically encompass a variety of approaches to the same problem. The choice of which techniques to apply is left to the user because it depends on characteristics of the data set and the type of analysis the user desires to do on the results.

For example, consider the option in SANDER to perform minimization or molecular dynamics. To choose to do minimization, a user sets **`imin=1`** in an input file which is read by SANDER at runtime. Within the main program, the **`imin`** parameter determines which subroutines to call. SANDER allows many other options that impact which subroutines are called.

Similar multifunctionality exists in SEISMIC and GAMESS. SEISMIC allows users to choose which computational modules to use and the order in which to apply

[1] SANDER is one component of AMBER that is written in FORTRAN 77 and encompasses the computationally intensive routines.

the modules. Users can select the wavefunction to use in GAMESS from choices such as: restricted or unrestricted Hartree Fock, restricted open shell Hartree Fock, generalized valence bond, and multiconfigurational self-consistent field.

2.1.1 Impact on Automatic Parallelization

For multifunctional applications, the compiler must assume that all combinations of choices are possible since the compiler cannot determine which of the choices a user will select. Since the choice of which computational modules to use is determined by the user at runtime, compile-time transformations and analyses must consider the multiple choices to be possible. Analysis that does not take conditionals into account must approximate to address all possible conditions, which leads to more general but less precise analysis.

Analysis that incorporates conditionals, such as an analysis algorithm that uses Guarded Array Regions[7] or the Gated Single Assignment representation[5],[9], is able to address some cases of multifunctionality but multiplies the amount of analysis that is required with every conditional block whose condition depends on one of the variables used to indicate a selection of the application's options.

Since multifunctionality causes the amount of compile-time analysis required to multiply as the compiler attempts to account for many possible control flow paths, precise analysis can become infeasible in terms of the analysis techniques required to compare array access patterns across control flow paths, even though the paths may never be taken within the same execution in practice. Consequently, the compiler makes conservative assumptions which can reduce the compiler's ability of finding significant parallelism.

2.2. Reuse of the Enclosing Framework

To enable reuse and iterative development without requiring recoding of the application's execution framework, a layer of abstraction is added between the main execution process and the subprocesses containing the computational techniques.

Industrial-grade applications are often made to be extensible and reusable. SEISMIC facilitates extensibility by providing templates for linking the computations of a module to the execution framework of the application suite. Every computational module in SEISMIC has a preparatory routine that abides by the template: MODULEPREP(ldim,maxtrc,nra,nsa). The values of the parameters of **MODULEPREP** are set within the code of **MODULEPREP**. The variable ldim is the size of the leading dimension of the multidimensional array that will hold seismic traces for this module. The variable maxtrc is the maximum number of traces (dependent on the size of the input data set and whether the computational module operates on each individual trace or an aggregation of traces.) The variable nra is the amount of persistent storage this module needs and nsa is the amount of scratch space this module needs.

A SEISMIC module also has several computational subroutines that are called in sequence with subroutines of the other modules. The computational module has a similar template to **MODULEPREP** but also includes the arrays and the number of seismic traces passed in (**ntri**) and that this module will produce (**ntro**) as in: **MODULECOMP(. . .,otra,ra,sa,nrti,ntro)**. Arrays **OTR**, **RA**, and **SA** are allocated outside of the code for the module. Array **OTR** is used for the input data. Array **RA** is for persistent data. Array **SA** is scratch space.

2.2.1 Impact on Automatic Parallelization

Due to a layer of abstraction, compiler analyses and transformations must function with limited knowledge of the control flow across computational modules. The compiler can determine the control flow for portions of the code, such as the control flow within the code of a computational module in SEISMIC, but it is not feasible for the compiler to determine the full global control flow of large application suites since any computational module may follow any other. Without control flow information, analysis techniques, such as dataflow analysis, cannot be performed across a layer of abstraction.

2.3. Shared Data Structures

Data structures can be shared across the layer of abstraction mentioned above. Consequently, the same data structures, allocated in outer contexts, may be used by multiple computational modules to house different types of data. The computational modules employed by an industrial application suite can be diverse in terms of their data access patterns and the types of data they operate on.

A shared data structure holding wave function data is used in GAMESS by the computational modules that perform the wave function calculation chosen by the user. As mentioned above, possible wave function calculations include restricted and unrestricted Hartree Fock, restricted open shell Hartree Fock, generalized valence bond, and multiconfigurational self-consistent field. The choice of wave function calculation and additional features of either the dataset or the preferences of the user impacts the size and shape the shared data structure must have to facilitate the selected wave function computations.

