Portable Parallel Programs using Architecture-aware Libraries

Fadi Zaraket Mohamad Noureddine Mohamed Sabra Ameen Jaber

American University of Beirut

{fz11,man17,mns28,aaj15}@aub.edu.lb

Abstract

Programs written for an architecture with n processors require a re-write when migrated to an m processor architecture $\{(m > n)\}$ to benefit from additional resources. Compiler based solutions do not match manual optimizations. Annotation based and api-based solutions such as the OpenMP and the Intel Array Building Blocks work well with data parallel programs and do not scale well with branching programs. We present Portable Parallel Programming (TripleP), a parallel programming methodology that is composed of a declarative programming language, a set of libraries of data structures and algorithms optimized per architecture, and a synthesizer. We evaluated TripleP with the computation of array median, and breadth first traversal of a graph. Our results show that TripleP enables portable programs that benefit from additional resources across architectures with near optimal performance gains.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming - Parallel programming; D.3.2 [*Programming Languages*]: Language Classifications - Concurrent, distributed, and parallel languages

General Terms Algorithms, Intelligence

Keywords Declarative language, software synthesis, concurrency

1. Introduction

Sequential software programs do not benefit from advances in multi-processor and multi-core architectures. With the emergence of commodity multi-processor and multi-core architectures, this problem gained more attention form the research community. Sequential programs are strictly ordered sequences of operations. Parallel programs specify a partial order between operations and thus can execute several operations at once. Compiler based parallelization techniques try to automatically find and use partial orders in sequential code [17]. Compiler based techniques fail often to match manual optimizations in terms of performance [13]. API based techniques such as POSIX [4] require the programmer to specify the partial order between program operations in terms of constructs such as threads, locks, and semaphores. Annotation based API techniques such as OpenMP [5] require the programmer to specify code segments amenable to parallelization with annotations that can be ignored when compiling and running in a sequential mode. Other techniques such as CUDA [10] and OpenCL [15] require user knowledge of the target computational platform.

Synthesis based parallelization techniques for signal processing [8, 9] and linear algebra [3] as well as content-based parallel data structures [2, 7] emerged. We discuss and compare against these techniques in section 3.

A declarative program expresses the logical function only rather than the number of processors, locks, and semaphores. Declarative programming languages are amenable to automatic parallelization [16].

We propose *portable parallel programming* (TripleP), a programming methodology that uses declarative programming to enable automatic parallelization, a comprehensive architecture-aware library of data structures and algorithms, a library-aware synthesizer, and a multi-threaded task execution framework to automate parallelization. For each parallel architecture $\mathcal{A}$, TripleP provides a library of data structures and algorithms optimized for $\mathcal{A}$.

TripleP takes as input a declarative program $\mathcal{P}$ and passes a directed acyclic parse graph (DAG) of $\mathcal{P}$ to the synthesizer. For each construct in the parse DAG, the synthesizer instantiates an optimized library component as a task in the multi-threaded task execution framework.

The TripleP declarative language is intuitively parallel. The user does not need to write code in two modes, one sequential and one parallel. This promotes a culture of parallel data structures and algorithms. The TripleP programs are portable across supported architectures and run with near optimal performance gains on more parallel architectures without the need to be modified or rewritten.

2. TripleP

A TripleP program is a list of statements. A statement can be either a type declaration, a variable declaration, an algorithm or an expression. TripleP has one *scalar* primitive type. The keywords *sequence, set, relation, function* express user-defined ordered set of elements, unordered set, many to many property from one type to another and a many to one property from one type to another, respectively.

TripleP supports addition, subtraction, multiplication and division as algorithms. It also supports set membership, enumeration, Cartesian product, exponentiation, union, intersection and subtraction algorithms.

An expression can either be an initialization expression, an iterator expression, an equality expression, or a conditional expression. Iterator expressions iterate over container elements where a condition holds. Conditional and equality expressions express logic conditions on variables.

Listing 2 describes a weighted graph $G = \langle V, E, W \rangle$ where V is a set of vertices, $E \subseteq V \times V$ is a set of edges, and $W : E \mapsto \mathbb{Z}$ is a weight function. It also visits the vertices in G with a breadth first traversal algorithm. TripleP allows connections between objects. For example, the `vertices` parameter of G is connected to '`v`'.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SAC'12 March 26-30, 2012, Riva del Garda, Italy.

