Automatically Generating Coarse Grained Software Pipelining from Declaratively Specified Communication

Nilesh Mahajan* Sajith Sasidharan* Arun Chauhan Andrew Lumsdaine

School of Informatics and Computing, Indiana University, Bloomington, IN 47405, USA
{nnmahaja,sasasidh,achauhan,lums}@cs.indiana.edu

Abstract

We have been developing a declarative approach to writing parallel programs, in a language called Kanor, with the goal of addressing the productivity challenge in parallel programming. A guiding principle in the design of our approach has been to abstract away those details that a compiler can feasibly and effectively handle, while still affording the programmers the ability to optimize their parallel programs. A critical job of the compiler is to optimize communication in the parallel program in the context of the surrounding computation. In this paper, we demonstrate that it is possible to perform advanced optimizations in such a compiler by devising an algorithm that identifies software pipelining opportunities in a program written with Kanor and generates software pipelined code for a cluster, using MPI.

1 Introduction

Writing parallel programs for high performance is widely regarded to be difficult and error-prone. Experience with parallel programming languages and compilation technology has shown that balancing demands from users having widely varying expertise with practicable compiler techniques is a difficult issue [9]. We have strived to address some of those issues with our *declarative* language, called Kanor, aimed at improving the productivity of parallel programmers. Kanor is designed to be a domain-specific language (DSL) embedded in a host language (currently C/C++), which is used to declaratively specify communication [7]. The goal of the language is to allow users to write explicitly parallel programs so that they are not only encouraged to "think in parallel", but also enjoy a greater control over the execution of their program than implicitly parallel languages could provide, enabling finer performance tuning. At the same time, the Kanor compiler eliminates the need for manually carrying out communication optimization, such a communication-computation overlap, greatly improving the program's readability and portability. Declarative specification of communication also opens up opportunities for formal analysis of the parallel program, letting the compiler provide correctness guarantees, such as freedom from deadlocks. The BSP style Kanor approach is better than *low level* message passing[5].

One of the key optimization techniques in high performance parallel programs is software pipelining. Compiler support for pipelining varies across languages[11]. Pipelining is especially useful in cases where there are cross-processor data dependencies that limit parallelism, as the receivers wait for the senders to finish. Strip-mining the computation creates a pipeline, leading to parallelism that did not exist in the original form of the code. In this paper we present an algorithm that transforms Kanor programs into pipelined MPI programs. The algorithm first identifies the cases where opportunities for software pipelining might exist and then transforms the communication and its dependent computation to take advantage of the pipeline. It is a part of our under-development Kanor-to-C++ compiler, which is written using the ROSE compiler framework [12]. We evaluate the performance of our algorithm on three benchmarks that are amenable to software pipelining.

*Student author.

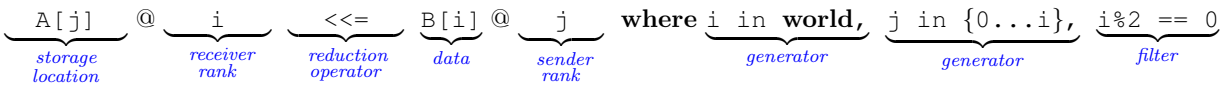

Figure 1: Kanor syntax.

2 Software Pipelining in Kanor

Kanor was motivated by applications written primarily in the Bulk-Synchronous Parallel (BSP) style [13]. Therefore, BSP-style applications are the most common use case for Kanor. A communication step is of the form $\{e_0@e_1 << op << e_2@e_3$ `where` $e_4\}$ enclosed within a block marked `@communicate`. Communication steps are embedded in the host language and respect its semantics. The operator `op` can be any general reduction operation defined by the user, as long as it is commutative and associative. The expression e_4 may contain multiple clauses, as illustrated by the example in Figure 1. The example uses the special syntax `<<=` for the most commonly occurring "reduction", which is data copying. The set comprehension syntax within the `where` clause emphasizes the fact that all communication in one `@communicate` block is parallel, thus free of dependencies that might cause serialization.

