LU Factorization: Towards Hiding Communication Overheads With A Lookahead-free Algorithm

Tan Nguyen and Scott B. Baden
Department of Computer Science and Engineering
University of California, San Diego, La Jolla, CA 92093 USA
Email: {nnguyent, baden}@ucsd.edu

Abstract—Lookahead is a well-known technique for masking communication in matrix factorization, but at the cost of complicating application software. We present a new approach, based on automated code-restructuring, that realizes the benefits of lookahead while avoiding the complications. We apply our technique to HPL, the Linpack benchmark used to assess the performance of supercomputers. Starting with the simpler non-lookahead version of the application, we are able to meet the performance of lookahead on the Stampede mainframe.

I. INTRODUCTION

At present, it appears that significant increases in scalable system performance will come largely at the node level [1]. This expectation poses significant software challenges, since increased node performance amplifies the cost of communication, requiring significant algorithmic changes and performance programming. Lookahead is a traditional technique for masking communication costs in matrix factorization such as LU decomposition [2]. However, lookahead adds complicating implementation details [3] that have prevented it from being used in practice. This predicament has motivated new algorithmic reformulations [4], [5] or data-driven implementations [4], [6]–[8] to realize overlap.

We present a new approach to masking communication in factorization arising on distributed memory systems. Our approach realizes the benefits of lookahead but without having to reformulate existing source code. We use a custom source-to-source translator to restructure MPI source into a task parallel form that runs as a data-driven program under the control of an external scheduler. Our translator collects static information about communication call sites (i.e., arguments of message passing calls) and then generates code that will construct, at run time, a distributed task precedence graph representing data dependencies expressed by the communication patterns arising in the running application. The scheduler orchestrates communication and computation to realize overlap automatically. The result of this reformulation is to reinterpret the execution of an MPI program in terms of a data driven execution model. We found that task scheduling plays an important role in meeting performance goals, in particular, application specific knowledge was crucial in determining a good schedule. A simple system of priorities was sufficient to enable the code to compete with lookahead.

II. RELATED WORK

HPL [9] implements a lookahead variant of distributed-memory LU factorization that overlaps communication with computation, and is written in MPI. Lookahead is difficult to implement in practice. For example, Chan et al. enumerated the challenges in applying lookahead in libraries of LU factorization and other algebra algorithms [4]. These challenges include difficulties in finding the computational bottlenecks and partitioning execution in a split-phase manner to enable parts of different iterations to run in parallel. Lookahead is not supported in the well known ScaLAPACK libraries [10].

To our knowledge, no prior work has applied automatic code restructuring to realize the performance benefits of lookahead via the simpler non-lookahead version of the code. Our result is based on the observation that a data-driven execution model is a viable solution to the problem. Others have applied data-driven execution to various dense linear algebra kernels, though not to the problem of handling lookahead. Lifflander et al. implemented HPL in Charm++ [8], weakly scaling on up to 8K processors of a Cray XT5. DPLASMA [11] is a distributed-memory implementation of linear algebra factorizations (Cholesky, LU, QR), based on the PLASMA library [12], which represents the program as a DAG. Husbands and Yelick described an implementation of LU factorization in Unified Parallel C (UPC) [6] that oversubscribes the processors. Although we have employed multiple tasks per processor, these tasks are executed by a single thread that runs to completion. Thus, we don’t oversubscribe the processors with multiple threads. Different from all these approaches, we use a translator and run time to generate the task graph automatically.

III. LU FACTORIZATION AND LOOKAHEAD

The LU factorization algorithm consists of n-1 stages, each corresponding to a column of the input matrix. A stage begins by identifying the element of the maximum magnitude in the portion of the column below the diagonal, called the pivot. If necessary, the row containing the pivot is then swapped with the current row i. To maximize locality, we use the blocked
algorithm shown in Fig. 1(a). Except for pivot selection and row interchange, the blocked algorithm updates $L$, $U$, and $A$ at the block granularity. Since the trailing submatrix of $A$ shrinks as factorization proceeds to the right, to balance the workloads we map blocks cyclically to MPI processes (Fig. 1(b).) Finally, we update the trailing submatrix below and to the right of the diagonal, the block labeled $A$ in Fig. 1(a).