Consider the **JKDER** subroutine which contains a significant parallelizable loop. Within the loop are calls to

DABGVB and **DABDFT**, both of which use a portion of array **X** indexed from **LVec**. The choice of whether to call **DABDFT** and **DABGVB** depends on input parameters chosen by the user. **ROGVB** is set to true if the user wants to do generalized valence bond calculations. **HFSCF** is set to true due to other user selections involving the wave function calculation and whether or not to apply the Moller-Plesset perturbation theory.

Subroutines **DABDFT** and **DABGVB** both access **X[LVec...*]** but for different data. **DABDFT** declares the memory at **X[LVec]** as a single dimension array and accesses it within a loop using an indirect array access through array **IA**. In contrast, **DABGVB** declares the memory at **X[LVec]** to be a two-dimensional array, **V**, and accesses **V** within an additional nested loop. The array access patterns to the same segment of memory as well as the declared shapes of the array segment differ among subroutines called within the parallelizable loops of **JKDER**.

The arrays **RA** and **SA** used by the generic framework for computational modules in SEISMIC are other examples of shared data structures that hold different data depending on user input selections. If the user selects the **M3FK** routine to execute, **RA** is used as a series of two dimensional arrays that holds complex numbers. In contrast, if a user selects the **STAK** routine to execute, **RA** is used as a three dimensional array that holds floating point numbers.

Both **M3FKB** and **STAKB** are called from a loop in **SEISPROC**. The user sets up an input file to determine which routines will be called from **SEISPROC** and in which order. At compile-time, we cannot know whether **M3FKB**, **STAKB**, or both will be called during runtime. Therefore, the compiler assumes that both are called and that **RA** is used in both ways described above.

2.3.1 Impact on Automatic Parallelization

Shared data structures impact automatic parallelization because many parallelization techniques rely on describing the portion of memory accessed by an array expression in the context of the declaration of the array. The declared sizes and shapes of shared data structures may not relate to the access patterns in the code for a particular computational module because the sizes and shapes of shared data structures are made to be general enough to facilitate the memory needs of all the computational modules.

State-of-the-art automatic parallelization techniques fail to perform precise comparisons among an array's accesses when the size and multidimensional shape of the representation of an array's accesses are not clearly defined portions of the size and shape used to describe the array's declaration. The result is that the compiler makes conservative assumptions that may limit the amount of parallelism the compiler is able to discover.

2.4. Multilingual Code

We refer to applications with code from multiple languages as multilingual. Multilingual applications are becoming increasingly common as code is reused in multiple contexts. It is common to find multilingual applications in use in industry since much of the code of a full application suite is reused for a number of years as new techniques and applications emerge.

The SEISMIC application consists of FORTRAN and C. C is used to perform memory allocation and low level file activities. The main program is written in FORTRAN 77. The main program calls a C routine to allocate memory. The C routine calls several FORTRAN 77 routines that perform the data processing. Other low-level C routines are called from within the data processing subroutines to store and read files on disk.

In SEISMIC, the main program encloses the calls to **SEISPREP** routines, written in FORTRAN 77, where relationships are applied among input parameters and some common block variables. Data structures are allocated in **CPROC**, written in C, using variables passed in from the main program. The data structures allocated in **CPROC** are then used by FORTRAN 77 subroutines called from **CPROC**. The compiler is unable to link the use of variables in array index expressions to the declarations of the arrays because the size of the array is determined from within the C code.

2.4.1 Impact on Automatic Parallelization

Most compilers take code of only a single language as input. Compilers that handle more than one language translate to a low-level, intermediate representation and do not leverage compile-time analysis across subroutines programmed in multiple languages at the high level required for automatic parallelization. Consequently, compilers typically make the most conservative assumptions about the possible side effects of executing code in a different language, which prevents parallelization of loops enclosing code with multiple languages.