Copyright © 2012 ACM 978-1-4503-0857-1/12/03...$10.00

Algorithm 1 Declarative Language Example

```
DeclareType Set of Integer Vertices;
DeclareType Relation from Vertices to Vertices Edges;
DeclareType Function from Edges to Integer Weights;
DeclareTye Sequence of vertices, edges, weights Graph;

Vertices V;
Edges E from V to V;
Weights W from E to Integer;

Graph G (vertices = V, edges = E, weights = W);
TraverseBF B (graph = G, visitor = Default);
```

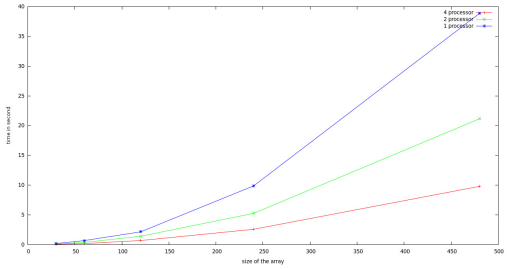

Figure 1. Median and average execution time

Run time framework. TPTask is the multi-threaded task-based run time framework implemented using POSIX [4]. TripleP algorithms and data structures use TPTask internally to perform their work. The TripleP end user can not interface with TPTask.

A pool of worker threads monitors a task priority queue. A worker thread removes a task from the queue and runs it. When done it reports to its parent. A task may be placed back on the queue in case it needs data from other tasks.

Synthesizer. The TripleP synthesizer (TPSyn) traverses a parse graph of a declarative program recursively. For each node in the parse graph, TPSyn selects the corresponding algorithms, data structures, and adequate parameters for the architecture at hand from the TripleP library of optimized components. TPSyn deploys these components as tasks in TPTask. TPSyn also picks parameters for the task such as the number of preallocated threads, the maximum number of threads, the size of preallocated memory, and the memory allocation mechanism.

3. Related Work

Compiler based techniques reorder sequential into parallel code by exploiting dependence relations between operations [17],they fail to match manual optimizations in terms of performance [13].

API based techniques, such as POSIX, MPI and OpenCL, require the programmer to explicitly specify the order of operations and use architecture specific features [4, 15].

Annotation based techniques, such as OpenMP, work well with data parallel sections of code. They do not scale well with branching and dependent code [5].

The Array Building Blocks (ArBB) library from Intel supports parallel operations on scalar containers [7]. The ArBB just in time compiler allows dynamic transformations of sequential code into parallel code on parallel entry points defined by the user. TripleP differs in that it uses synthesis at compile time to generate parallel binaries from declarative programs, and uses a dynamic task-based multi-threaded run time framework. TripleP takes as input declarative code that is explicitly parallel, while ArBB requires the user to write sequential code with sophisticated annotations and partially parallel code.

Hierarchical tiled arrays (HTA) provides data structures that interpret tiles as array objects and abstracts the locality and parallelization potential of array operations [2]. HTA provides data parallelism through HTA tiles in the context of single threaded programs. TripleP differs in that it uses a declarative rather than a sequential imperative language, and provides data and control parallelism through its synthesis and task-based multi-threaded runtime framework.

SPIRAL is a platform that generates portable optimal performance programs for linear digital signal processing (DSP) computations [8]. It uses search and machine learning techniques based on the structure of the algorithm and the implementation to find the best matching algorithmic and implementation choices for a specific architecture. FFTW [9] and ATLAS [3] target discrete fast Fourier transforms and linear algebra computations, respectively. They use empirical learning to select from a library of tunable algorithms.

TripleP is similar to SPIRAL, FFTW and ATLAS in that it selects the algorithm and the implementation that best fits the subject architecture, via knowledge based rules and manual effort. TripleP works across several problems and is not limited to linear DSP and algebra. TripleP can use the choices made by SPIRAL, FFTW, and ATLAS for a specific architecture wherever these choices prove to be better.

Existing parallelization techniques that extract parallel programs from declarative programs use automatic program transformations that do not scale well to extract efficient parallel programs, and may in turn require scheduling annotations [16]. TripleP differs in that its synthesis approach is library based. TripleP also tunes the parallelism and memory allocation mechanisms at run time.

4. Results

The current version of TripleP supports four architectures with shared memory with 4, 8, 16, and 32 processing units each respectively. We had access to these architectures through the Intel Many-core Testing Lab (MTL) [1].

The TripleP library supports parallel vector and matrix arithmetic operations, dense and sparse graphs, graph depth first and breadth first traversal. We picked the data structures and algorithms from different sources such as textbooks,articles [6, 11, 18], and existing C++ libraries [14].

Median and average Figure 1 shows the results of running two user tasks that compute the median and the average of a sequence of scalars using TripleP on three different architectures with one, two, and four cores respectively simulated using the Metropolis framework [12]. The results show the execution time taken by the simulation on each of the architectural models, versus the size of the input data.

Breadth first search The charts in Figure 2 show the running time for the breadth first traversal of dense graphs with 80K, 100K, and 200K vertices. The TripleP implementation ran multiple breadth first traversals and merged the results at the end of each traversal. TripleP achieved near optimal gains across architectures except for the 32 core because of the dense nature of the graphs.

