An Approach to Parallelizing Fortran Programs using Rewriting Rules Technique

Anatoliy Doroshenko[1] and Kostiantyn Zhereb[1]

[1]Institute of Software Systems of National Academy of Sciences of Ukraine, Glushkov prosp. 40, 03187 Kyiv, Ukraine
doroshenkoanatoliy2@gmail.com, zhereb@gmail.com

Abstract. We present an ongoing research in the area of transforming existing sequential Fortran programs into their parallel equivalents. Our approach is to use rewriting rules technique in order to automate the transformation process. Sequential source code is transformed into parallel code for shared-memory systems, such as multicore processors. Parallelizing and optimizing transformations are formally described as rewriting rules which facilitates their reuse. Using high-level algebraic models allows describing program transformations in a more concise manner. Performance measurements demonstrate high efficiency of obtained parallel programs.

Keywords: rewriting rules technique, algebraic program models, multicore processors, Fortran, OpenMP.

Key Terms: High Performance Computing, Model, Methodology.

1 Introduction

Despite being one of the first programming languages, Fortran is still widely used, in particular for solving scientific and engineering computation-intensive problems. Its popularity is due to its relative simplicity and lack of complex facilities (e.g. pointers), closeness to mathematical description of problem and efficiency of generated binary code. Another reason for continued use of Fortran is that in more than 50 years of its existence a vast repository of programs, libraries and routines for solving different scientific problems has been created. Algorithms implemented in such programs are still valuable, however there is a need to adapt this legacy code to new parallel computational platforms. Furthermore, due to size and complexity of existing code, manual adaptation is not a practical option: there is a need for automated tools to facilitate conversion of legacy code to modern parallel platforms [5].

In this paper we describe an ongoing research on parallelizing Fortran programs using rewriting rules technique. Sequential source code is transformed into parallel code for shared-memory parallel platform (such as multicore processors) using automated transformations. Parallelizing and optimizing transformations are formally described as rewriting rules which facilitates their reuse. Such approach is aimed at two main goals: to improve runtime efficiency of programs and to increase developer's productivity. We illustrate our approach on two sample programs: a simple Gauss elimination algorithm and an applied problem of calculating electron density from the field of quantum chemistry.

There has been an extensive research in the area of parallelizing existing sequential code, in particular for multicore architectures. Some approaches require manual code modification and provide facilities that help a developer express parallelism. Such approaches include parallel libraries [13], parallel extensions to existing languages [14] and new parallel languages [16]. Another research direction is interactive parallelization [11], when a developer manually selects the loops to be parallelized, and the tool applies transformation automatically (our approach also belongs to this category). Finally there are numerous approaches to automated parallelization, mostly implemented as parallelizing compilers [1]. Such systems use the static analysis of the source code to detect possible areas of parallelism and generate parallel binary code. Some papers also use the dynamic analysis to detect parallelism based on concrete input data [15], or machine learning approaches to select most appropriate transformations [17], or auto-tuning to discover optimal parameters of transformations [6]. The key differences of our approach is the use of the source-to-source transformations, allowing the developers to examine transformed program code, and the description of the transformations in terms of the formal models and rewriting rules, making easier for developers to add new parallelizing transformations or to modify existing ones.

This paper continues our research on automation of process of designing and development of efficient parallel programs, started in [3], [8], [9]. Our previous papers [3], [9] applied a similar approach to the development of parallel programs written in C# language for Microsoft .NET framework, while this paper concentrates on parallelizing Fortran programs. We have already described our first experiences with Fortran programs in [8]. However, as we moved from simple examples to real-world legacy code, we were forced to revise our approach, as described in this paper (see section 2). Also this paper places more significance on choosing place of application of existing program transformation, rather than developing new transformations.

Below we describe our approach in more detail, provide examples of parallelizing transformations and illustrate them with parallelization and evaluation of two programs: small example program (Gauss elimination) and applied quantum chemistry problem (electron density).