2.5. Compile-Time Complexity

Using SEISMIC, GAMESS, and SANDER to represent full applications, and the PERFECT BENCHMARKS and LINPACK to represent CPU kernel codes, we measure the amount of elapsed time required by Polaris to compile the various codes. Traditional automatic parallelization techniques cannot be applied to a code when the compile time required to parallelize the code exceeds "reasonable" time and memory limits on a modern workstation. We subjectively define the threshold for what a reasonable time limit

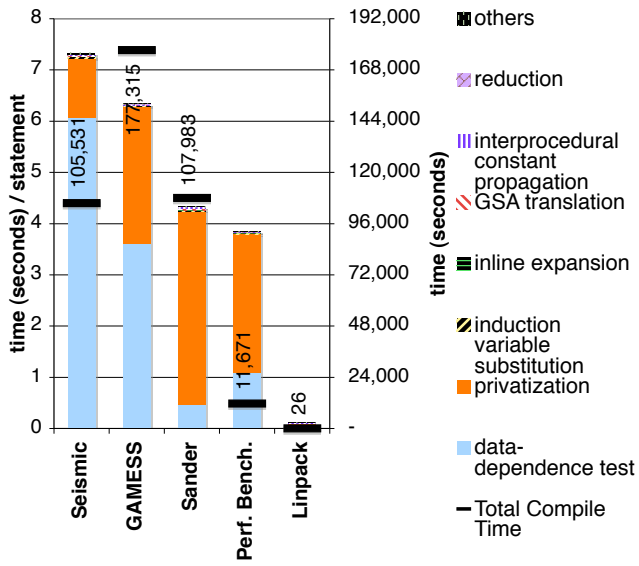

Figure 2. Compile Time per Code Statement

is to be twelve hours of elapsed time on a modern workstation, and a reasonable memory limit to be four gigabytes. If reasonable compile-time limits are exceeded when applying Polaris to all of the code for an application, a smaller portion of the call graph is compiled in isolation. The portion of the call graph to compile is chosen by selecting a subroutine, all of the code that is reachable through subroutine calls from within the selected subroutine, and all subroutines enclosing the selected subroutine. This approach is applied to a number of subroutines until the majority of the code is included in the results.

The elapsed compile time measure graphed in Figure 2 provides a quantitative means to compare the complexity of compiling full applications with that of compiling CPU kernel codes. The total elapsed compile time is displayed as a black dash at a height corresponding to the axis on the right. The columns show the total compile time divided by the number of FORTRAN statements compiled. The segments of the columns indicate a breakdown by compiler pass. For the PERFECT BENCHMARKS and LINPACK, the total compile time is averaged across the codes since each code is executed independently. For each of SEISMIC, GAMESS, and SANDER, the entire set of codes is considered as one application. Therefore, it is expected that the required time to compile the full applications is an order of magnitude larger than for compiling one of the CPU kernel codes.

The columns of Figure 2 divides the total compile time by the number of FORTRAN statements to provide a measure of the complexity per statement. The SEISMIC and GAMESS applications require significantly more compile time in order for Polaris to apply automatic parallelization techniques per statement than PERFECT BENCHMARKS and the complexity of compiling LINPACK codes is insignificant with respect to all the other benchmarks. Consequently, full applications not only have more code to compile but also require more resources to compile the same amount of code as CPU kernel codes.

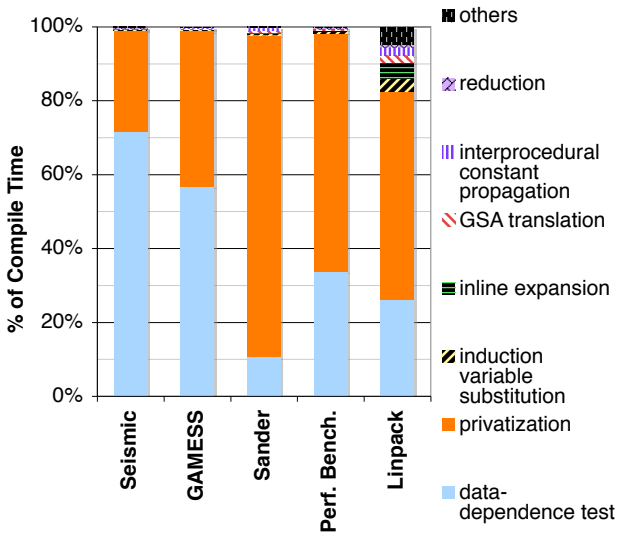

Figure 3. Compile Time per Compiler Pass

To understand why industrial-grade codes require more effort to compile a statement, note that the majority of compile time is spent in the data dependence and array privatization passes, as indicated by the segments of the columns. Figure 3 shows the significance of the main compiler passes with respect to the elapsed compile time for each benchmark. The data dependence tests and array privatizations dominate the compile time in all cases, though the other compiler passes are more significant to the average PERFECT BENCHMARKS or LINPACK code. Both the data dependence and array privatization pass use symbolic analysis. Symbolic analysis is used to resolve comparisons of array accesses across different loop iterations to determine if they reference the same memory locations. Induction variable substitution and reduction recognition also use symbolic analysis but only when specific patterns exist.