A message passing implementation of LU factorization consists of 3 principal operations: panel factorization ($pFact$), panel broadcast ($pBcast$), and the trailing submatrix update ($pUpdate$). $pFact$ finds the pivots in a column panel. This step is costly since we have to factorize a skinny matrix over a subset of the processes that own the panel (the regions $D$ and $L_i$ in Fig. 1(a)), including a sequence of row swap-broadcasts, one for each pivot within a single column of the panel. In addition, once the panel has been factorized it must be broadcast to column processes within the same row ($pBcast$). An efficient implementation uses a ring broadcast, shifting data to the right along column processes. The $pUpdate$ operation broadcasts $U$ among row processes and then performs a rank-1 update. It accounts for the lion’s share of LU’s computational work, performing $O(N^3)$ multiply-adds.

Lookahead [4], [9] is a technique for overlapping communication with computation that fills idle gaps in the execution of LU. It utilizes the dependence structure of the blocked algorithm to orchestrate computation and data motion, employing split-phase coding [13] to compute multiple iterations in advance. The lookahead strategy splits the trailing submatrix update into 2 phases: partial update performs the trailing submatrix update on the current panel only, the remaining update computes on remaining panels owned by a process. Once a process receives a panel, it performs a partial update so that it can quickly factorize the next panel. During the panel broadcast of the next panel, the process can perform useful work in the remaining update phase.

## IV. SCHEDULING

### A. Task graph execution

We restructure an MPI implementation so that it runs as a task-parallel program. To this end we utilize the Bamboo translator [14] with enhancements to improve task scheduling. In effect Bamboo re-interprets an MPI program in terms of a task precedence graph, in which vertices denote tasks and edges represent task dependencies. The underlying execution model is like dataflow: a task can execute once all inputs are ready and parallelism can be realized among independent tasks. Our data-driven model provides for task state, which includes performance meta-data that can be examined by the scheduler. These meta data are annotations to the task precedence graph, and are abstract entities, such that scheduler and application are unaware of one another. Thus, meta-data provide a convenient mechanism for expressing application specific scheduling.

The scheduler assigns tasks to resources and enforces the data-driven firing rules, moving data between tasks when a task finishes. We cannot be precise about when data actually moves; since data motion is not imperative, but comes as a consequence of satisfying dependencies, it moves as needed and at times of the scheduler’s choosing. When a task finishes computing it either terminates (this task has completed the final iteration) or it suspends, waiting on data for the next iteration. Data motion is handled outside the scope of task execution. As a consequence, there is no explicit waiting on data and a task won’t execute until all data are ready. To realize overlap, it is generally necessary to employ processor virtualization. Virtualization specifies that the tasks be more numerous than physical processing cores. The primary goals of virtualization are to balance the workloads and to hide latency [15].