In order for a communication step to lead to software pipelining, certain conditions must hold.

Global knowledge case All free variables in the `@communicate` block must evaluate to the same value on all processors (we refer to this as the *global knowledge case* [7]). This means that no free variable is rank- or I/O-dependent. The compiler performs information flow analysis to make this determination. The dependence may be due to either data or control.

Sending and receiving ranks All processors must take part in communication. Specifically, for N processors, senders $\in \{0 \ldots N-2\}$, and receivers $\in \{1 \ldots N-1\}$. To verify this, the compiler analyzes the filters in the `where` clause, which are boolean conditions on the rank variables. The filter conditions, formulated as a constraint satisfaction problem, must have a solution such that sender ranks $\in \{0 \ldots N-2\}$ and receiver ranks $\in \{1 \ldots N-1\}$. Strictly speaking, not all processors need to participate in a communication step for software pipelining to be useful, only a *sufficiently large number* of them. However, a collective communication operation involving all processors is a common case in most current data-parallel applications.

Directional communication We require that there exist a total ordering of processor ranks (say, denoted by $\prec$), such that p sends data to $q \Rightarrow p \prec q$. Unconstrained communication patterns can substantially complicate the problem of determining whether all the data flows in a single direction. Fortunately, most practical applications that can benefit from software pipelining follow simple communication patterns. Nevertheless, multi-dimensional processor grids—common in high-performance code—make the problem non-trivial to solve. An imprecise test that works in practice is verifying that the data flows in only one direction individually along each processor dimension. If it does then the overall data movement is also in a single direction. If it does not, there might still be a total ordering of processors that results in a directional movement of data, however we interpret the outcome to mean that software pipelining does not apply.

Granularity of communication Since we are primarily interested in coarse-grained software pipelining that is useful on clusters, it is desirable to have larger granularity of communicated data. Thus, we assume that communication has already been aggregated where possible [1]. Only those communicate steps are candidates for software pipelining where the communicated data size, after

```
1  Algorithm: Generate Software Pipeline
2  Input: Code region, C; Communicate block, B; Strip-mining factor, S; Dependence graph, G
3  Output: Software pipelined code
   ---------------------------------------------------------------------------
4  F ← empty set of loops
5  F ← Strip-mine B
6  for each loop, L ∈ C, that uses B's send or receive buffer do
7      F ← F ∪ Strip-mine L
8  F' ← typed-loop-fusion(F, G) // apply Allen and Kennedy's algorithm [1]
9  abort, if fusion fails
10 return C, with F replaced by F'
```

Algorithm 1: Algorithm to generate software pipelined code from Kanor.

aggregation, is "sufficiently" large[1].

Once a candidate `@communicate` block has been identified, the compiler uses Algorithm 1 to generate pipelined code. It takes the communicate block and its surrounding code block, which it will attempt to pipeline. The algorithm also takes the strip-mining factor as a parameter. We leave determining the exact value of this factor for future work.

Figure 2 shows examples of the benchmarks we studied, and the pipelined version for one of them. Note that the transformed code is still in Kanor, not MPI. The code is eventually translated into MPI through a sequence of steps that have been omitted here for the sake of brevity. Correctness of the transformation follows from the observation that it is always legal to strip-mine communication, as well as element-wise array computation, and from the correctness of Allen and Kennedy's loop-fusion algorithm.

3 Experimental Evaluation and Discussion

We studied three well-known kernels used in high performance numerical applications, iterative Jacobi, Cholesky factorization, and Sweep3D. Each of these kernels operates on dense matrices and is amenable to software pipelining, but each exhibits distinct communication and computation patterns.

Figure 2 shows how each of the benchmarks is coded in Kanor. The figure also shows the resulting software pipelined code after applying Algorithm 1 on one of the benchmarks (Sweep3D). The code in the figure uses a two-dimensional *virtual processor* grid, to keep it simple for the sake of exposition. Pipelined versions of others are similar and have been omitted for the sake of brevity.

