International Journal of Computer Applications (0975 8887)
Volume 86 - No. 9, January 2014

Bridging the Performance Gap between Manual and Automatic Compilers with Intent-based Compilation

Waseem Ahmed
College of Computer Science
King Khalid University, Abha, Saudi Arabia

ABSTRACT

In spite of years of research in automatic parallelization, progress has been slow in terms of tools that can consistently generate scalable, portable and efficient code for multiple architectures. Moreover, a substantial difference in efficiency exists between code generated automatically and code generated by an expert programmer. Although the fact that the best sequential algorithm for a problem can be very different from the best parallel algorithm is well known, the feature of algorithm substitution is absent from most tools available today. However, automatically identifying an algorithm used in code is not trivial considering the nuances in programming style, algorithmic representations and expressions. This paper presents a novel Intent Based Compilation approach that uses a rule-based Expert System Engine to identify the intent of the algorithm used in the code based on fine- and coarse-grained features extracted from code. Using this information, the most optimized algorithm for the target architecture is then substituted. Results obtained by using Amoeba, a framework that incorporates this methodology, on codes obtained from the public domain are presented.

General Terms:

Automatic Parallelization, Parallelization Tools

Keywords:

Intent-based compilation, automation, code-to-code transformers, parallelization, parallel compilers

1. INTRODUCTION

With the increased pervasiveness of massively parallel GPUs, accelerators and FPGAs on general-purpose commodity computers, parallel computing is no longer restricted to elitist machines [1]. The popularity of clusters made with off-the-shelf components, the increasing ratio of number of cores per processor die and the presence of multiple processors on single machines, will have a large impact on the parallel programming community. Parallel programming that was once restricted to the HPC community, will soon involve the mainstream programmers in fields as diverse as Embedded Systems, Browser development, Game Programming and Operating Systems for smart phones, tablets, netbooks and game consoles. The next sub sections highlight the effects of these trends on automatic parallelization.

1.1 Software and Software compiler Requirements

In the past, each generation of hardware brought increased performance for existing applications and a code rewrite was not needed [2]. With the pervasiveness of diverse and specialized computing architectures in HPC, high-end servers, Multi-processor System-on-Chips (MPSoCs), Laptops and mobile platforms, code portability will soon become a major challenge. The responsibility of ensuring scalable, portable and efficient parallel programming for these specialized architectures will rest on the application developers and on the suite of tools available.
The HPC community relies on a large base of legacy sequential code for its scientific computation. Parallelizing such applications for even a single architecture is a complex exercise that incorporates both domain expertise and sophisticated programming skills. In many cases, these applications are executed on various platforms during their lifetime. This makes the task of debugging, testing, porting, maintenance and versioning of code for these applications challenging.

1.2 Automatic versus manual parallelization

In spite of decades of research in parallel development tools (automatic compilers, code-to-code transformers, parallel debuggers, auto tuners and parallel development environments collectively referred to as parallel tools or parallel development environments in the rest of this paper) manual parallelization still continues. One main reason is that the automatically generated code, in the general case, can never be as efficiently optimized for execution on a particular architecture as hand-programmed code [3].
Indeed, the ability of a specialized human programmer to make complex code transformations judiciously by intuition and experience clearly defines the path that future parallel tools should take. This *intelligent* human-factor is seldom incorporated in automatic compilers.

1.3 Loops and algorithms

The choice of algorithms used in a program heavily influences the efficiency of the final application. To illustrate, consider two sequential algorithms A_1 and A_2 that consists of $O(n^3)$ and $O(n^2 logn)$ operations, respectively, available to solve a particular problem. A sequential implementation will prefer the use of the second more efficient algorithm. An automatic parallelization tool will work on the premise that this is the best possible implementation. The ease and the degree of parallelization is not con-

sidered, although parallelizing A_1 may result in a better parallel time complexity. In manual hand-parallelization this is invariably considered.
This paper presents a novel Intent Based Compilation (IBC) methodology that incorporates human intelligence and expertise into the parallel development environment. This research advances the field in the following novel ways

—the use of an Expert System Engine in the parallelization process.
—supports for algorithmic replacements, when needed.

