Evaluating the Separation of Algorithm and Implementation within Existing Programming Models

Michelle Mills Strout[1], Christopher Krieger[1], Andrew Stone[1], Christopher Wilcox[1], John Dennis[2], James Bieman[1]

[1]Colorado State University, Department of Computer Science, 1873 Campus Delivery, Fort Collins, CO 80523-1873, USA
[2]National Center For Atmospheric Research, 3090 Center Green Drive, Boulder, CO 80301, USA

Email: mstrout@cs.colostate.edu

Abstract. Implementation details and performance tuning obfuscate the algorithms in programs. In the SAIMI project (Separating Algorithm and Implementation via Programming Model Injection), we are investigating ways to use simpler, more restricted programming models to orthogonally express important subcomputations and express implementation details as transformations on the subcomputations. In this initial phase of the project, we are evaluating separation in existing programming models especially with respect to look-up table, grid-based, sparse computation, and task graph programming models. In this paper, we illustrate the general approach of the SAIMI project with a source-to-source compilation tool called Mesa that automates most aspects of applying look-up transformations in scientific codes; we use the CGPOP miniapp as an example geoscience code to discuss the coupling between the discretization grid implementation details and the communication routines that support the simulation algorithms; and we evaluate the ability of existing programming languages such as Cilk, TBB, CnC, and OpenMP to represent general, dynamic task graphs that arise in sparse computations. We conclude with future directions for developing orthogonal implementation abstractions and building on the orthogonal capabilities in existing programming languages.

1 Introduction

Solving important scientific problems through computational modeling requires high performance computing. High performance computing requires performance on individual processors and parallel programming to take advantage of the multitude of parallel computing architectures. Using general-purpose parallel programming models is error prone and results in code where implementation details obfuscate the original algorithm. Additionally implementation decisions necessary for one target platform are not necessarily applicable to future platforms, therefore requiring significant development costs when porting applications to new systems.

There are a range of ways in which implementation details are exposed in programming models. Declarative programming languages seek to avoid the exposure of implementation details. This approach avoids obfuscation due to parallelization implementation details, but does not provide a mechanism for the application programmer or performance programmer to take control of the implementation. Explicit parallel programming in general purpose programming models like C or Fortran with MPI provide almost complete control over implementation details such as the distribution and scheduling of data and computation, but the implementation is

strongly coupled with the algorithm. This control and the use of the MPI interface results in good performance portability, but the entangling of the algorithmic and implementation specifications can lead to significant code obfuscation.

Recent programming model research and development includes programming languages and libraries with constructs that enable the orthogonal specification of algorithms and implementation details. Examples include OpenMP [14], which provides pragmas for labeling a loop/algorithm as having `forall` parallelism in and indicating implementation preferences; Parallel Global Address Space (PGAS) languages such as Chapel [12, 13], which provide user-defined distributions; and Standard Templates Adaptive Parallel Library (STAPL) [36, 5], which provide orthogonal scheduling of data structure iterators. There are also more restrictive programming models such as the polyhedral programming model [48, 17, 39, 25, 19, 24, 9], which enables orthogonal specification of scheduling and storage mapping within the compiler, and MapReduce [15], which enables implementation details, such as the runtime environment, to be specified and to evolve at runtime. Wholesale conversion of existing codes to new models is problematic due to the magnitude of the work involved and to an initial lack of tool support. Conversion to programming models that are minimally intrusive like OpenMP is more feasible, but the interface between algorithm and implementation specification in such models lacks power in terms of both the algorithmic abstractions and the implementation abstractions.

In the SAIMI project, we use injectable programming models to orthogonalize programming implementation decisions such as performance transformations and raise the abstraction level for implementation specifications. An injectable programming model is a more restricted, declarative programming model made available within the context of a more general programming model such as C/C++ or Fortran. We are initially making the more restricted programming models available through pragmas and/or library interfaces followed by source-to-source transformation. We are developing injectable programming models for specifying performance critical algorithms and optimizations such as lookup tables, stencil computations, dense and sparse matrix computations, and task graph computations. We envision high-level, orthogonal implementation abstractions that will enable application and performance programmers to specify the implementation for each algorithm differently for each target architecture.

The SAIMI project research goals include (1) evaluating existing mechanisms for orthogonally specifying implementation details, (2) raising the level of abstraction for implementation specification in existing mechanisms for a set of injectable programming models, and (3) developing libraries and preprocessors capable of composing orthogonal algorithm and implementation specifications. We evaluate the programmability and performance of our approach on performance-critical computations in various small, medium, and large scientific computing benchmarks written in C++ and Fortran.

Some defining characteristics of the SAIMI project are the incremental approach embodied by injectable programming models and the concept of providing abstractions for the high-level, orthogonal specification of implementation details. This paper presents three injectable programming models at various stages of evaluation and development. As a demonstration of the SAIMI concept of injectable programming models, Section 2 presents an injectable programming model for the lookup table optimization that is being used to simulate the forward problem in an interdisciplinary project for analyzing Small Angle X-ray Scattering (SAXS) data. Section 3 overviews a geosciences miniapp called CGPOP, which we have developed as a testbed for evaluating the separation of algorithms from the details of discretization. In Section 4, we evaluate existing programming model constructs for specifying and executing arbitrary task graphs that occur in sparse matrix computations.