2.5.1 The Nesting Depth of Parallel Loops

To understand why the use of symbolic analysis by data dependence tests and array privatization is more expensive in terms of compile time for full applications, consider a case study involving loops known to provide significant speedup in the overall execution time of SEISMIC

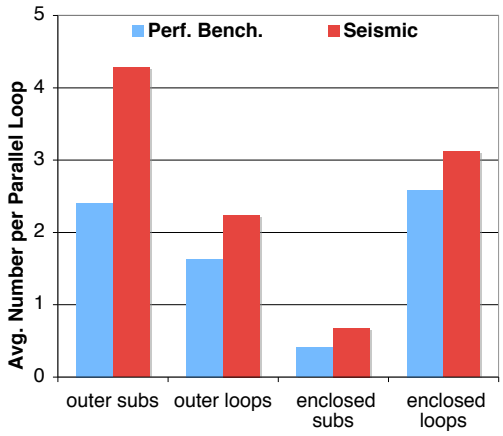

Figure 4. Nesting Characteristics of Loops Manually Identified as Parallel

and the PERFECT BENCHMARKS. We refer to these loops as the "target loops". Target loops are identified in SEISMIC through manual analysis. When the target loops of SEISMIC are parallelized, we obtain overall speedups on par with the hand parallelized version of SEISMIC. We leverage work on manual parallelization of the PERFECT BENCHMARKS, performed under the Center of Supercomputing Research and Development at the University of Illinois at Urbana-Champaign, to identify target loops of the PERFECT BENCHMARKS. Parallelizing the target loops serve as a goal of automatic parallelization.

Figure 4 shows that the "nesting depths" of target loops in SEISMIC are significantly greater than those of target loops in the PERFECT BENCHMARKS. A loop within a loop is referred to as a nested loop. Likewise, a subroutine called from within another subroutine is referred to as a nested subroutine. We sum up the number of loops and subroutines that enclose a loop to specify the total nesting depth of the loop. The figure shows the average of the number of loops and subroutines for loops that have been selected for hand parallelization. "Outer subs" and "outer loops" are the number of subroutine calls and loops from the program level to the target loops in the deepest call graph paths. "Enclosed subs" and "enclosed loops" are the number of subroutine calls and loops enclosed within the target loops.

Figure 4 breaks down the total nesting depths by the contexts enclosing and contexts enclosed by the target loops. The break down reveals that target loops in SEISMIC are enclosed by significantly more subroutines than target loops in PERFECT BENCHMARKS.

The challenges to parallelizing full applications, described in the previous sections, typically involve using additional subroutine calls. The PERFECT BENCHMARKS were created by extracting a smaller portion of the call graph that encompassed computationally intensive part of scientific applications and explicitly assigning any required variables that are defined in the outer contexts to static values, which is why the outer subroutine nesting depths for the target loops in the PERFECT BENCHMARKS are much smaller than for the target loops in SEISMIC, whereas the enclosed subroutine and loop nesting depths are more similar across the two types of codes.

2.5.2 Impact on Automatic Parallelization

A larger loop nesting depth requires additional symbolic analysis for data dependence and array privatization analysis, resulting in a greater compile-time complexity for parallelization. Since the compiler analyzes an array reference for cross-iteration dependencies for each enclosing loop, the Range Test permutes the loops in a loop nest to determine which loops are parallel, and the compiler compares array references in different loops when the loops share a common enclosing loop, the amount of symbolic analysis required relates to the loop nesting depth.

The amount of symbolic analysis required also relates to the subroutine nesting depth since interprocedural analysis or inlining must be used to translate array access patterns within subroutines into the calling contexts of the subroutines in order to analyze cross-iteration dependencies in any loops enclosing the subroutine calls. Subroutines enclosing a loop can require symbolic analysis if an array referenced within the loop is declared in a calling subroutine.

3. Issues Preventing Effective Automatic Parallelization

What prevents sufficient precision required for automatic parallelization of full applications? The challenges outlined in Section 2 each impact application of traditional automatic parallelization techniques. To determine the way forward for automatic parallelization, we categorize the issues that must be addressed in order to enable traditional parallelization techniques to be effective with industrial-grade applications. The following issues prevent the automatic parallelization of a set of target loops of SEISMIC, GAMESS, and SANDER: rangeless variables, aliased arrays, array indirection, complex symbolic expressions, complex array access patterns, and compile-time complexity.