2 Our Approach: Algebraic Models and Rewriting Rules

As in our previous works [3], [8], [9], we use formal facilities, namely rewriting rules technique and high-level algebraic models of programs, to automate parallelizing existing sequential code. Legacy source code of sequential program written in Fortran is transformed into parallel version targeting shared-memory parallel platform (multicore processors). As a part of transformation process, we create high-level algebraic models of legacy source code based on Glushkov algebra [2]. As described in [3], the models are created in two steps. First we use target language parser (Fortran in this paper) to build low-level syntax model, and then rewriting rules of special form (*patterns*) to extract language-independent algebraic operators from language constructs. Using high-level algebraic models allows describing program transformations in more concise manner. The additional benefit of such models when applied to legacy code is that they aid in understanding of algorithms by hiding the (frequently obsolete) implementation details. To this end, using multiple levels of algebraic models can be useful – e.g. the highest level describes just general structure of algorithm, while lower levels supply implementation details (the example of such models is described in section 3).

After high-level program model is created, we use parallelizing transformations to implement a parallel version of the program on a given platform. Transformations are represented as rewriting rules and therefore can be applied in automated manner. (Selection of loops that could be transformed is performed manually.) The declarative nature of rewriting rules technique simplifies adding new transformations. Also transformations work with high-level model elements (on any level of abstraction), which means they are language-independent.

Usage of high-level algebraic models also allows proving correctness of the developed transformations [3]. Based on program models, we have developed the algebra-dynamic models of program execution for multicore architecture using discrete dynamic (transitional) systems [2]. For these models, we have (manually) proved that each of proposed code transformations is correct under certain conditions, i.e. that initial and transformed programs are equivalent.

To automate program transformations we use the rewriting rules system Termware [7]. Termware is used to describe transformations of *terms*, i.e. expressions of form $f(t_1,\ldots, t_n)$. Transformations are described as Termware *rules*, i.e. expressions of form `source [condition]-> destination [action]`.

Here `source` is a source term (a pattern for match), `condition` is a condition of rule application, `destination` is a transformed term, `action` is additional action that is performed when rule fires. Each of 4 components of a rule can contain variables (denoted as `$var`), so that rules are more generally applicable. Components `condition` and `action` are optional. They can execute any procedural code, in particular use the additional data on the program.

Termware supports a number of evaluation strategies, including TopDown (used in this paper), BottomUp and a possibility to implement additional strategies. Termware system itself doesn't check that transformation process terminates, however the rules used in this paper are designed in such way that each model element is processed at most once, therefore the transformation process is guaranteed to terminate.

In addition to rewriting system, our tools include parsers and generators for target languages that perform transformation between source code and low-level (syntax) program model, which is represented as Termware term. We have previously developed such tools for C# language [3], [9]; in this paper we have developed a Fortran parser and generator based on GCC Fortran Compiler.

3 Parallelization for Shared-memory Systems Using OpenMP

In this section we describe the process of parallelizing sequential Fortran programs for parallel systems with shared memory, such as multicore processors. We parallelize source code of Fortran programs by replacing suitable loops with parallel loop constructs. To create multithreaded Fortran program we use OpenMP framework [14]. OpenMP *PARALLEL DO* directives are used to parallelize loops. For simple loops, just addition of such directive can produce quite efficient parallel code. In this case there is additional advantage of keeping transformed parallel code similar to existing sequential code. In more complex cases (when there is data dependency between iterations) there is a need of more significant transformations, such as using OpenMP library subroutines for advanced thread management. In such cases, the transformed source code contains significant changes. However, usage of high-level algebraic models allows describing these changes in concise and understandable form.

We will describe the details of our approach using as an example a Fortran program implementing Gauss elimination algorithm for solving systems of linear algebraic equations. The Fortran source code was transformed into a low-level syntax model using developed parser, then into a high-level algebraic model using Termware patterns. When working with legacy code, we found it useful to apply several levels of patterns. First we used generic linear algebra patterns, such as vector and matrix operations. The obtained algebraic model was language-independent, but still quite detailed. Then we applied patterns specific to the problem in question. In this way we obtained schematic representation of algorithm useful for its understanding and deciding where parallelizing transformations should be applied.

The high-level model of relevant fragment of program has the following form:

```
DoCnt(K,1,N-1,
   FindMaxElement, CheckDetZero, SwapMaxRowColumn,
   CalculateRow(K),  UpdateElements )
```

We will parallelize only two of the operators present in program, namely *FindMaxElement* and *UpdateElements*. Other operators have less computational complexity, therefore their parallelization is less effective.