Table 1: Saxs performance and error statistics.
Relative error is shown for SAXS, absolute error for slope aspect.

	SAXS Continuous	**SAXS Discrete**	**Slope Aspect**
Original Time	.434s	170s	234ns
Optimized Time	.208s	24s	53ns
Performance Speedup	2.1X	7.1X	4.6X
Maximum Error	$1.1X10^{-3}\%$	$4.0X10^{-3}\%$	$3.2X10^{-5}$
Average Error	$7.8X10^{-5}\%$	$8.2X10^{-4}\%$	$1.0X10^{-6}$
Memory Usage	3MB	4MB	800KB

2 The SAIMI Approach (The Lookup Table Example)

The first injectable programming model developed as part of the SAIMI project addresses mathematical expressions. The implementation details are those associated with applying a lookup table (LUT) optimization. LUT optimization is applicable when one or more computationally expensive expressions are evaluated a significant number of times with a restricted input domain and the application can experience some approximation error and still be considered correct. LUT optimization can reduce the computational load of scientific applications, but it has generally been incorporated into source code by hand due to a lack of support methodology and tools. Such manual tuning lowers programmer productivity, results in no or only ad hoc approximation error analysis, and can obfuscate code, impacting maintainability and adaptability. Our source-to-source compilation tool called Mesa automates the tedious and error-prone aspects of the LUT optimization process [47] and helps the programmer guide the critical tradeoff between application performance and application approximation error.

Figure 1 shows a comprehensive methodology for LUT optimization. Mesa automates the steps in the methodology marked with an asterisk. In Step 1, programmers use typical performance profiling to identify expressions in loops that may be candidates for LUT optimization, and then the programmer marks such expressions with a pragma as can be seen in Figure 2a. Mesa automates domain profiling (Step 2) by instrumenting a version of the application to capture domain boundaries. LUT size selection (Step 3) is specified on the command line as illustrated in Figure 2b, but Mesa provides detailed error analysis (Step 4) so that the programmer can try various table sizes and perform error analyses until the LUT optimization meets the application requirements. For the small angle x-ray discrete scattering application (SAXS discrete), Figures 3a and 3b show the tradeoff between application approximation error and application performance that can currently be derived by running Mesa with various table sizes and options (an example option is whether to interpolate between table entries or not). Mesa reduces development time by automating code generation and integration (Steps 5 and 6). The comparison between the original and optimized code (Step 7) is currently manual.

We evaluated Mesa on three scientific applications developed at Colorado State University. Our results include a 7.1× speedup for a SAXS discrete scattering application, 2.1× speedup for SAXS continuous scattering, and 4.6× speedup for a slope aspect calculation. We find that Mesa enables LUT optimization with less effort than manual approaches, while tightly controlling error. Table 1 shows the performance speedup for each application, with the maximum and average errors introduced by the optimization. Performance was measured on an Intel Core 2 Duo E8300 running at 2.83GHz, with a 6MB L2 cache.

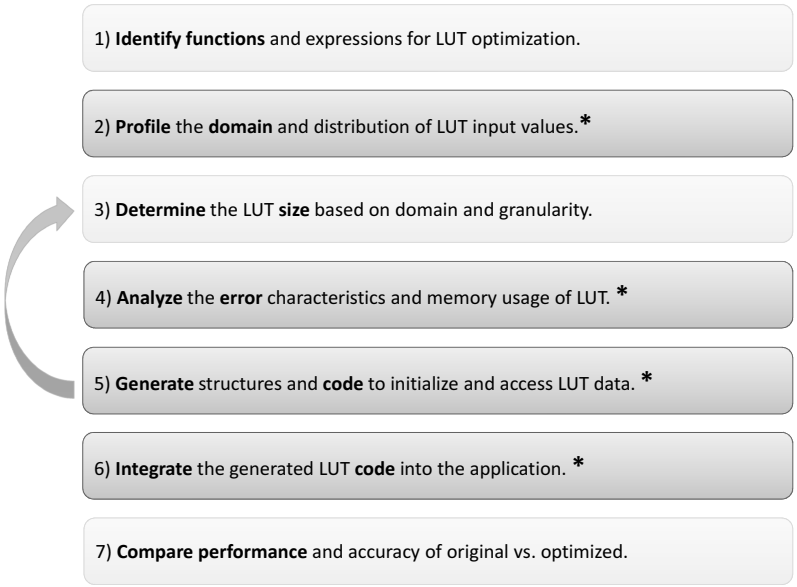


Figure 1: Methodology for LUT optimization. Mesa automates the steps marked with an asterisk.

```
// Iterate atoms (outer loop)
for (U32 atom1=0; atom1<vecAtoms.size(); ...
{
  // Iterate atoms (middle loop)
  for (U32 atom2=atom1; ...
  {
    // Loop on steps (inner loop)
    for (U32 step=0; step<uSteps; ++step)
    {
      // Combine parameters to scatter
      float rTheta = fDistance * fTheta;

      // Optimize subexpression shown below
      #pragma LUTOPTIMIZE
      fIntermediate =
          sin(4*rtheta) / (4*rtheta);
    }
  }
}
```

