Communication Avoiding and Overlapping for Numerical Linear Algebra

Evangelos Georganas*[1], Jorge González-Domínguez†[1], Edgar Solomonik*,
Yili Zheng‡, Juan Touriño† and Katherine Yelick*‡
*Dept. of Electrical Engineering and Computer Sciences, University of California at Berkeley, Berkeley, CA 94720
†Department of Electronics and Systems, University of A Coruña, A Coruña, Spain
‡Lawrence Berkeley National Laboratory, Berkeley, CA 94720

***Abstract*—To efficiently scale dense linear algebra problems to future exascale systems, communication cost must be avoided or overlapped. Communication-avoiding 2.5D algorithms improve scalability by reducing inter-processor data transfer volume at the cost of extra memory usage. Communication overlap attempts to hide messaging latency by pipelining messages and overlapping with computational work. We study the interaction and compatibility of these two techniques for two matrix multiplication algorithms (Cannon and SUMMA), triangular solve, and Cholesky factorization. For each algorithm, we construct a detailed performance model which considers both critical path dependencies and idle time. We give novel implementations of 2.5D algorithms with overlap for each of these problems. Our software employs UPC, a partitioned global address space (PGAS) language that provides fast one-sided communication. We show communication avoidance and overlap provide a cumulative benefit as core counts scale, including results using over 24K cores of a Cray XE6 system.**

I. Introduction

Communication cost is a significant factor in application performance, and hardware trends indicate that the cost of data movement within and between nodes will continue to grow relative to the cost of computation. With exascale computing as the long-term goal, the community needs to develop techniques that minimize communication cost through better algorithms, programming techniques, systems software and architecture. In this paper, we consider two techniques to minimize the performance impact of communication: *communication-avoiding algorithms* reduce the volume of communication, while *communication overlapping* reduces the impact of each communication event by overlapping it with computation or with other communication [1], [2]. We explore a set of parallel communication avoiding algorithms for dense linear algebra that are provably optimal in their communication volume and have been shown to substantially reduce communication and thus improve performance [3]. They are particularly beneficial for strong scaling (computing the same total problem-size on more processors). These so-called “2.5D” algorithms trade-off extra memory and, in some cases, more messages in favor of reduced use of bandwidth. Communication overlap changes neither volume nor the number of messages, but lowers the per-message cost. To maximize the potential for overlap, we use one-sided communication as provided by the UPC language [4] and schedule the communication to be hidden whenever possible.

Communication avoidance and communication overlap may appear to be orthogonal optimization strategies, but there are complicated interactions and trade-offs if they are used together. With performance models incorporating both communication avoidance and overlap, we systematically study how to combine and balance these two techniques for various linear algebra algorithms with different problem sizes given the target system configurations.

We believe this is the first study that combines communication overlap with communication avoiding linear algebra algorithms, and does so at the scale of tens of thousands cores. Specifically, we made the following contributions:

- Developed communication avoiding and overlapping implementations for three popular linear algebra computations: matrix-multiplication, triangular solve and Cholesky factorization.
- Measured the performance of these algorithms on tens of thousands of processors, which are the first experimental results on 2.5D implementations of triangular solve and Cholesky factorization.
- Compared the performance of 4 versions of each algorithm, i.e., a 2D and 2.5D algorithm, with and without overlap.
- Provided a methodology for performance modeling of communication avoiding and overlapping algorithms.
- Analyzed the performance trade-offs of communication avoidance versus communication overlap, and their combination, under different situations.

In the following sections, we will first provide the necessary background for the classical (2D) and communication-avoiding (2.5D) algorithms, and then have more detailed discussions on applying communication overlapping techniques to these algorithms.

II. Background

We studied parallel algorithms for three problems: Matrix Multiplication (MM), Triangular Solve (TRSM) and Cholesky factorization. For MM, we compare Cannon's algorithm [5]

[1] Evangelos and Jorge contributed equally to this work and are listed alphabetically.

with a broadcast-based SUMMA [6]. Parallelization of TRSM and Cholesky are more complex due to the introduction of a longer critical path in the problem. We describe 2.5D algorithms for each of these problems [3]. This optimization attempts to avoid communication at the cost of a controlled increase in memory usage.