Performance benefits of software pipelining are well-established. Therefore, a higher performance from pipelined code should be expected. However, in order to gain maximum possible benefit the compiler will need to account for the target platform characteristics to choose an appropriate blocking factor or even decide when software pipelining could be advantageous. We expect to incorporate a cost model into the Kanor compiler to equip it to make platform-specific decisions.

Since Kanor is designed to target clusters as well as shared memory architectures [8], software pipelining decisions may be dramatically different depending on the availability of shared memory for inter-process communication. For example, lower inter-process communication latencies with shared memory may make software pipelining worthwhile in a larger set of scenarios.

4 Related Work

Early work on software pipelining in compilers focused on pipelining across loop iterations for instruction-level parallelism [10]. Recently, streaming languages, such as StreamIt, have imple-

[1]Determining the right threshold data size is an orthogonal problem, which is outside the scope of this paper.

Jacobi

```
do {
  /* Local matrix expanded by shadow columns/rows to store remote values */
  @communicate {xlocal[M]@(x,y-1) <<= xlocal[1]@(x,y) where x, y in {0...N-1}}
  @communicate {xlocal[M]@(x,y-1) <<= xlocal[1]@(x,y) where x, y in {0...N-1}}
  @communicate {xlocal[M]@(x,y-1) <<= xlocal[1]@(x,y) where x, y in {0...N-1}}
  @communicate {xlocal[M]@(x,y-1) <<= xlocal[1]@(x,y) where x, y in {0...N-1}}
  compute_interior_and_diffnorm(xlocal, &gdiffnorm);
  // all-reduce
  @communicate {gdiffnorm@dest_rank << sum <<  diffnorm@src_rank
    where dest_rank, src_rank in {0...N-1};}
  gdiffnorm = sqrt( gdiffnorm );
  itcnt++;
} while(diffnorm > 1.0e-2 && itcnt < 100);
```

Cholesky

```
for (int curr_col = 0; curr_col < MATSIZE; curr_col++) {
  if (owns(myid, curr_col)) {
    // If I own the current column then compute
    for (int k = 0, k < curr_col; k++) { // previous columns
      for (int i = curr_col, i < MATSIZE; i++) { // rows of current column
        A[i][j] -= temp_cols[i][k] * temp_cols[j][k];
      }
    }
    A[curr_col][curr_col] = sqrt(A[curr_col][curr_col]);
    for (int k = curr_col + 1; k < MATSIZE; k++) { // update current column
      A[k][curr_col] /= A[curr_col][curr_col];
    }
  }
  @communicate {temp_cols[][x]@(x+i) <<= A[][curr_col]@x
          where x in {0...N-1} and i in {1...(N-1-x)} and owns(x, curr_col)}
}
```

Sweep3D

```
for (int i = 0; i < OCTANTS; i++) {
  for (int j = 0; j < ANGLES; j++) {
    // loop though the diagonals, N is the number of processors
    for (int diag = 0; diag < 2 * N + 1; diag++) {
      if ((myid.x + myid.y) == diag) { compute(); } /* wave front */
      @communicate {temp_s@(x, y+1) <<= A[lastrow]@(x, y)
                    where x, y in {0...N-1} and x + y = diag;}
      @communicate {temp_e@(x + 1, y) <<= A[][lastcol]@(x, y)
                    where x, y in {0...N-1} and x + y = diag;}
}}}
```

Sweep3D pipelined

```
for (int i = 0; i < OCTANTS; i++) {
  for (int j = 0; j < ANGLES; j++) {
    for (int s = 0; s < min(SIZE, s + BLOCK_SIZE); s+=BLOCK_SIZE) {
      // loop though the diagonals, N is the number of processors
      for (int diag = 0; diag < 2 * N + 1; diag++) {
        if ((myid.x + myid.y) == diag) { strip_mined_compute(); }
        @communicate {temp_s@(x, y+1) <<= A[lastrow]@(x, y)
                      where x, y in {0...N-1} and x + y = diag;}
        @communicate {temp_e@(x + 1, y) <<= A[][lastcol]@(x, y)
                      where x, y in {0...N-1} and x + y = diag;}
}}}}
```

Figure 2: Jacobi, Sweep3D, and Cholesky kernels in Kanor, and Sweep3D pipelined.

mented coarse-grained software pipelining [4]. However, these languages usually have a more restrictive programming model than Kanor, with emphasis on streaming applications.