Out of two operators, the simplest is *UpdateElements*, responsible for calculating new values for elements of submatrix:

```
UpdateElements = DoCnt(I,K+1,N, Assign(S,A(I,K)),
                 DoCnt(J,K,N+1, Update(A(I,J),S)))
```

Here, `DoCnt` denotes common DO loop with counter. The iterations of the outer loop are independent, so this fragment is easily parallelized. We use the following rewriting rule:

```
DoCnt($var,$start,$end,$body,_MARK_Parallel)->
              ParallelDoCnt($var,$start,$end,$body)
```

The loop to be transformed is marked with *_MARK_Parallel* symbol to enable rule application. *ParallelDoCnt* operator is high-level model element responsible for parallel loop. In particular, for OpenMP platform it is transformed into *OmpParallelDo* operator that describes OpenMP directive represented in Fortran as a pair of special comments: *!$OMP PARALLEL DO … !$OMP END PARALLEL DO.*

Notice that for C language the same operator is represented as a single pragma statement: *#pragma omp parallel for*. Therefore, using multiple levels of patterns allows us to provide operators that are common for given platforms, use these generic operators in most rewriting rules and then specialize them only when transforming program model back into source code.

While *UpdateElements* operator can be parallelized by simple application of OpenMP directive, the other operator *FindMaxElement* is more complex. It also has the form of loop, but iterations of the loop update the same set of variables (value of the maximum element in submatrix and its indices). This is the case of reduction, when some local values are calculated on each iteration and then merged into one global value. OpenMP supports such cases with REDUCTION clause, however only a set of predefined reduction operators are supported: while finding just maximum value can be accomplished using OpenMP directives, finding maximum value and indices where it occurs is not directly supported.

Therefore we need to provide transformations that parallelize the loop in general case of reduction. We represent `FindMaxElement` as following combination:

```
FindMaxElement=FindMaxElLoc1*…*FindMaxElLocTN*FindMaxEl
Reduct
```

On each thread we execute local version of operator (`FindMaxElLoc1,…, FindMaxElementLocTN`), and then execute reduction operator `FindMaxElReduct` that combines local values into one global value.

Both already described parallelizing transformations are aimed at high-level structure of algorithm. However, as we observed in [3], low-level implementation details, in particular memory access, can have profound impact on overall performance.

In the Gaussian elimination program we have observed the same effect. We noticed that for certain sizes of input matrix (N=256*M) there was a sudden increase of execution time. We attribute this increase to the peculiarities of memory access: namely, caching adjacent matrix elements. For such matrix size, the adjacent matrix elements were put into the same cache items, therefore increasing the number of cache misses and greatly reducing overall performance. To overcome this peculiarity, we declare the matrix size as N+1 instead of N. The extra elements are not used in calculations, but they change location of elements and improve efficiency of memory access. The transformation is implemented with the following rules:

```
1. [Declaration(N,Integer,$val):$next]
   ->[Declaration(N,Integer,$val):
   [Declaration(MN,Integer,$val+MShift($val)): $next]]
2. MShift($val) [$val%32==0]->1 !->0
```

```
3. Declaration(A,Array(Double,[N,N+1]))
   -> Declaration(A,Array(Double,[MN,MN+1]))
4. Procedure($name,[N:[A:$next]])->
   Procedure($name,[N:[MN:[A:$next]]])
5. [Parameter(N,Integer,In):$next]
   -> [Parameter(N,Integer,In):[Parameter(MN,Integer,In):
   $next]]
6. Call($name,[N:[A:$next]])
   -> Call($name,[N:[MN:[A:$next]]])
```

The rule 1 adds new parameter, MN, denoting declared matrix size. The rule 2 specifies for which values of matrix size the transformation should be applied. The rule 3 modifies matrix declaration to use new size MN instead of N. Rules 4-6 propagate new parameter to all procedures, procedure parameters and procedure calls.