The rest of the paper is organized as follows. The next sections reviews the literature on which this work is based. Section three describes the IBC methodology. Section four explains the structure of the Amoeba framework that is used to implement the IBC methodology. Results are presented in the following section. Section seven concludes the paper.

2. RELATED WORK

The past few decades has seen a lot of research on automatic compilers, auto tuners and code-to-code transformers and the wide performance gap between manual parallelization and that achieved by an automatic compiler has continued to narrow. However, and considering the acute need for such tools in many areas today, a lot of research is further needed. A programming framework is needed to allow the large developer community to develop code for emerging heterogeneous architectures and develop parallel code that is at least as correct as the sequential codes written today [2]. To start with, a look at the shortcomings in such existing frameworks, tools and approaches is essential so that these shortcomings can be minimized or eliminated.
Traditional automatic compilers have continued to focus primarily on loops which, in general, happen to be the performance bottlenecks and the most compute intensive portions of sequential code. To correctly analyze such loops, the tools discourage the use of dynamic aliasing (where many pointers point to the same memory location) and complex and hybrid data types [3]. In addition, iterative versions of algorithms are preferred over recursive functions for easier analysis. Existing code that use recursive functions may need to be rewritten to enable efficient parallelization by automatic tools.
Automatic parallelization tools have continued to place restrictions on the sequential programmer. [2] and [3] have listed such suggestions, but they constrict the creativity of the sequential programmers. Moreover, these restrictions cannot be applied to the existing legacy sequential applications code base popularly used by the HPC and the Embedded System community.
Existing programming paradigms and models like CUDA, PThreads OpenMP, MPI and OpenCL require the programmer to extensively modify existing code with parallel constructs while simultaneously ensuring the correctness of parallel execution. For example, the unstructured nature of PThreads constructs has made the development of correct and maintainable programs difficult [1]. These languages do not provide safety guarantees that significantly reduces opportunities for hard-to-track bugs while improving developer productivity [2]. These features exist in some serial languages like Java and C#. Also, no currently available parallel language, methodology or programming framework can completely ensure a deterministic-by-default result [4]. As there is an increasing trend towards heterogeneous computing, application developers have to contend with multiple, sometimes incompatible programming models (OpenMP, MPI, PThreads, CUDA, OpenCL, etc.) and development environments [2]. As the degree of heterogeneity in architectures increases, the development of software for these hybrid architectures becomes all the more complex. Furthermore, the multilevel deep memory hierarchies in massively threaded GPUs further exacerbate this case.
Autotuners help to an extent in alleviating the problem by generating a set of alternative implementations for blocks of code [5]. But exploring these various combinations to select the best implementation takes time, especially in cases when the search space is not smooth and consists of many local minima or maxima [5]. Also, autotuners and many code-to-code transformers insert tool-specific compiler directives and profiling statements within code. For large evolving codes, this poses an additional burden during program maintenance, evolution and porting.
A solution to this, proposed in [2], suggests using a two-tier software framework that uses domain specific languages at one layer and programmer expertise at the other. This is based on the prediction that programming paradigms in the future will employ domain specific languages at the abstraction level of MATLAB or SQL. While this may be applicable while developing new applications, this is not applicable for the large legacy software set that is available.
Another suggestion is to use highly parallelized and optimized libraries like BLAS, LAPACK, MKL, ATLAS, FFTW and OSKI [6, 7, 8] but the code description of these libraries is need for automatic parallelization when porting to a new hybrid architecture. Moreover, production of these carefully tuned parallel libraries will involve performance coding and domain experts [2].
Many approaches use task graphs, Polyhedral models and variants as intermediate representation to extract parallelism and for dependency analysis. Although reverse engineering tools exist, analysis is difficult for code that has not been structured well and that uses complex data structures. Some automation tools like Cetus [9] use basic parallel transformation techniques like privatization, reduction variable reduction and induction variable substitution. Although these techniques are most relevant, they cannot bring the performance of a badly written code to a hand-optimized level.
In spite of all these advances, the progress is only minimal [10]. The performance of these tools measured with codes from popular benchmark suites does not correctly reflect the advancement in the automation field. Standard benchmarks, like NAS Parallel Benchmarks, PARSEC [11] and SPEC CPU2006, used to test the automation tools are well written and well structured having undergone various code reviews, revisions and public inspections. These benchmarks do not reflect the variations in style and capability of the mainstream and average programmer for which these tools are supposedly being developed. Additionally, these tools may be coded against these benchmarks and optimized specifically for these. The amount of speedup obtained using existing commercially available parallelization tools as compared to what can be potentially obtained with manual parallelization reflects this. Also, to obtain the highest possible performance for an application human intervention, sometimes rewriting code from scratch, will be required [3]. One lesson from years of research in parallel programming is that the complexity of parallel programming should be hidden from the programmer as far as possible [12]. Also code comprehension [13] will need to be made part of automation tools. Two main reasons may be attributed to this. The first is that the algorithm used in the sequential code is assumed to be well suited for the target architecture. A replacement with another more parallelizable algorithm is not considered. This is the case with almost all existing parallelization approaches including [10, 14, 9, 12, 15, 16], to name a few. The second reason is that majority of the automatic parallelizers and compilers convert the sequential program into a