Comparison with ArBB The TripleP median selection marginally outperformed the ArBB median selection algorithm. We implemented the breadth first traversal and the minimum spanning tree computations with the ArBB library. We could not produce running time results to compare against because the ArBB run-time library produced faulty code for the branching parts of the BF algorithm. We reported the issue to Intel ArBB engineers.

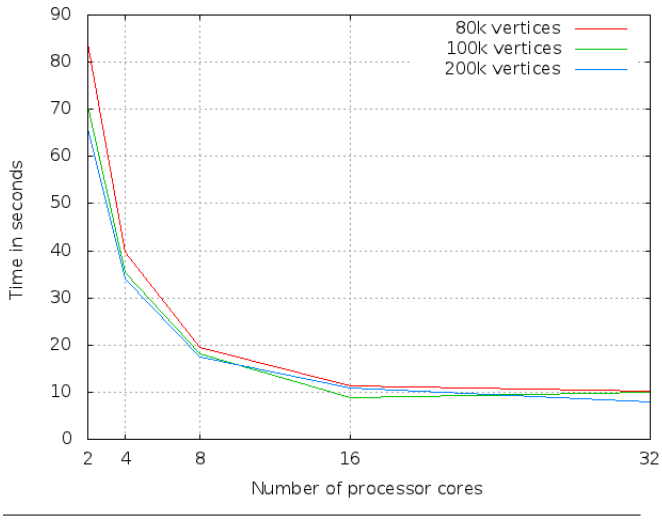

Figure 2. Breadth first traversal results

Acknowledgment This work was made possible by a grant from the University Research Board at the American University of Beirut.

5. Conclusion

We presented TripleP, a Parallel Programming Platform composed of a declarative language, a multi-threaded task-based run time framework, a set of libraries of parallel algorithms and data structures, and a synthesizer. The TripleP declarative language abstracts the execution order of the program away from the developer and allows for explicit parallelism without requiring architecture specific annotations and structures. We evaluated TripleP using selection, and graph traversal algorithms. In the future, we plan to extend the TripleP library to include more graph computations, and regular expressions.

References

[1] Intel manycore testing lab. http://software.intel.com/en-us/forums/intel-manycore-testing-lab.

[2] *Programming for parallelism and locality with hierarchically tiled arrays*. ACM, 2006.

[3] C. W. Antoine, A. Petitet, and J. J. Dongarra. Automated empirical optimization of software and the atlas project. *Parallel Computing*, 27:2001, 2000.

[4] B. Barney. POSIX Threads Programming, 2010.

[5] N. Copty. Introducing OpenMP: A Portable, Parallel Programming API for Shared Memory Multiprocessors. *Sun Studio Technical Articles*, 2010.

[6] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 3rd edition, 2009.

[7] A. G. et al. Array Building Blocks: A Flexible Parallel Programming Model for Multicore and Many-Core Architectures. *Dr. Dobb's Journal*, 2010.

[8] M. P. et al. Spiral: A generator for platform-adapted libraries of signal processing algorithms. *Journal of High Performance Computing and Applications*, 18:21–45, 2004.

[9] M. Frigo and S. G. Johnson. Fftw: An adaptive software architecture for the fft. In *ICASSP*, pages 1381–1384. IEEE, 1998.

[10] D. Luebke. CUDA: Scalable parallel programming for high-performance scientific computing. In *Biomedical Imaging: From Nano to Macro*. IEEE, 2008.

[11] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 1997.

[12] N/A. Metropolis: Design Environment for Heterogeneous Systems. http://embedded.eecs.berkeley.edu/metropolis/, 2004.

[13] G. Ren, P. Wu, and D. Padua. An empirical study on the vectorization of multimedia applications for multimedia extensions. In *International Parallel and Distributed Processing Symposium*, volume 01 of *IPDPS*, pages 89.2–, Washington, DC, USA, 2005. IEEE Computer Society.

[14] J. G. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost Graph Library*. Addison-Wesley Professional, Dec 2001.

[15] J. Stone, D. Gohara, and G. Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in Science & Engineering*, 12:66, 2010.

[16] E. Trichina. Derivation of explicitly parallel code from declarative program by transformations. In *Perspectives of System Informatics*, volume 1181, pages 178–190. Springer Berlin / Heidelberg, 1996.

[17] M. Wolfe. Parallelizing compilers. *ACM Computing Surveys*, 28:261–262, March 1996.

[18] C. Xavier and S. S. Iyengar. *Introduction to parallel algorithms*. John Wiley & Sons, Inc., New York, NY, USA, 1998.