In the original design of Bamboo, there is neither task pre-emption nor processor yielding. This design choice was made to maintain high hit rates for cache and TLB [14]. However, we found that processor yielding is necessary to improve the performance of LU factorization. To understand how Bamboo realizes communication overlap, consider the execution snapshot in Fig. 2. We virtualize in only one dimension—column tasks—by a factor of 4. Thus, the number of column tasks is 16 (4 MPI processes x 4) and the number of row tasks is 2 (2 MPI processes x 1). Task 00 ($T_{00}$) starts as the root of its processor row. It factorizes its panel and delivers the factorized panel out to $T_{01}$. $T_{00}$ yields the processor since the data ($U$) for the next step is likely unavailable. $T_{01}$ receives input from $T_{00}$ which it forwards to $T_{02}$. $T_{01}$ then yields the processor and waits, for $U$ for the next step, coming from the neighboring column process. This pattern continues with $T_{02}$ and $T_{03}$. Now, there may be more than 1 task ready to execute (e.g. $T_{00}$ and $T_{01}$.) Let’s say that the scheduler picks $T_{01}$ and executes $U_{00}$ (Fig. 2). $T_{01}$ next becomes the root, factorizes its panel, and starts a new panel broadcast. Although we have not completed the current step, $T_{01}$ initiates communication of $U$ for the next step. During this time $T_{00}$, $T_{02}$ and $T_{03}$ can perform the trailing update of the current phase ($U_{0,j}$, $j = 0, 2, 3$). Thus, we have succeeded in scheduling computation of one task while
performing communication of a different and independent task, thereby overlapping communication and computation. This strategy is different from that of the lookahead algorithm used in HPL, which must poll, at various points in the code, for opportunities to realize overlap. The data-driven strategy is a more natural means of realizing overlap than polling, as it factors scheduling out of the application.

B. Scheduling heuristics

There are many possibilities for ordering tasks but some orderings result in better performance than others. Referring to Fig. 2, if the scheduler picks $U_{00}$ of task $T_{00}$ before $T_{01}$’s $U_{00}$, panel factorization and the broadcast of panel 01 will be delayed. A question then arises: how do we determine a good task ordering in general? There are 2 possibilities: assign all tasks the same priority or assign different priorities based on knowledge about the graph and its execution. In the first option, we simply employ a FCFS queue of tasks, which is the default supported by Bamboo. The second option is guided by the insight that information flows in two different directions in the task geometry. First, the task that is performing panel factorization needs to be scheduled early on so that it can initiate the next phase of execution. Second, although panel broadcast (within a row) is a bit less important, it can be performed quickly compared to the panel factorization since all tasks share the same MPI process and the broadcast involves a shift only.

With these two observations in mind, we apply the following heuristics to task scheduling, remembering that the scheduler knows nothing about the application, but does have access to meta data introduced by the user, that decorates the task graph. First, after a task has been factorized and has shifted a panel, its becomes less urgent and thus it should yield the processor and lower its priority. Second, a task receiving input shifted in from the root will become the next root. A question then arises: how do we determine a good task ordering in general? There are 2 possibilities: assign all tasks the same priority or assign different priorities based on knowledge about the graph and its execution. In the first option, we simply employ a FCFS queue of tasks, which is the default supported by Bamboo. The second option is guided by the insight that information flows in two different directions in the task geometry. First, the task that is performing panel factorization needs to be scheduled early on so that it can initiate the next phase of execution. Second, although panel broadcast (within a row) is a bit less important, it can be performed quickly compared to the panel factorization since all tasks share the same MPI process and the broadcast involves a shift only.

With these two observations in mind, we apply the following heuristics to task scheduling, remembering that the scheduler knows nothing about the application, but does have access to meta data introduced by the user, that decorates the task graph. First, after a task has been factorized and has shifted a panel, its becomes less urgent and thus it should yield the processor and lower its priority. Second, a task receiving input shifted in from the root will become the next root, meaning that it becomes more urgent. However, this task should yield the processor to the next tasks (if any) that shift data and then increase its priority to regain the processor. It can then update the current panel, perform factorization and broadcast the next panel.

V. EXPERIMENTAL EVALUATION

A. System specifications

We ran on Stampede, located at at the Texas Advanced Computing Center (TACC). We used only the two 8-core Sandy Bridge processors on each node, but not the MIC coprocessor. Nodes communicate via an InfiniBand interconnect with a 2-level fat-tree technology. We compiled code with the Intel C++ compiler, version 13.0. We use Mvapich’s implementation of MPI for off-node communication.

B. Code variants and problem sizes

As the starting point for our experiments, we used the publicly available HPL code. We used two variants, the original MPI code with and without the lookahead optimization. We then generated two additional variants. Variant 3 is the result of translating the non-lookahead code with Bamboo. The fourth variant was obtained by making modest changes to the code generated by Bamboo, that provide hints to the scheduler to prioritize tasks in order to improve overlap. We select small enough problem sizes so that the communication overhead is significant and thus we can see the benefit of overlapping communication with computation.