binary executable format. This leaves very little for the parallel developer to work on in case additional parallelization could be manually extracted. The work presented in this paper closely reflects that in [17]. The major difference is that IBC does not require the application developer to learn a new language nor is application developer intervention required before the autotuning stage.
Intent-based compilation (IBC) presented in this paper addresses many of the above mentioned shortcomings.

3. INTENT BASED COMPILATION

Intent-based Compilation (IBC) is a novel approach incorporates human decision making into the compilation process. The human expertise that is involved in manual parallelization of sequential code generates, in most cases, parallel code that has a substantial speedup when compared to parallel code obtained with automatic parallelization.
The approach adopted by most automatic compilers and source-to-source translators is a mechanical identification of nested loops and inserting appropriate pragmas or parallel directives around them. Some tools employ static or dynamic profiling of code to aid this process. However, no attempt to identify the algorithm used in the code is made nor is the replacement of the algorithm with a more suitable one is considered.
IBC, similar to a programming expert in manual parallelization, considers both these issues. Statements, blocks of code or entire algorithms are considered for replacement if better alternatives exist and are available. These replacements of algorithms, in most cases, result in improvements in the time complexity [18, 19, 17].
To correctly emulate the manual process, the stages that a manual hand-coded parallelization passes through are identified. The next subsection explains these steps in the manual parallelization stage. The next subsection explains the manual analysis and transformation process.

3.1 Manual Parallelization

The following questions are considered by an expert programmer before parallelizing the sequential code

(1) Where are the performance bottlenecks?
(2) What problem is this bottleneck trying to solve (or which algorithm is the programmer trying to use)?
(3) About the sequential algorithm itself -
  (a) Can this sequential algorithm be easily parallelized? If not,
  (b) Is a better parallel algorithm already available that can solve this problem? or
  (c) Is there another sequential algorithm that can both be easily parallelized and yield better performance on being parallelized?

The first question can be easily answered by using static or dynamic profiling. The second and third require expertise in the domain. Once definite answers to these are obtained, the programmer then proceeds to parallelize this bottleneck specifically and not every loop block.