(a)

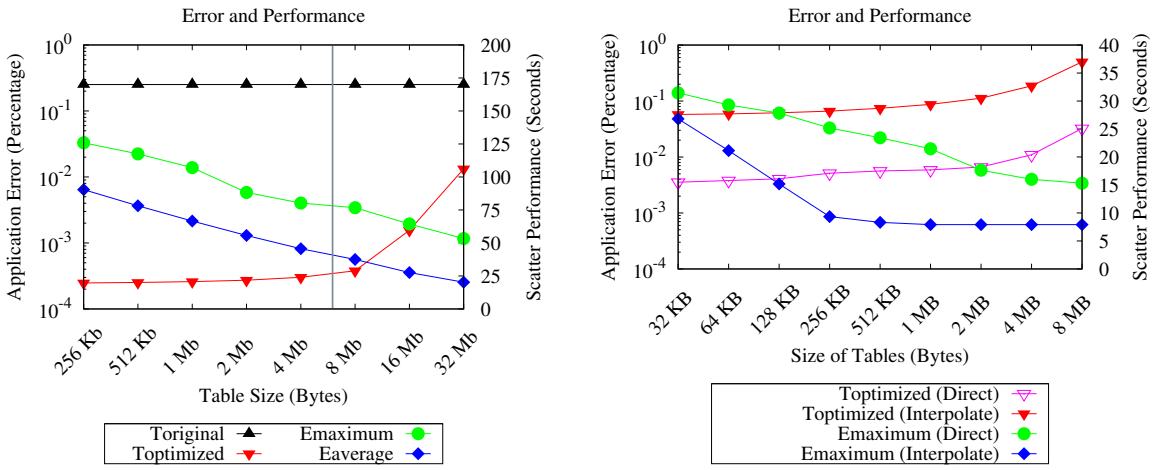
```
./Mesa ScatterOriginal.cpp
    ScatterOptimized.cpp
    -exhaustive -pragma -lutsize 100000

Mesa LUT optimization started
Variable: rTheta
Lower Bound: 2.000000e-02
Upper Bound: 2.000000e+01
Granularity: 1.998000e-04
Lut size (lut): 100000
Error analysis: exhaustive
Emax: 5.476423e-04, Eavg: 1.884966e-05
Mesa LUT optimization completed
```

(b)

Figure 2: **(a)** Indicating a candidate expression for LUT optimization with a pragma. **(b)** An example of executing Mesa from the command line.

The LUT optimization improves single-core performance by navigating a tradeoff between memory usage (the size of the lookup table) and execution time of expression evaluation. As can be seen by inspecting the Toptimized data in Figure 3, on a single core the LUT optimization is beneficial while the lookup table fits within mid-level cache, whose size of 6MB is indicated



(a) (b)

Figure 3: **(a)** Performance and accuracy (SAXS discrete). **(b)** Performance and accuracy of linear interpolation (SAXS discrete).

by the black vertical line. We have also been investigating this space versus time tradeoff in the context of OpenMP parallelized versions of the SAXS discrete and continuous scattering applications. Figure 4 shows that our optimizations scale well in the multi-core environment. We also find that the optimized version scales at least as well as the unoptimized code on up to 24 cores on a node of a Cray XT6m. Future work includes quantifying the tradeoff between parallelism and space more precisely.

LUT optimization is motivated by the observation that a number of applications evaluate elementary functions repeatedly with inputs in a restricted domain. Hardware designers have long tried to accelerate elementary functions with LUT memory and associated circuitry, as done by Gal [18]. Frequently cited papers by Tang [45, 46] apply LUT methods to approximate elementary functions using IEEE math. There are few academic references on software LUTs, but some books [37] and articles encourage the use of *ad hoc* techniques. A series of papers on LUT hardware for function evaluation in FPGAs [40, 16] has led to a crossover paper by Zhang et al. [49] that discusses hardware and software LUT methods. Zhang et al. describe a compiler that transforms functions written in a MATLAB-like language into C/C++ code suitable for multi-core execution. Mesa performs a similar transformation in the context of C/C++ source code using the ROSE [38] compiler infrastructure thus allowing optimization of entire applications.

3 A Testbed for Evaluating Injectable Programming Models

Because programming models are crucial in supporting the programming of parallel machines, the programming languages community should evaluate programming models along the productivity and performance dimensions. Recently, we proposed and published some qualitative criteria for evaluating the programmability of programming model features with respect to the extent to which they enable the orthogonal specification of implementation details [26, 27].