Notice that rules 4-6 are applied multiple times in a single program: for each procedure definition (rules 4-5) and for each procedure call (rule 6). One of the advantages of rewriting rules technique is that single rule can describe changes in multiple places, reducing effort to make the changes and preventing mistakes possible when applying such changes manually. Notice also that rules 1-6 work on lower level of abstraction compared with previously described rules. The ability to describe transformations on different model levels is another advantage of proposed approach and it allows describing different types of transformations with the same tools.

4 Performance Evaluation: Test Program and Real-world Example

To evaluate effects of developed transformations, we have measured the performance of different versions of initial program of Gauss elimination. We have compared performance of 4 versions: initial sequential program (SEQ), parallel program with *UpdateElements* operator parallelized (PAR1), parallel program with both *UpdateElements* and *FindMaxElement* operators parallelized (PAR2), and program with both operators parallelized and memory optimization applied (MEM). The measurements were performed on 4-core parallel system, for matrix sizes from 256 to 2048. Obtained speedup (compared with SEQ program) is shown on fig. 1.

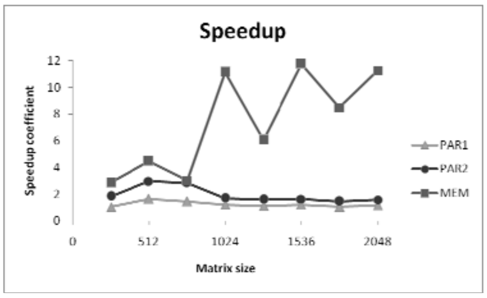

Fig. 1. Speedup of transformed programs (Gauss elimination).

As can be seen from the diagram, all transformations result in some performance increase, although their effect differs. For small matrix sizes, both PAR1 and PAR2 show some noticeable speedup, while MEM is not very effective and is very close to PAR2. However, for larger matrix sizes (N>1024), the situation changes. PAR1 and PAR2 become less efficient, close to SEQ. However, MEM becomes much more efficient and demonstrates speedup of more than 10x. Therefore both high-level transformations of algorithms and taking care of low-level implementation details is necessary to obtain efficient parallel programs. Measurement results also demonstrate complex dependency of execution time on real parallel systems, as compared to ideal theoretical models that suggest simple $O(N^3)$ dependency.

After developing our tools on sample problem (Gauss elimination) we have tried them on real-world program in area of quantum chemistry. The program calculates electron and spin density in atoms of polycyclic aromatic hydrocarbons on N*N grid [12]. The size of the program is 1680 lines of Fortran code. Source code is not well structured – actual calculations are mixed with I/O operations, debug code and some hardcoded data. Also it contains mix of features from different versions of language – from Fortran 77 to Fortran 95. Therefore usage of high-level algebraic models helped us to understand this legacy code and apply parallelizing transformations in most efficient way.

We were able to reuse parallelizing transformations developed for Gauss elimination program also in electron density program. Only the first, most simple loop transformation was applied. However, the challenge was to select the most suitable loop for this transformation, as the program contained 54 loops and trying all of them was not a feasible option. We have used a profiler tool, Intel VTune Amplifier [10], to find hotspots in source code. Then we applied rewriting rules technique to detect all loops enclosing such code fragments. Thus the number of candidate loops was significantly reduced from 54 to 6. Out of these 6 loop, we applied transformation to second outermost loop (as the outermost loop contained too few iterations, and parallelizing inner loops was less efficient because of repeated cost of creating and synchronizing threads each time inner loop was executed).

We have compared execution time of initial sequential program (SEQ) and parallelized program (PAR) for grid dimensions N from 200 to 800 (see fig. 2).

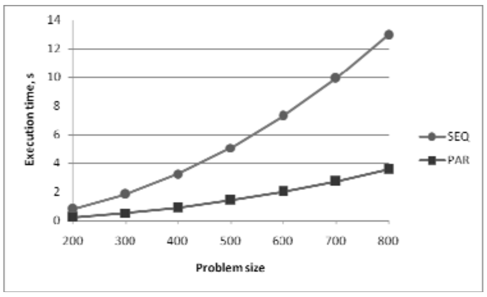

Fig. 2. Comparison of initial and transformed program (electron density).

Applying transformations has resulted in quite significant speedup – from 3.3X to 3.6X (depending on problem size) on 4-core system.