3.2 Algorithm Identification in IBC

Algorithms that commonly recur in the fields of HPC and Embedded Systems, referred to as kernels, algorithmic species, dwarfs, motifs or patterns, have been thoroughly studied and classified in literature [20, 21, 22], These classifications are uses as a basis by IBC to identify bottlenecks in code. The identified bottlenecks are at the loop-block level of granularity. The appropriate *slices* [14] are then replaced, if possible, by known versions of more efficient and parallelizable solutions. This is further explained in section 3.4.
Algorithm can be implemented in different ways which makes this step challenging. Consider the code snippets in Listing **??** for dense *matrix-matrix multiplication* obtained from the public domain on the Internet. Listings **??** (a) and (b) follow the naive text book description of the *matrix-matrix multiplication* algorithm taught in elementary programming courses and one that is commonly used. Matrices are defined as two-dimensional arrays of elements. This implementation is easily parallelized using automatic compilers and easily transformed into parallel code using code-to-code translators. The implementation of the algorithm using arrays is simple, encouraged [3] and makes the code readable, portable and maintainable.
Another implementation of the same algorithm is given in Listing **??** (c). This uses a dynamically allocated pointer-to-pointer representation with rows stored in contiguous memory space. To access any particular element, a calculation involving the number of columns and rows is used. For an expert sequential programmer, this may be the preferred way of implementation, although it makes the code less readable, less portable and less maintainable. Since this version uses pointers, it makes automatic parallelization difficult. Most parallel tools discourage the use of pointers and cannot easily parallelize this version [3].
Even when using the same language for implementation, differences in implementations will exist for the same algorithm. For expert and experienced programmers, these patterns in code are easily evident. For example, given any of the implementations of the previous example, the dense matrix-matrix multiplication pattern can easily be recognized. He can then easily parallelize this. This feature of identification of an algorithm based on patterns is built into IBC.
Using this feature, many of the passes made by source-to-source translators and automatic parallelizers can be avoided. For example, IBC avoids the detailed array section analysis and data dependency analysis used by Cetus [9]. Instead, patterns are used to match complete algorithm blocks. Hard-wired heuristics used by traditional compilers [16] are re-framed as flexible Expert System *rules*. This methodology gives IBC the added ability to correctly and easily handle recursive functions, pointers, variations in implementations, and complex and recursive data structures like linked lists and trees - features not commonly found in source-to-source transformers and other automatic parallelization tools.

3.3 Focused Identification

In traditional approaches, *profitable* loops are prioritized for parallelization. To correctly identify them, static and dynamic profiling of code is necessary. In IBC, on the other hand, such loops are directly identified by scanning the code for pattern matches. Features extracted from the code are matched against a collection of patterns of HPC kernels, represented as *rules* in a knowledge base in the *Expert System Engine* (ESE). These rules encapsulate an expert programmers skills and a domain expert's expertise in the ESE. A rule for an *algorithmic pattern* is placed on the agenda only if all the *facts* that satisfy the rule are found in the code. When a *rule* is fired, all *actions* corresponding to the rule on the agenda are executed [23].
This focused approach of IBC safely ignores loop-blocks that do not contribute to the performance bottleneck. For example, loop blocks used in the initialization of matrices and loop blocks that

perform addition operations on vectors and two-dimensional matrices do not significantly affect the performance of the application and can be safely ignored. This feature of IBC substantially reduces the profiling time.

3.4 Loop Block Transformation

In manual parallelization, loop transformations may involve complete algorithmic replacements. To further illustrate this, consider dense two-dimensional matrix-matrix multiplication as an example. Various algorithms with different time complexities exist to achieve this - the naïve three-nested-loop matrix multiplication with a sequential time complexity of $O(n^3)$; Strassen's algorithm with a sequential time complexity of $O(n^{2.81})$; and other improvements to these algorithms with sequential time complexities as low as $O(n^{2.376})$ [18]. Although some of these are complex to implement, the performance improvements obtained for large values of *n* makes the effort worthwhile. Similarly, many parallel algorithms exist along with their sequential counterparts to solve problems, differing widely in implementation complexity, ease of parallelization, and scalability [24, 19, 18, 25, 26]. Additionally, libraries optimized to implement algorithms for particular architectures like BLAS, ATLAS, FFTW and OSKI [6, 7, 8] have also been implemented.
The identification and subsequent 'intelligent' replacement of algorithms, although common practice in manual parallelization, is seldom considered by automatic parallelization tools while parallelizing sequential code. This feature is made available in IBC whereby an entire block of code and the relevant slices if necessary, if necessary, are substituted by a better algorithm.

4. DESCRIPTION OF AMOEBA

The IBC approach has been implemented in Amoeba, an automatic compiler framework. It consists of three main components as shown in Figure 1.

(1) Feature Extractor (FE)
(2) Pattern Identifier (PI)
(3) Code Instrumentor (CI)

The Pattern Identifier and the Code Instrumentor components together comprise the Embedded System Engine (ESE). A detailed explanation of the components is given in the next subsections.