To encourage the separation of algorithm and implementation details, we proposed programmer control and tangling as two qualitative measures for evaluating programming model

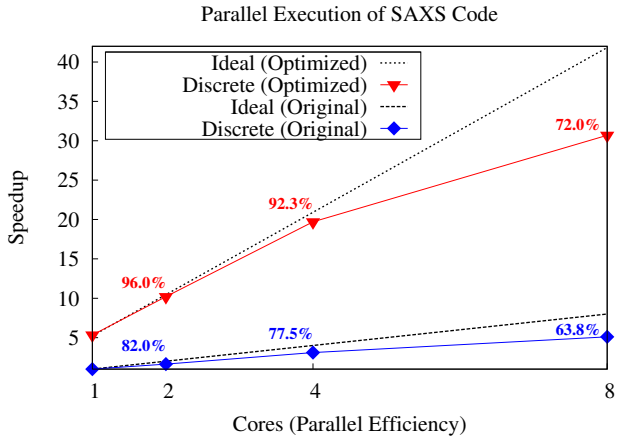


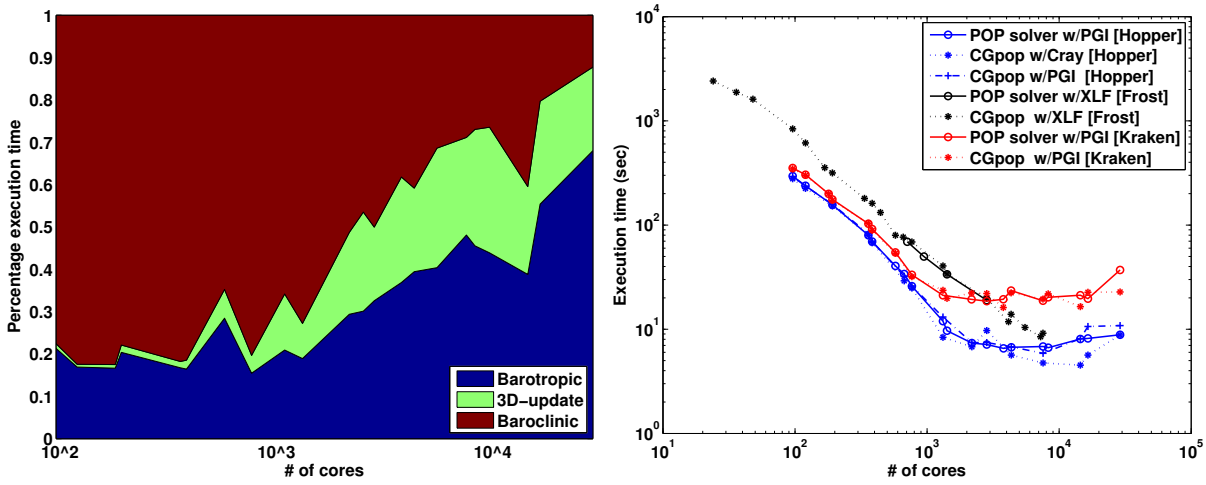
Figure 4: Performance of parallel execution (SAXS applications).

constructs. Because parallel programming implementation details often obfuscate the original algorithm and make later algorithm modifications and maintenance difficult, we believe that parallel programming models should provide features that enable separation between algorithm and implementation detail specifications. A similar idea was suggested in the Parallel View from Berkeley paper [6], which suggests that programming models provide a separate interface for application programmers and parallel implementors.

In addition to the more general qualitative measures of programmer control and tangling, we have also developed a testbed miniapp specific to geoscience. In geoscience applications linear algebra kernels are commonly used to find solutions to partial differential equations. The data operated on is typically structured using a grid. However, despite the prevalence and importance of these kernels, this code is often obfuscated by details specific to the architecture and grid used. This obfuscation makes the code difficult to maintain, port, and makes it hard to try possible performance optimizations and/or data structure modifications. Since many high-performance applications are maintained over several years and ported across several different architectures, the cost associated with code maintenance can be large. The underlying grid is also modified as scientists and mathematicians find new ways to model the earth. Therefore, one ongoing research project within SAIMI is investigating library interfaces for specifying algorithms orthogonally from the grid discretization and then generating specialized computation and communication code for specific grids.

The Parallel Ocean Program (POP) is an example of an application that uses more than one grid discretization. The Parallel Ocean Program (POP) [23] was developed at Los Alamos National Laboratory and is an important multi-agency code used for global ocean modeling and is a component within the Community Earth System Model (CESM) [1]. Between versions LANL POP 2.0 and NCAR POP 2.1, POP was modified to include support for the tripole grid as well as the dipole grid. This modification required rewriting large sections of code because the new boundary conditions seen in the tripole grid affects how the stencil computation is applied to the grid as well as what data is passed during communication.

POP's developers would like to experiment with other types of grids as well as with new



(a) (b)

Figure 5: **(a)** Relative cost of the different components of the POP 0.1° benchmark on Kraken: a Cray XT5. **(b)** Execution time in seconds for 1 day of the barotropic component of the POP 0.1° benchmark and the MPI-2S version of the CGPOP miniapp.

algorithms, data-structures, and programming models. However, experimentation is difficult within the context of the full application. Thus it is useful to have a smaller application that can be modified and serve as a proxy for examining the performance and productivity impact that changes would have on the larger application. Such an application enables POP's developers to ensure that performance portability of the most critical portion of the application is maintained while also testing new algorithms, data-structures, and programming models.