Each target loop is identified manually with one of the categories that prevents its parallelization in Figure 5. Only loops in the *autoparallelized* category were found to be parallel by Polaris using state-of-the-art techniques. For loops in the *aliasing* category, Polaris incorrectly assumed that dependencies exist between subroutine array parameters that

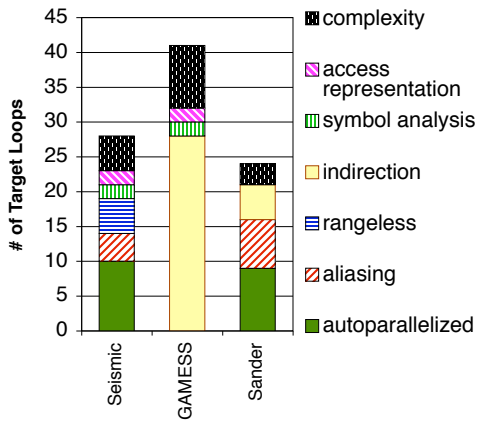

Figure 5. Remaining Hindrances to Automatic Parallelization of Target Loops

are aliased. We refer to variables for which the compiler cannot determine the limits on the possible values the variables can hold as *rangeless* variables. When comparison of array index expressions depends on rangeless variables, symbolic analysis of the comparison is futile and the compiler must resort to conservative assumptions. Target loops for which Polaris could not perform precise symbolic analysis due to the presence of rangeless variables are part of the*rangeless* category in the figure.

Loops with array indirection were not parallelized because Polaris had no support for analysis of arrays indexed by arrays, indicated by the *indirection* category. The *symbolic analysis* category encompasses cases that require additional symbolic analysis capabilities than what is implemented in Polaris. The *access representation* category includes cases where the array access representation used within Polaris is not sufficiently precise for determining that specific array references do not contain cross-iteration dependencies. Symbolic analysis required to analyze loops in the *compile-time complexity* category exceeds reasonable compile-time limits, preventing Polaris from parallelizing the loops.

4. Related Work

Excessive compilation time has been addressed in techniques that trade precision for compile-time efficiency, such as interprocedural techniques that summarize array access patterns per subroutine and reuse the array summaries across multiple call sites for the same subroutine. However, compiler techniques do not exist to determine the minimal amount of precision needed to effectively parallelize an application to achieve speedups.

To address compile-time complexity that prohibits whole program, interprocedural optimization for the purpose of register scheduling, instruction-level parallelism, and even such high-level optimizations as loop parallelization, techniques have been proposed for reducing the amount of the program to consider for analysis at any one time while still allowing the benefits obtained from interprocedural optimization. A Procedure Boundary Elimination framework was proposed that automatically determines portions of the call graph that can be analyzed as completely encapsulated units [8]. Each portion of the call graph is supplied with information from the rest of the whole program call graph that is useful for supporting optimization of the portion. This approach to enabling interprocedural optimization is promising when compile-time complexity is prohibitive. In order to apply automatic parallelization in Polaris to the target loops in Section 3, appropriate portions of the whole program call graph were manually extracted, always starting with the main program or an outermost routine and including all subroutines called from within the target loop. An approach such as Procedure Boundary Elimination may be able to automatically determine the portion of the call graph to consider. The heuristic used to automatically determine the portion of the call graph to consider, and which resulted in speedups of near ten percent for low-level, classical compiler optimizations, register allocation and scheduling, does not identify the software engineering patterns described in Section 2. Considering different optimizations, such as high-level loop parallelization, requires different heuristics that address the challenges that are not present in CPU kernel codes.

Additionally, some of the portions of the whole program call graphs selected for compiling the target loops were prohibitive in terms of compile-time complexity when applying loop parallelization techniques, meaning that even if the compiler is able to apply a heuristic that mimics what we did manually to identify the portion of the call graph to compile, the analysis required to parallelize at the target level is prohibitive. In other words, determining how to make interprocedural optimization feasible still must be addressed in order to parallelize full applications and achieve results comparable with manual parallelization efforts.

Blended analysis is an approach developed in order to address issues similar to the challenges caused by multifunctionality discussed in 2.1, but in the context of framework-based, object-oriented applications. Authors of [3] describe a hybrid approach to enable post-execution understanding of performance when the execution path through the call graph is unknown at compile time. However, this technique only applies to a specific configuration of input parameters, which is appropriate when addressing similar usage patterns in framework-based applications, but is not as easily applied to codes used in this paper since

slight changes to input parameters can have significant impact on the available parallelization.