4.1 Feature Extractor

The *Feature Extractor* is the lexical analyzer component in IBC that extracts specific features present in input code. These features in code uniquely and collectively identify a block(s) of code as an algorithmic pattern and are used to distinguish one HPC pattern from another. Features are extracted at five different levels of granularity - at the *function* level, *loop-block* level, *loop* level, the *statement* level and the *variable* level. Features at the *function* level include the number and type of parameters, whether or not the function is recursively called, whether the function uses loops, etc. Features at the *loop-block* level include the nesting depth (number of loops), etc. Features at the *loop* level include details about the loop indices, the values of loop increments, the initial and termination values, nature of the statements within each loop, etc. Features for each *statement* within the loop include the type of expression, type of parameters on the LHS and RHS of the operator, the main operator, sub operators, etc. Features for the *variable* include the dimension of variable, its indices, lower and upper bound, etc. The extracted features in code are stored as facts in a *fact list* for subsequent analysis by the ESE.
The Feature Extractor has been implemented using *flex* and C. The output of the Feature Extractor is a facts list that follows the CLIPS [23] format. For every feature extracted, a *deftemplate fact* is asserted (added) to the *fact* list. As CLIPS is a non-procedural programming language, the sequence of facts in the fact list is not important.

4.2 Pattern Identifier

The *Pattern Identifier* (PI) is part of the Expert System Engine. This component captures the expertise of both the expert programmer and the domain expert as *rules* in the ESE's knowledge base. *Rules* describe algorithms and their variations based on the presence or absence of *facts* in the facts list. They help the ESE to easily locate and identify matches in the fact list. Separate set of rules are needed for each kernel classified in [20, 21]. The input to the *Pattern Identifier* is the *fact list* generated by the *Pattern Extractor*. On execution, a *rule* for a kernel pattern is fired on the presence or absence of *facts* in the *fact* file.
To further illustrate, consider two-dimensional dense matrix-matrix multiplication. The *2D-dense-matrix-matrix-multiplication* rule needs about nine *facts* to be fired. On the presence of all of these nine *facts* in the *facts list*, the rule is activated and a new *fact* asserted into the *facts list* indicating the presence of the *2D-dense-matrix-matrix-multiplication* pattern in code along with its location specific details in code. Additional rules exist to eliminate unneeded and irrelevant facts (for example, statements within a single nested loop), and to concatenate multiple facts into single facts. This substantially prunes the facts list.
These new *facts* in the *facts list* serve as input to the next component of Amoeba.

4.3 Code Instrumentor

The Code Instrumentor (CI) is the third main component of Amoeba. The inputs to the CI are the modified *facts list* from the *Pattern Identifier*, the original input file and the specification of the target architecture. Based on the facts list, the CI is responsible for the following

(1) Correctly locate the loop blocks to be instrumented in the original input file
(2) Identify the pertinent slices [14] related to the loop blocks for replacement
(3) Identify the best existing algorithmic variant for replacement based on the specification of the target architecture (if necessary)
(4) Replace the loop blocks and the slices in the original code with the relevant algorithmic implementation
(5) Judiciously insert compiler directives (OpenMP, MPI, CUDA, OpenCL or a mix) based on the target specification.

Step four is an optional step used only when a better algorithmic implementation is available. If used, the replaced algorithm should have an improved time complexity compared to the one that existed in the original code. Step five ensures that the generated code is suitable for the target architecture. For an architecture that uses an NVidia GPU, the CI should generate code with CUDA-specific statements. For a shared memory system, code with OpenMP directives is to be generated.
The C file generated by the CI can be further optimized using existing autotuners [5, 27, 28].