A. Matrix multiplication

Matrix multiplication of two square n-by-n matrices computes the product

$$C[i,j] = \sum_k A[i,k] \cdot B[k,j].$$

We study classical matrix multiplication, which computes all n^3 multiplications, though the reduced-complexity Strassen's algorithm [7] can also minimize communication and outperform the classical algorithm in a high-performance setting [8]. Since each of the n^3 multiplications can be done independently, parallelization is very simple to load balance and schedule.

Performing computation by blocks reduces the amount of data that must be moved between processors, as well as in a memory hierarchy. 2D algorithms distribute matrices in blocks among processes and communicate on a 2D grid of p processors. SUMMA performs communication with row and column broadcasts on this 2D grid. While SUMMA can flexibly be formulated in terms of rank-1 updates, to minimize messages it is best to block these updates into bundles of up to $n/\sqrt{p}$. Alternatively, these broadcasts can be done via all-gathers as implemented in the Elemental framework [9].

Cannon's algorithm performs shifts of data among near-neighbor processes on a 2D grid of p processors, with blocks of size $n/\sqrt{p}$-by-$n/\sqrt{p}$. The algorithm starts by performing a skew on the initial matrices along rows and columns of the processor grid. The blocks are lined up so that at each subsequent step, only a single shift needs to be done between each block multiplication. The advantage of Cannon's algorithm over SUMMA is that it can reduce the latency cost by using near-neighbor sends rather than collective communication. On the other hand, Cannon's algorithm is hard to generalize to non-square processors grids.

3D matrix multiplication [10], [11], [12], [13] is a different parallelization of the problem, which partitions the 3D computational graph rather than the matrices. This algorithm stores blocks of A, B and a temporary C matrix redundantly and performs independent updates on the copies of C, which are combined using a reduction operation at the end. Given sufficient memory, the 3D algorithm is provably optimal in both number of messages and amount of data communicated between processors. The effective block size becomes $n/p^{1/3}$. The 2.5D formulation of this algorithm controls the trade off between memory usage and communication by replicating the matrix to fill up as much extra memory as available. In particular, on p processors, the 3D algorithm stores $p^{1/3}$ matrix copies, while the 2.5D algorithm works for any number of copies $c \in [1, p^{1/3}]$. The 2.5D algorithm typically operates with a block size of $n/\sqrt{p/c}$. Algorithm 1 describes the 2.5D version of the SUMMA algorithm of a grid of processors Π with c layers of processors, each computing a different contribution to the matrix C. The 2.5D version of Cannon's algorithm, described in [3], also asymptotically minimizes both bandwidth and latency costs.

Algorithm 1: $[C]$ = 2.5D-SUMMA(A,B,Π,n,c)

```
Input: On a cuboid grid $\Pi$, $n$-by-$n$ matrix $A$ and $n$-by-$n$
       matrix $B$, are each spread over $\Pi[:,:,0]$
Output: square $n$-by-$n$ matrix $C = A \cdot B$ spread over
        $\Pi[:,:,0]$

Replicate $A$ and $B$ on each $\Pi[:,:,k]$, for $k \in [1,c]$

// Perform outer products on processor layers in parallel:
parallel for $k = 0$ to $k = c-1$ do
    // Each layer performs $n/c$ rank-1 updates:
    pipelined for $t = 1$ to $t = n/c$ do
        Broadcast $A[:, k \cdot n/c + t]$ on columns of $\Pi[:,:,k]$
        Broadcast $B[k \cdot n/c + t, :]$ on rows of $\Pi[:,:,k]$
        $C_k := C_k + A[:, k \cdot n/c + t] \cdot B[k \cdot n/c + t, :]$
    end
end
// Compute C via a sum reduction:
$C := \sum_{k=0}^{c-1} C_k$
```

B. Triangular solve

Triangular solve (TRSM) is used to compute a matrix X, such that $X \cdot U = B$, where U is upper-triangular and B is a dense matrix. This problem has dependencies across the columns of X, while each row of X can be computed independently. For n-by-n matrices, solving this problem has the same asymptotic computational cost as matrix multiplication. The problem has also been shown to require at least as much communication as matrix multiplication [14].

TRSM can be parallelized on a 2D processor grid with broadcasts or all-gathers used for communication between processors, similar to SUMMA. At each step of the 2D algorithm, a block-column of X is computed. This block-column can then be used to update the trailing matrix B. In particular, if we consider partitioning the matrices in blocks

$$\begin{bmatrix} X_1 & X_2 \end{bmatrix} \cdot \begin{bmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{bmatrix} = \begin{bmatrix} B_1 & B_2 \end{bmatrix},$$

the computation proceeds as follows,

1) compute column via TRSM, $X_1 = B_1 \cdot U_{11}^{-1}$,
2) compute update via product, $B_2 = B_2 - X_1 \cdot U_{12}$,
3) compute next column via TRSM, $X_2 = B_2 \cdot U_{22}^{-1}$.