C. Panel size and processor geometry

We manually tuned the panel size and found that a value of 96 delivered optimal performance for the selected problem sizes on Stampede. HPL’s performance is also sensitive to the processor geometry. Specifically, if the width of the processor geometry is too large, the panel broadcast will be very costly. However, if the height of the processor geometry is too large, the overheads of the panel factorization and U broadcast will be substantially increased. As a result, we always choose a nearly square processor geometry with a constraint that the size of each dimension is a power of 2. In the task graph variant we virtualize processor rows only. Thus, the width of the task graph is always 4 times the height.

D. Analysis of Results

Fig. 3 shows that the lookahead and prioritized task graph variants always outperform the non-lookahead one on every problem size and on any number of nodes. It can also be seen that, for a fixed number of nodes, the performance improvement is more significant with small problem sizes. On 32 nodes the benefit of overlapping is 8% with the smallest value of N but only 4% with the largest N. Similarly the benefit ranges from 10% to 6% on 64 nodes and 8% to 5% on 128 nodes. Reducing the problem size further, however, may decrease the performance benefit since we would not have enough computation to overlap with communication. Fig. 3(d) presents the amount of computation (matrix multiplication, $dgemm$) that can be used to hide communication. The fraction of time spent in $dgemm$ ranges from 58% to 73%. These results also validate our earlier analysis, showing that the relative overhead of communication shrinks as the problem size grows. Generally speaking, the results in Fig. 3 show that the prioritized task graph variant meets the performance of lookahead on most problem sizes. This result is likely to hold with larger problem sizes and is independent of the number of nodes.

The importance of task prioritization is inevitable. Theoretically, if we use a random scheduling algorithm and run with the unprioritized task graph variant for a large number of times, there is a possibility of observing the performance of the prioritized task graph variant. However, the required number of experiments will grow quickly as the number of panel columns of the input matrix increases. We performed many experiments, but results without task prioritization performed far below that of lookahead. Compared to the non-lookahead variant, the performance of the unprioritized task graph is at best comparable and in some case it is even lower.

The data-driven strategy requires some additional buffer copying which requires carefully management. Since tasks exchange data asynchronously, data must be copied from the application’s send buffer to a temporary buffer prior to transmission, and from the runtime system’s buffer to the corresponding application’s receive buffer. To reduce the copying overhead, the Bamboo translator automatically generates
OpenMP code to parallelize the memory copying whenever possible.

VI. CONCLUSION

We have presented a new approach to realizing overlap in LU factorization, based on the data-driven execution of a task graph generated from a conventional MPI program. We showed how to obtain the equivalent benefits of a complicated lookahead technique through analysis of conventional and simpler code. We applied our technique to High Performance Linpack, a popular benchmark that performs LU factorization. Experimental results on up to 2K processors of the Stampede mainframe demonstrate that the performance realized by our approach meets that of the more complicated lookahead variant. We also devised a task prioritization scheme to guide the task scheduling process to realize high performance. Our approach demonstrates the ability of custom translation to realize the benefits of complicated communication tolerant algorithm, using less complicated conventional algorithms.

ACKNOWLEDGMENT

This research was supported by the Advanced Scientific Computing Research, the U.S. Department of Energy, Office of Science, contracts No. DE-ER08-191010356-46564-95715 and DE-FC02-12ER26118. Tan Nguyen is a fellow of the Vietnam Education Foundation (VEF), cohort 2009, and was supported in part by the VEF. The authors acknowledge the Texas Advanced Computing Center (TACC) at the University of Texas at Austin for providing HPC resources that have contributed to the research results reported within this paper. Scott Baden dedicates his portion of this work to the memory of J. R. Bachman (1980-2015).

REFERENCES