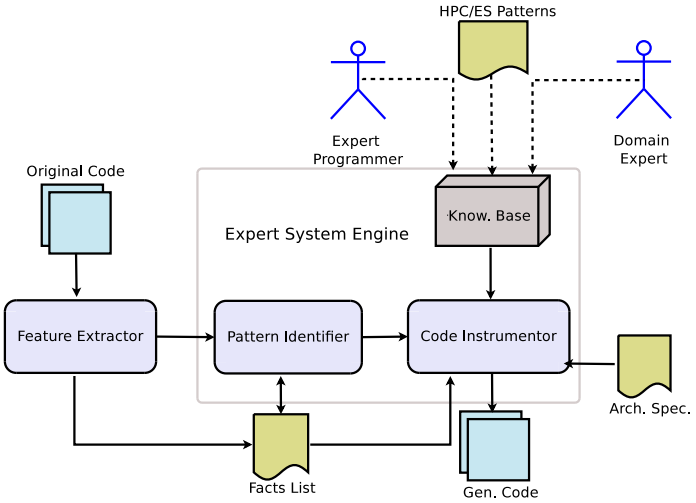

Fig. 1. Components in Amoeba

5. EXPERIMENTS AND RESULTS

A small test input set was used to demonstrate the efficacy of the IBC methodology. As the approach closely models manual parallelization, the results for all other cases should be similar, or nearly similar, to those obtained using manual parallelization.

5.1 Test Input Set

The test input set used in this research consisted of code in C obtained from the public domain. Standard benchmarks were not used for reasons explained earlier in section 2. Code for dense 2D matrix-matrix multiplication (MM) and 1-D and 2-D Jacobi iteration (JI) with contrastingly different implementations were used as the test input set. Only minor cosmetic changes were made to these representations to preserve their originality.

For the MM test set, the first code used the standard text-book implementation. The three matrices ($A = B \times C$) were represented using dynamically allocated two-dimensional arrays, where an element in row x and column y of matrix A was accessed using the form $A[x][y]$ as represented in Listing **??** (a) and (b) . The program consisted of seven loop blocks; three loop blocks (nesting depth = 1) for pointer initialization for rows of each matrix, three separate loop blocks (nesting depth = 2) to initialize the elements of each matrix and a loop block (nesting depth = 3) for the matrix-matrix multiplication. All seven loop blocks were located in the same function.

The second code used the pointers-of-pointers representation to store the matrices, where an element in row x and column y of matrix A was accessed using the form $A[x \cdot N + y]$, as represented in Listing **??** (c), where N indicated the number of rows in matrix A. The code consisted of two loop blocks; one loop block (nesting depth = 1) to initialize the elements of all three matrices and a loop block (nesting depth = 3) for the matrix-matrix multiplication. The matrix multiplication block in this program was defined in a separate function, with all matrices being passed as reference to the function.

The data type of the matrix elements was *double* in both the program codes.

The JI test set consisted of both C and C++ implementations also obtained from the test domain. One code consisted of a 1-D Jacobi iteration and the remaining were 2-D. Some consisted of a fixed number of iterations and the others iterated till an error threshold was satisfied. Similar to the MM test set, the 2-D matrices were represented in some codes using the form $A[x][y]$ and in some as $A[x \cdot N + y]$.

All the codes compiled and executed correctly using the gcc compiler in Linux.

5.2 Experiments

To validate the effectiveness of the presented methodology, a comparison with Cetus [9] was performed.

First, the sequential programs in the test set were compiled as-is and executables obtained (Exec-set #1). Next, the programs were analyzed using the Amoeba framework. The *Pattern Extractor* was used to generate *facts*. These facts, combined with the pre-defined expert *rules,* were loaded into the *Pattern Identifier,* the environment implemented using *CLIPS* version 6.24. The Pattern Identifier correctly identified the two-dimensional matrix-matrix multiplication pattern which activated the *2D-matrix-multiplication* rule and placed it on the agenda. For the *Expertise Injector's* pre-defined expert rule for two-dimensional matrix-matrix multiplication, an OpenMP representation obtained from the public domain was used. Only one loop block was doctored for each of the programs in the test set and OpenMP directives inserted for only the outer loop of the matrix multiplication loop block. The other loops blocks were ignored.

Next, the Cetus GUI was used with the default optimization flags to transform the programs to their respective OpenMP implementations. For both the programs in the test set, Cetus inserted OpenMP directives for each of the loops, twelve for the first and four for the second.