Software pipelining has also been studied in the context of PGAS (Partitioned Global Address Space) languages, including X10, UPC, and HPF [2, 3, 6]. Fortran D compiler identified cross-processor loops and tried to strip-mine them [6]. Even though similar in its goal, our analysis occurs in the context of an explicitly parallel program, which requires us to first infer the communication pattern. However, the declarative specification of communication helps the compiler extract an accurate high-level picture of the communication pattern and dependencies.

5 Conclusion and Future Work

In this paper, we have presented an algorithm to identify opportunities for coarse-grained software pipelining in Kanor programs, and a code generation algorithm to automatically generate the pipelined code. Unlike prior efforts in the context of languages with implicitly specified parallelism, the algorithm to exploit software pipelining in Kanor is simpler and easier to assess for potential effectiveness. We expect that this will enable the compiler to perform more aggressive optimizations with Kanor than are possible with implicitly parallel languages. Future work includes studying a larger class of applications and platforms, and extending the algorithm to sparse matrix algorithms.

6 Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. CNS-0834722. We thank Eric Holk, Jeremiah Willcock, and William Byrd for helpful discussions.

References

[1] J. R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, California, USA, 2001.

[2] R. Barik, J. Zhao, D. Grove, I. Peshansky, Z. Budimlic, and V. Sarkar. Communication optimization for distributed-memory X10 programs. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1101–1113, 2011.

[3] W.-Y. Chen, C. Iancu, and K. Yelick. Communication optimizations for fine-grained UPC applications. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 267–278, 2005.

[4] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 151–162, 2006.

[5] S. Gorlatch. Send-receive considered harmful: Myths and realities of message passing. *ACM Transactions on Programming Languages and Systems*, 2004.

[6] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, Aug. 1992.

[7] E. Holk, W. E. Byrd, J. Willcock, T. Hoefler, A. Chauhan, and A. Lumsdaine. Kanor: A declarative language for explicit communication. In *Proceedings of the Thirteenth International Symposium on the Practical Aspects of Declarative Languages (PADL)*, 2011.

[8] F. Jiao, N. Mahajan, J. Willcock, A. Chauhan, and A. Lumsdaine. Partial globalization of partitioned address spaces for zero-copy communication with shared memory. In *Proceedings of the 18th International Conference on High Performance Computing (HiPC)*, 2011.

[9] K. Kennedy, C. Koelbel, and H. Zima. The rise and fall of High Performance Fortran: An historical object lesson. In *Proceedings of the third ACM SIGPLAN Conference on History of Programming Languages*, pages 7–1–7–22, 2007.

[10] M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 318–328, 1988.

[11] E. C. Lewis and L. Snyder. Pipelining wavefront computations: Experiences and performance. In *IN FIFTH IEEE INTERNATIONAL WORKSHOP ON HIGH-LEVEL PARALLEL PROGRAMMING MODELS AND SUPPORTIVE ENVIRONMENTS (HIPS)*, 1999.

[12] D. Quinlan, C. Liao, T. Panas, R. Matzke, M. Schordan, R. Vuduc, and Q. Yi. *ROSE User Manual: A Tool for Building Source-to-Source Translators*. Lawrence Livermore National Laboratory, Livermore, CA 94550, version 0.9.5a edition, Nov. 2010.

[13] L. G. Valiant. Bulk-synchronous parallel computers. In M. Reeve, editor, *Parallel Processing and Artificial Intelligence*, pages 15–22. John Wiley & Sons, 1989.