In [1], we described some of the challenges expounded on in this paper in Section 2 using only SEISMIC and GAMESS as examples of industrial-grade applications. In this paper, we have described the impact that each of these challenges has on automatic parallelization. We also added SANDER as another example of an industrial-grade scientific computing application. Whereas our previous paper identified challenges, this paper provides a metric to measure the complexity of these full applications with respect to automatic parallelization, which is not evident from simple measures of the number of lines of code or number of subroutines. Using this metric, we compare the complexities of industrial-grade applications to CPU kernel and linear algebra benchmarks to quantify how much more complex full applications are to compile than several well known benchmarks used to evaluate compiler performance. Lastly, we investigated a set of target loops in the full applications to determine what hindered state-of-the-art automatic parallelization techniques.

5. Conclusion

In the process of analyzing why applying traditional automatic parallelization techniques to full applications fails to achieve speedups comparable with the performance achieved using CPU kernel benchmarks, a number of characteristics unique to full applications are identified. Each of these characteristics impacts automatic parallelizing compilers by hindering ability to find profitably parallelizable loops using traditional techniques.

The challenges posed by industrial-grade applications include software engineering practices that support user-selectable multifunctionality, extensible and reusable execution frameworks, and data structures that store generic types of data shared for use by diverse computational routines. Industrial-grade applications typically leverage libraries and support utilities written in different languages, requiring the compiler to be capable of compiling multiple languages in order to analyze all subroutine calls.

Additionally, we show that applying automatic parallelization techniques to full applications is more complex than applying these techniques to CPU kernel benchmarks, which ultimately limits the application of automatic parallelization to subsets of the call graph to keep the compile time reasonable. Further analysis of the compile-time complexity indicates that the requirement of symbolic analysis capabilities to support the Range Test and array privatization is responsible for most of the elapsed compile time.

We show that profitably parallelizable loops in SEISMIC have greater nesting depths than such loops in the PERFECT BENCHMARKS. The nesting depth of a loop impacts the complexity of performing the Range Test and array privatization on the nest of loops and subroutines. The resulting complexity of compiling a statement of the full applications considered in this paper is shown to be greater than the complexity of compiling a statement of the PERFECT BENCHMARKS and significantly higher than statements of LINPACK, indicating the unique challenge posed by full applications in terms of the demands placed on compile-time performance.

Finally, we categorize the remaining challenges needed to profitably apply automatic parallelization to SEISMIC, GAMESS, and SANDER with the goal of achieving results on par with manual parallelization. In order for automatic parallelization to become utilized in today's scientific computing industries, the challenges described in this paper must be addressed.

References

[1] B. Armstrong and R. Eigenmann. Challenges in the automatic parallelization of large-scale computational applications. In *Proceedings of SPIE/ITCOM 2001*, volume 4528, pages 50–60, Aug. 2001.

[2] 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*, pages 78–82, Dec. 1996.

[3] B. Dufour, B. G. Ryder, and G. Sevitsky. Blended analysis for performance understanding of framework-based applications. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, pages 118–128, New York, NY, USA, 2007. ACM.

[4] M. W. Hall, J.-A. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12):84–89, Dec. 1996.

[5] P. Havlak. Construction of thinned gated single-assignment form. In U. Banerjee, D. Gelernter, A. Nicolau, and D. A. Padua, editors, *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, pages 477–499, London, UK, 1994. Springer-Verlag.

[6] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[7] Z. Li, J. Gu, and G. Lee. *Interprocedural Analysis Based on Guarded Array Regions*, volume 1808 of *Lecture Notes in Computer Science*, chapter 7, pages 221–246. Springer-Verlag New York, Inc., New York, NY, USA, 2001.

[8] S. Triantafyllis, M. J. Bridges, E. Raman, G. Ottoni, and D. I. August. A framework for unrestricted whole-program optimization. In *PLDI '06: Proceedings of the 2006 Conference on Programming Language Design and Implementation*, pages 61–71, New York, NY, USA, 2006. ACM.

[9] P. Tu and D. A. Padua. Gated SSA-based demand-driven symbolic analysis for parallelizing compilers. In *Proceedings of the 9th International Conference on Supercomputing*, pages 414–423. ACM Press, 1995.