The programs were compiled using *gcc* (SUSE Linux) version 4.5.1 with the -O3 optimization flag and executed on an Intel i7 2160 processor with 4 cores (8 threads) and 4GB L2 shared memory to obtain the other executable sets (Exec-set #2 and Exec-set #3). The programs were executed using all 8 threads with different large matrix sizes. The speedups obtained are shown in Figure 2.

For the JI test set, the Pattern Identifier correctly identified all forms of the Jacobi Iteration present in the test set. No comparisons with

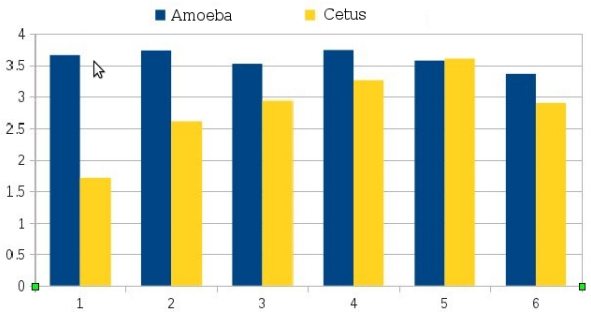

Fig. 2. Comparison of execution times

existing tools were made in this case for reasons cited in the next section.

5.3 Observations

It was observed that the programs obtained using the Amoeba framework performed faster than those obtained using Cetus. As noted in [16], this result was expected as Amoeba's generated programs more closely resembled hand-coded programs than those obtained with Cetus. The speedups in the former case could have been more if rules in the *Expertise Injector* modeled a specialist programmer's optimized transformation instead of those obtained from the public domain.

Also, it was observed that one of the programs transformed using Cetus had run-time errors (error in computed result). The exact reason for the error has not been looked into. As such, the results for this code program were was not used in the speedup comparison.

The amount of speedup obtained clearly depends on the code to be inserted - the better the code, the higher the speedup. This in turn depends on the quality of rules fed into the CI and the replacement blocks that have been made available.

6. CONCLUSION

As stated in [2], few domain experts have the time to develop performance programming skills and few computer scientists have the time to develop domain expertise. This paper attempts to capture both their expertise in a rule-based Expert System Engine. A novel Intent Based Compilation approach has been presented to further advance the field of automatic parallelization. The Amoeba framework, that implements the methodology was used to parallelize codes obtained from the public domain. As expected, performance gains obtained were close to those obtained with manual parallelization.

The Amoeba tool framework, at present, is still evolving and relies on code obtained from the Internet to frame expert rules for different architectures. Specialized rules are needed for optimized algorithmic replacement to further improve the performance of the results. As highlighted in [reference blinded], creating rules for the entire set in [21, 22, 20] and their variations is a dedicated and time consuming exercise. As future work, we plan to incorporate more variations of the patterns in the knowledge base targeting the average mainstream programmers rather than well structured and efficient benchmarks.

7. ACKNOWLEDGEMENTS

This work was supported by a grant from King Khalid University (KKU_S270_33).

8. REFERENCES

[1] Javier Diaz, Camelia Muñoz-Caro, and Alfonso Niño. A survey of parallel programming models and tools in the multi and many-core era. *Parallel and Distributed Systems, IEEE Transactions on*, 23(8), 2012.

[2] Bryan Catanzaro, Armando Fox, Kurt Keutzer, David Patterson, Bor-Yiing Su, Marc Snir, Kunle Olukotun, Pat Hanrahan, and Hassan Chafi. Ubiquitous parallel computing from Berkeley, Illinois, and Stanford. *Micro, IEEE*, 30(2), 2010.

[3] Mehdi Amini, Ronan Keryell, Beatrice Creusillet, Corinne Ancourt, and FranÃ§ois Irigoin. Program sequentially, carefully, and benefit from compiler advances for parallel heterogeneous computing. Technical report, MINES-ParisTech CRI, 2012.

[4] John Cavazos. Intelligent compilers. In *in Proceedings of the IEEE Int. Conf. on Cluster Computing*, 2008.

[5] Ananta Tiwari, Chun Chen, Jacqueline Chame, Mary Hall, and Jeffrey K Hollingsworth. A scalable auto-tuning framework for compiler optimization. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, 2009.