Benchmarks have historically served the role of acting as smaller applications that can be modified to examine the possible effects of changes on a larger application. Benchmarks are small to medium-sized programs that model common performance bottlenecks in existing computational simulations. Commonly used benchmark packages include the HPCS Challenge suite [30], NPB [8], and High-Performance Linpack [34]. Although these benchmarks are small enough that researchers can quickly modify them or reimplement them in a new programming model, their simplicity comes at the cost of excluding implementation details that have a significant programmability impact within the larger applications of interest. These implementation details may impact the performance characteristics of the application or impact how easily a new technique can be integrated into a larger application. As such, a technique may be evaluated as improving performance or productivity within the benchmark, but may not actually be as productive or have the same performance impact when implemented in the larger application.

Miniapps can serve as more accurate proxies for larger applications. Miniapps are applications on the order of 1000 lines of code, that include a simple build system, and accurately model the performance of a larger application. Heroux et al. [20] suggest that miniapps can be used to compare programming models and improve compilers. Our initial efforts in separating the discretization grid from the algorithm specification in PDE solvers has involved the development of a miniapp called CGPOP that serves as a proxy of POP. The CGPOP miniapp is the conjugate gradient solver from LANL POP 2.0.

We focus on the conjugate gradient solver because it is the largest performance bottleneck when POP is run on thousands of cores. In Figure 5a, we provide a breakdown of the relative computational cost of the three different components of the POP 0.1° benchmark when run

using varying numbers of cores on Kraken – a 99,072 core Cray XT5 system located at the National Institute for Computational Science (NICS). While the relative cost of the baroclinic component dominates at fewer than 1000 cores, the relative contribution of the communication intensive sections of the code, barotropic and the 3D-update dominates the total cost at greater than 1000 cores. For this reason we extracted CGPOP from the barotropic component, and the conjugate gradient algorithm within it.

CGPOP serves as a performance proxy of POP. In Figure 5b we compare the scalability of POP and CGPOP; note the proximity of each pair of similarly colored lines. The scalability of POP and the CGPOP miniapp are effected in a similar manner when changes are made in the machine and compiler used. Additionally, we examined the scalability of CGPOP and POP across three different platforms: Hopper, a Cray XE6 located at the National Energy Research Supercomputing Center (NERSC); Frost, a BlueGene/L located at the National Center for Atmospheric Research (NCAR); Lynx, a Cray XT5 also located at NCAR; and Kraken, a Cray XT5 located at the National Institute for Computational Science (NICS). The compilers we used in our examination were PGI Fortran, Cray Fortran, and XL Fortran.

The immediate benefit of developing CGPOP is that the POP developers are able to experiment with different performance optimizations and data structures. Future work includes developing an injectable programming model that enables the expression of stencil computations and communications orthogonally from the grid discretization specification. The CGPOP miniapp will provide a testbed for evaluating prototype injectable programming models.

4 Evaluating Task Graph Programming Constructs

Task graphs are frequently used to express course-grained partial parallelism and a number of static scheduling techniques are available for implementing such task graphs [28]. The dynamic scheduling of tile dependence graphs, which are essentially task graphs, has shown promise in terms of the tiling of regular applications [35] and of irregular/sparse applications [42]. As part of the SAIMI project we are developing abstractions that expose the static and/or dynamic scheduling of task graphs. We focus on task graphs that arise due to tiling and sparse tiling [43].

Sparse tiling is a transformation that tiles between and across loops in an irregular/sparse computation through the use of inspector/executor strategies [31]. One of the injectable programming models we have been developing is the Sparse Polyhedral Framework [41, 29]. The SPF enables the expression of loops with indirect memory accesses as well as the orthogonal expression of run-time reorderings of such loops where the loop transformation is done at compile-time and inspector code is generated to perform data reordering and create input data specific schedules at runtime. Sparse tiling is one of the transformations expressible within the SPF injectable programming model.

Sparse tiling results in the creation of an arbitrary task graph at runtime, where each tile is a task. Unfortunately, most existing schedulers today are optimized for one of two specific task graph topologies, namely DOALL as in the OpenMP parallel for loop or fork-join as in the Cilk spawn and sync. Task graphs for DOALL parallelism have a single entry node that spawns a single broad layer of tasks, each of which converges back to a single exit node. Fork-join parallelism takes the form of a task graph in the shape of a diamond. From the single entry node, some number of tasks are spawned. Each of these tasks can spawn additional tasks and so forth until all tasks are spawned. The lower portion of the task graph mirrors the upper portion symmetrically as each task spawned in the upper half reconverges into its continuation in the lower half. The upper half of the diamond task graph therefore performs the "fork"

```
1  ...
2  for (int i = firstInitialTileIndex; i <= lastInitialTileIndex; i++)  {
3      cilk_spawn computeJacobiTile(tile[i]);
4  }
5  cilk_sync;
6  // task graph execution complete
```

(a) Code using cilk_sync command and "diamond" task graph

```
1  ...
2  for (int i = firstInitialTileIndex; i <= lastInitialTileIndex; i++) {
3    cilk_spawn computeJacobiTile(tile[i]);
4  }
5  pthread_mutex_lock(&leafNodeMutex);
6  if (numPendingLeafNodes > 0)
7  {
8        pthread_cond_wait(&terminationCond, &leafNodeMutex);
9  }
10 pthread_mutex_unlock(&leafNodeMutex);
11
12 // task graph execution complete
```

(b) Code using additional pThread synchronization and arbitrary task graph

Figure 6: Additional tangling and code introduced to detect task graph execution completion

function while the lower half does the "join".