Conclusion

In this paper we have described our approach for parallelizing Fortran programs by applying formalized program transformations to existing sequential Fortran code. Using rewriting rules technique automates application of transformations and prevents mistakes that can appear when applying changes to source code manually. High-level algebraic models simplify understanding of legacy programs and their transformations, and enable transformation on different levels of abstraction. We have applied our approach both to simple program implementing Gauss elimination algorithm and real-world quantum chemistry problem (calculating electron density). Performance measurements demonstrate significant speedup for both programs.

Further research directions include development of the same approach for transforming legacy Fortran applications to target distributed-memory systems and GPUs. Our future plans also include extension to Grid and cloud-based platforms. Also we are planning to improve support for large and complex Fortran programs, in particular automate selection of most suitable place of application for transformations.

References

1. Allen, R., Kennedy, K.: Optimizing Compilers for Modern Architectures: A Dependence-Based Approach. Morgan Kaufmann, San Francisco (2001).
2. Andon, P.I., Doroshenko, A.Yu., Tseitlin, G.O., Yatsenko, O.A.: Algebra-algorithmic models and methods of parallel programming (in Russian). Academperiodika, Kiev (2007)
3. Andon, P., Doroshenko, A., Zhereb, K.: Programming high-performance parallel computations: formal models and graphics processing units. Cybernetics and Systems Analysis 47, 4, 659–668 (2011)
4. Asanovic, K. et al.: A view of the parallel computing landscape. Commun. ACM 52, 10, 56–67 (2009)
5. Buttari, A., et al.: The impact of multicore on math software. In: Kagstrom, B., Elmroth, E., Dongarra, J., Wasniewski, J. (eds.) PARA 2006. LNCS vol. 4699, pp. 1–10. Springer, Heidelberg (2007)
6. Datta, K., et al.: Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In: ACM/IEEE Conference on Supercomputing (SC '08), pp.1–12. IEEE Press, Piscataway (2008)
7. Doroshenko, A., Shevchenko, R.: A Rewriting Framework for Rule-Based Programming Dynamic Applications. Fundamenta Informaticae 72, 1–3, 95–108 (2006)
8. Doroshenko, A.Yu., Zhereb, K.A., Tyrchak,Yu.M., Khatniuk, A.O.: Creating Efficient Parallel Programs in Fortran Using Rewriting Rules Technique (in Russian). In: International Conference on High-Performance Computations (HPC-UA'2011), pp. 76–83. Kyiv, October 12-14, 2011
9. Doroshenko, A., Zhereb, K., Yatsenko, O.: Formal Facilities for Designing Efficient GPU Programs. In: International Conference on Concurrency Specification and Programming (CS&P'2010), pp. 142–153. Bornicke, Sep. 27–29, 2010

10. Intel Parallel Studio http://software.intel.com/en-us/articles/intel-parallel-studio/
11. Ishihara, M., Honda, H., Sato, M.: Development and Implementation of an Interactive Parallelization Assistance Tool for OpenMP: iPat/OMP. IEICE Transactions on Information and Systems E89-D 2, 399–407 (2006)
12. Khavryutchenko, V.D., Tarasenko, Y.A., Strelko, V.V., Khavryuchenko, O.V., Lisnyak, V.V.: Quantum chemical study of polyaromatic hydrocarbons in high multiplicity states. International Journal of Modern Physics B 21, 26, 4507–4515 (2007)
13. Leijen, D., Schulte, W., Burckhardt, S.: The design of a task parallel library. In: 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '09), pp. 227–242. ACM, New York (2009)
14. OpenMP specification. http://openmp.org/wp/
15. Rus, S., Pennings, M., Rauchwerger, L.: Sensitivity analysis for automatic parallelization on multi-cores. In: 21st Annual International Conference on Supercomputing (ICS '07), pp. 263–273. ACM, New York (2007)
16. Saraswat, V.A., Sarkar, V., von Praun, C.: X10: concurrent programming for modern architectures. In: 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '07), pp. 271. ACM, New York (2007)
17. Tournavitis, G., Wang, Z., Franke, B., O'Boyle, M.F.P.: Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. SIGPLAN Not. 44, 6, 177–187 (2009)