[6] M. Frigo. A fast Fourier transform compiler. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, May 1999.

[7] R. C. Whaley and Jack Dongarra. Automatically tuned linear algebra software. In *Proceedings of Supercomputing*, Nov. 1998.

[8] R. Vuduc, , J. W. Demmel, and K. A. Yelick. Oski: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series*, 16:521–530, 2005.

[9] Chirag Dave, Hansang Bae, Seung-Jai Min, Seyong Lee, Rudolf Eigenmann, and Smauel Midkiff. Cetus: A source-to-source compiler infrastructure for multicores. *IEEE Computer*, December 2009.

[10] Matthew J. Bridges, Neil Vachharajani, Yun Zhang, Thomas Jablin, and David I. August. Revisiting the sequential progamming model for the multicore era. *IEEE Micro*, January/February 2008.

[11] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of PACT'08*, 2008.

[12] J. Ceng, J. Castrillon, W. Sheng, H. Scharwachter, R. Leupers, G. Ascheid, H Meyr, T. Isshiki, and H. Kunieda. MAPS : An integrated framework for MPSoC application parallelization. In *in Proceedings of the Design Automation Conference (DAC)*, 2008.

[13] Anja Guzzi, Lile Hattori, Michele Lanza, Martin Pinzger, and Arie van Deursen. Collective code bookmarks for program comprehension. In *Proceedings of the IEEE 19th Int. Conf. on Program Comprehension (ICPC)*, 2011.

[14] Shih-Wei Liao. *Suif Explorer: An Interactive and Interprocedural Parallelizer*. PhD thesis, Stanford, 2000.

[15] Wen-mei Hwu, Shane Ryoo, Sain-Zee Ueng, John H. Kelm, Isaac Gelado, Sam S. Stone, Robert E. Kidd, Sara S. Baghsorkhi, Aqeel A. Mahesri, Stephanie C. Tsao, Nacho Navarro, Steve S. Lumetta, Matthew I. Frank, and Sanjay J. Patel. Implicitly parallel programming models for thousand-core microprocessors. In *DAC '07: Proceedings of the 44th annual conference on Design automation*, pages 754–759, New York, NY, USA, 2007. ACM.

[16] Georgios Tournavitis, Zheng Wang, Björn Franke, and Michael FP O'Boyle. Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. In *Proceedings of PLDI'09*, volume 44, pages 177–187, 2009.

[17] Jason Ansel. Petabricks: a language and compiler for algorithmic choice. Master's thesis, MIT, 2009.

[18] Barry Wilkinson and Michael Allen. *Parallel Programming - Techniques and Applications using Networked Workstations and Parallel Computers*. Pearson Education, 2 edition, 2005.

[19] Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP*. Tata McGraw-Hill, 2003.

[20] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[21] Pieter Custers. Algorithmic species: Classifying program code for parallel computing. Master's thesis, Eindhoven University of Technology, 2012.

[22] Cedric Nugteren, Pieter Custers, and Henk Corporaal. Algorithmic species: A classification of affine loop nests for parallel programming. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9, 2013.

[23] Joseph C. Giarratano and Gary D. Riley. *Expert Systems - Principles and Programming*. Thomson, 2005.

[24] Ian Foster. *Designing and Building Parallel Programs: Concepts and tools for parallel software engineering*. Reading, MA: Addison-Wesley, 1995.

[25] S. G. Akl. *Parallel Sorting Algorithms*. Orlando FL: Academic Press, 1985.

[26] W. J. Camp, S. J. Plimpton, B. A. Hendrikson, and R. W. Leland. Massively parallel methods for engineering and science problems. *Communications of the ACM*, 37(4):30–41, 1994.

[27] Chirag Dave and Rudolf Eigenmann. Automatically tuning parallel and parallelized programs. *Languages and Compilers for Parallel Computing*, pages 126–139, 2010.

[28] Dheya Mustafa and Rudolf Eigenmann. Portable section-level tuning of compiler parallelized applications. In *Proceedings of the 2012 ACM/IEEE Conference on Supercomputing. IEEE Press*, 2012.