We evaluated the Cilk++ [10], OpenMP [3][33], Concurrent Collections [22], and TBB [4][21] programming models in terms of specifying the arbitrary task graphs that result from applying sparse tiling [44] to a sparse Jacobi computation. Most of the parallel programming models we surveyed assume a specific graph topology. For example, for Cilk++ and the OpenMP task model, the underlying assumption is that fork-join parallelism is being implemented [7]. The synchronization philosophy embedded in these programming models and the explicit synchronization constructs exposed to the programmer assume that each task will spawn other tasks, wait for them to complete, and then proceed with some type of reduction or data combining continuation. These models also support DOALL parallelism for parallel loops.

Concurrent Collections is designed to support dataflow pipeline parallelism, in which data and synchronization information flow dynamically through a statically specified pipeline [11]. Since the task graphs due to sparse tiling are generated at runtime, a general task pipeline must be statically implemented in CnC into which the specific dynamic tasks can be submitted for execution. Threading Building Blocks supports DOALL and fork-join parallelism and now supports task graphs directly in a community preview release [2]. We also implemented a pThreads scheduler specifically for executing task graphs, which takes advantage of the lower-level building blocks made available in the pThreads model.

Figure 6(a) shows a code snippet in Cilk++ that launches the initially available tasks from a task graph by submitting them to the underlying work-stealing scheduler. This code launches the initially ready entry tasks in the task graph. Similar code runs after each task completes and is used to launch successor tasks for which all dependencies are met. As can be seen, using this interface is quite simple and involves very few lines of code. The OMP Task model is very similar, with pragmas used in place of the cilk keywords. Unfortunately, this approach suffers from performance problems. If the code is left as-is, then there is a barrier invoked by the cilk_sync keyword after every task that waits for all the successors to complete. Each spawned successor task would likewise wait for all of its transitive children, resulting in several levels of cascaded barriers and reduced performance. This is a direct consequence of the model expecting fork-join parallelism, which follows from this execution pattern.

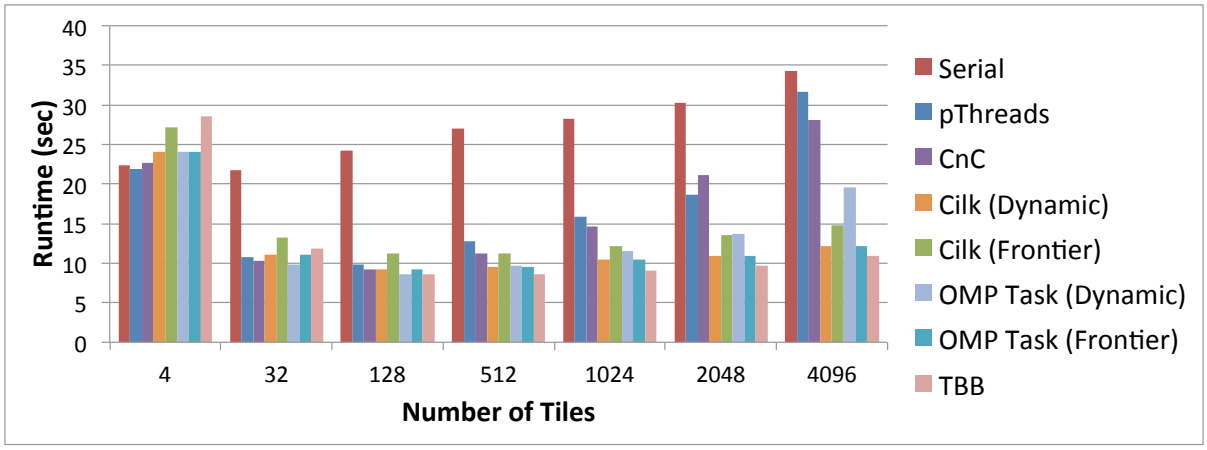


Figure 7: Run time using different parallel programming models for sparse Jacobi on the matrix Sphere150. Shorter bars are better.

To eliminate the nested barriers, a master thread could execute all available tasks, then wait at a single barrier. When these tasks completed, the next wave of tasks could then be launched. This approach reduces the barrier depth to a single barrier, but also introduces unnecessary synchronization between independent tasks. It also serializes the launching of new tasks onto a single thread. We refer to this approach as *frontier scheduling*.

A third approach is to remove the synchronization statement, e.g. cilk_sync, altogether. Each task launches any of its successors that are ready and then terminates. The difficulty with this approach is that the master thread, now without a continuation stack, is unaware of when the overall graph execution is complete. This problem can be resolved, but requires synchronization and communication facilities that are not available within the respective programming models. Partial code to accomplish this is shown in Figure 6(b). In our experimental results we use the third approach for OpenMP and Cilk++ and call it dynamic scheduling. Graph execution completion is communicated to the main thread using pThreads mutexes and condition variables. This approach reduces the call stack and barrier overhead and still allows tasks to execute without artificial synchronization, but greatly increases the program complexity and leads to tangling between the parallel implementation and the algorithm code.

Figure 7 shows the runtime of the sparse Jacobi solver using different parallel programming models as the task count increases. Execution time is in seconds on a Core i7 960 four core processor. The results shown in this graph suggest two findings. First, an inadequate degree of parallelism leads to underutilization of even a modest number of cores. This is unsurprising and accounts for the long runtimes seen with a low number of tiles. Secondly, as the tile count and the average degree of parallelism increase, the runtime also increases. This is due to an increase in overhead. When a frontier schedule is used for OpenMP, we observe only a minor increase in runtime at high tile counts. This increase suggests that much of the overhead is in synchronization and completion detection, rather than in assigning work to the threads. Others have found similar overhead costs [32]. The higher execution times for Cilk using a frontier schedule when compared to a dynamic schedule can be attributed to the mismatch between frontier scheduling and a work-first scheduling policy. When a single master thread spawns all child tasks under a work-first policy, the other cores must steal their tasks from that thread, leading to some inefficiency.

We believe that task graphs are a powerful and practical abstraction for the representation

of parallel scientific code. They encourage the separation of algorithm code from parallel implementation code. Unfortunately, none of the programming models we investigated provided a natural and succinct way to express arbitrary task graphs while also providing performance. We are examining the proposed TBB graph model, which appears very promising. We are also investigating techniques for improving data locality and task scheduling for task graph computation.

5 Conclusion

In the SAIMI project, the main approach we use for orthogonally specifying implementation details is to inject smaller programming models into full applications and specify implementation details as transformations on the injected programs. This paper presents specific examples of the SAIMI approach and how we are evaluating existing programming constructs in terms of the separation of algorithms and implementation details. The Mesa tool developed to semautomate the application of look-up table optimizations (Section 2) prototypes the pragma mechanism and realizes the expensive expression programming model as an injectable programming model. The other specific programming models that we plan to turn into injectable programming models include the polyhedral model applied to PDE discretizations (Section 3), the sparse polyhedral model for the specification and transformation of sparse computations, and the specification of dynamic task graphs that result from such transformations (Section 4). Finally, the development of evaluation criteria and evaluation of existing programming model mechanisms for the separation of algorithms and implementation details provides us ideas for developing injectable programming models and ways to evaluate future contributions within the context of the SAIMI project.

References

[1] Community Earth System Model. `http://www.cesm.ucar.edu/`.

[2] Intel TBB version 5 update 3 introduces graph as a community preview feature (http://software.intel.com/en-us/blogs/2010/12/23/intel-threading-building-blocks-version-30-update-5-introduces-graph-as-a-community-preview-feature-2).

[3] OpenMP (http://openmp.org).

[4] Threading Building Blocks (http://www.threadingbuildingblocks.org).

[5] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger. Stapl: An adaptive, generic parallel programming library for c++. In *Workshop on Languages and Compilers for Parallel Computing (LCPC)*, Cumberland Falls, Kentucky, August 2001.

[6] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. A view of the parallel computing landscape. *Communications of the ACM*, 52(10):56–67, 2009.

[7] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The design of openmp tasks. *IEEE Trans. Parallel Distrib. Syst.*, 20:404–418, March 2009.

[8] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks. Technical report, The International Journal of Supercomputer Applications, 1991.

[9] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th Interntional Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2004.

[10] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. Technical report, Cambridge, MA, USA, 1996.

[11] Z. Budimlic, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, and S. Tasirlar. Concurrent collections. *Sci. Program.*, 18:203–217, August 2010.

[12] D. Callahan, B. Chamberlain, and H. Zima. The cascade high productivity language. In *Proceedings of the Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments*, pages 52–60, 2004.

[13] B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[14] L. Dagum and R. Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Computational Science & Engineering*, 5(1):46–55, 1998.

[15] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Sixth Symposium on Operating System Design and Implementation (OSDI)*, 2004.

[16] L. Deng, C. Chakrabarti, N. Pitsianis, and X. Sun. Automated Optimization of Look-up table Implementation for Function Evaluation on FPGAs. volume 7444. SPIE, 2009.

[17] P. Feautrier. Some efficient solutions to the affine scheduling problem. I. one-dimensional time. *International Journal of Parallel Programming*, 21(5):313–347, October 1992.

[18] S. Gal. Computing Elementary Functions: A New Approach for Achieving High Accuracy and Good Performance. In *Proceedings of the Symposium on Accurate Scientific Computations*, pages 1–16, London, UK, 1986. Springer-Verlag.

[19] M. Griebl, C. Lengauer, and S. Wetzel. Code generation in the polytope model. In *In IEEE International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 106–111. IEEE Computer Society Press, 1998.

[20] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich. Improving performance via mini-applications. Technical Report SAND2009-5574, Sandia National Laboratories, 2009.

[21] Intel Corporation. *Intel Threading Building Blocks Reference Manual*, 2009.

[22] Intel Corporation. *Concurrent Collections for C++*, September 2010.

[23] P. Jones. Parallel Ocean Program (POP) user guide. Technical Report LACC 99-18, Los Alamos National Laboratory, March 2003.

[24] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. A framework for interprocedural locality optimization using both loop and data layout transformations. In *Proceedings of the 28th International Conference on Parallel Processing (28th ICPP'99)*, Aizu-Wakamatsu, Fukushima, Japan, 1999. University of Aizu.

[25] W. Kelly and W. Pugh. Finding legal reordering transformations using mappings. In K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*, volume 892 of *Lecture Notes in Computer Science*, pages 107–124. Springer, 8–10, 1994.

[26] C. D. Krieger, A. Stone, and M. M. Strout. Mechanisms that separate algorithms from implementations for parallel patterns. In *Workshop on Parallel Programming Patterns (ParaPLOP)*, March 2010.

[27] C. D. Krieger, A. I. Stone, and M. M. Strout. Qualitative evaluation criteria for parallel programming models. In *Proceedings of the Fun Ideas and Thoughts Session at PLDI*, June 8, 2010.

[28] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4):406–471, 1999.

[29] A. LaMielle and M. M. Strout. Enabling code generation within the sparse polyhedral framework. Technical report, Technical Report CS-10-102 Colorado State University, March 2010.

[30] P. Luszczek, J. J. Dongarra, D. Koester, R. Rabenseifner, B. Lucas, J. Kepner, J. Mccalpin, D. Bailey, and D. Takahashi. Introduction to the HPC challenge benchmark suite. `http://icl.cs.utk.edu/projectsfiles/hpcc/pubs/hpcc-challenge-benchmark05.pdf`, 2005.

[31] R. Mirchandaney, J. H. Saltz, R. M. Smith, D. M. Nico, and K. Crowley. Principles of runtime support for parallel processors. In *Proceedings of the 2nd International Conference on Supercomputing*, pages 140–152, 1988.

[32] S. L. Olivier and J. F. Prins. Evaluating OpenMP 3.0 run time systems on unbalanced task graphs. In *Proceedings of the 5th International Workshop on OpenMP*, pages 63–78, 2009.

[33] OpenMP Architecture Review Board. *OpenMP Application Program Interface*, May 2008.

[34] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary. HPL - a portable implementation of the high-performance linpack benchmark for distributed-memory computers, version 2.0. `http://www.netlib.org/benchmark/hpl/`, 2008.

[35] G. Quintana-Ortí, F. D. Igual, E. S. Quintana-Ortí, and R. A. van de Geijn. Solving dense linear systems on platforms with multiple hardware accelerators. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 121–130, New York, NY, USA, 2009. ACM.

[36] L. Rauchwerger, F. Arzu, and K. Ouchi. Standard templates adaptive parallel library. In *Proceedings of the 4th International Workshop on Languages, Compilers and Run-Time Systems for Scalable Computers (LCR)*, Pittsburg, PA, May 1998.

[37] J. Riley. *Writing Fast Programs: A Practical Guide for Scientists and Engineers.* Cambridge International Science Publishing, 2006.

[38] ROSE Project, 2011.

[39] V. Sarkar and R. Thekkath. A general framework for iteration-reordering loop transformations. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation (PLDI)*, pages 175–187, June 1992.

[40] K. Sobti, L. Deng, C. Chakrabarti, N. Pitsianis, X. Sun, J. Kim, P. Mangalagiri, K. Irick, M. Kandemir, and V. Narayanan. Efficient Function Evaluations with Lookup Tables for Structured Matrix Operations. In *Signal Processing Systems, 2007 IEEE Workshop on*, pages 463 –468, Oct. 2007.

[41] M. M. Strout, L. Carter, and J. Ferrante. Compile-time composition of run-time data and iteration reorderings. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, New York, NY, USA, June 2003. ACM.

[42] M. M. Strout, L. Carter, J. Ferrante, J. Freeman, and B. Kreaseck. Combining performance aspects of irregular Gauss-Seidel via sparse tiling. In *Proceedings of the 15th Workshop on Languages and Compilers for Parallel Computing (LCPC)*, Berlin / Heidelberg, July 2002. Springer.

[43] M. M. Strout, L. Carter, J. Ferrante, and B. Kreaseck. Sparse tiling for stationary iterative methods. *International Journal of High Performance Computing Applications*, 18(1):95–114, February 2004.

[44] M. M. Strout, L. Carter, J. Ferrante, and B. Kreaseck. Sparse tiling for stationary iterative methods. *Int. J. High Perform. Comput. Appl.*, 18(1):95–113, 2004.

[45] P.-T. P. Tang. Table-driven Implementation of the Exponential Function in IEEE Floating-point Arithmetic. *ACM Transactions on Mathematical Software*, 15(2):144–157, 1989.

[46] P.-T. P. Tang. Table-lookup Algorithms for Elementary Functions and their Error Analysis. In *The Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, 1991.

[47] C. Wilcox, M. Strout, and J. Bieman. Mesa: Automatic generation of lookup table optimizations. In *Proceedings of the 4th International Workshop on Multicore Software Engineering*, IWMSE '11, New York, NY, USA, in press. ACM.

[48] M. E. Wolf and M. S. Lam. Loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, October 1991.

[49] Y. Zhang, L. Deng, P. Yedlapalli, S. Muralidhara, H. Zhao, M. Kandemir, C. Chakrabarti, N. Pitsianis, and X. Sun. A Special-Purpose Compiler for Look-up Table and Code Generation for Function Evaluation. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pages 1130 –1135, Mar. 2010.