If the matrices are subdivided in blocks among processes, it is necessary to communicate the sub-matrices U_{11} and later U_{22} along columns of the process grid, and to communicate the matrices X_1 and U_{12} when performing the update.

Until X_1 is computed, processes on the right portion of the processor grid are not occupied. Therefore, X_1 is typically

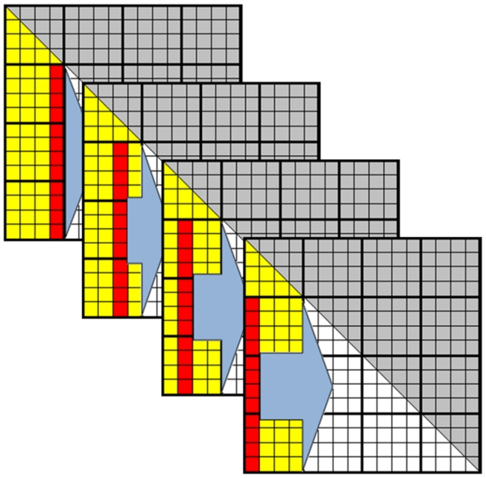

Fig. 1. 2.5D Cholesky 'fat panel' update broadcast.

taken to be a thin panel of the matrix, which yields significantly better load-balance. Communication is minimized by employing a block-cyclic layout, where each process owns multiple sub-blocks of the matrices. In particular, each process $\Pi[i,j]$ owns blocks $\{X,U,B\}_{kl}$ where $k = i + r\sqrt{p}$ and $l = j + q\sqrt{p}$ for each r,q. The communication of X_1 and U_{12}, needed to perform each update, becomes the dominant communication cost in this distribution.

2.5D TRSM lowers the bandwidth cost with respect to the 2D algorithm, when additional memory is available [3]. The 2.5D algorithm employs an additional level of block hierarchy, by performing the update in two stages. The matrices are partitioned into $O(c)$ 'fat' panels. Each fat panel of X is computed with the entire processor grid, arranged in a 2D rectangular layout of $\sqrt{pc}$-by-$\sqrt{p/c}$ processors. After each one of these 2D TRSMs, the rest of the 'fat' panels are updated. This big update is done as c independent outer-products, one on each layer. The update is accumulated redundantly and combined via reductions by panel. For more details on the data layout and communication in this algorithm see [3].

C. Cholesky factorization

Cholesky factorization computes the factorization of a symmetric positive definite matrix A, into product of a lower triangular matrix and its transpose, $A = L \cdot L^T$. The sequential algorithm performs a modified version of Gaussian Elimination. This algorithm has a dependency path across both columns and rows of L. So, parallelization of this algorithm must efficiently handle symmetry as well as load balance with consideration of both dependency paths.

We recall the 2D parallel algorithm for Cholesky by considering the factorization in blocks,

$$\begin{bmatrix} A_{11} & A_{21}^T \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \cdot \begin{bmatrix} L_{11}^T & L_{21}^T \\ 0 & L_{22}^T \end{bmatrix}.$$

We can start in the top left corner and compute L as follows,

1) factorize block via Cholesky, $A_{11} = L_{11} \cdot L_{11}^T$,
2) update panel via TRSM, $L_{21} = A_{21} \cdot L_{11}^{-T}$,
3) compute update via product, $A_{22} = A_{22} - L_{21} \cdot L_{21}^T$,
4) compute next block via Cholesky, $A_{22} = L_{22} \cdot L_{22}^T$.

Communication is required to send L_{11} across process columns, L_{21} across process rows and L_{21}^T across process columns. Thus, the parallelization is very similar to TRSM, except that the sequential factorization needed to compute L_{11} must be done before the panel TRSM. Further, less computation is involved in the symmetric update, while communication volume stays just as large.

2.5D Cholesky is decomposed hierarchically in the same fashion as 2.5D TRSM. Figure 1 demonstrates how updates are decomposed among processor layers and done on different matrix copies. Again, these updates are accumulated redundantly, which uses extra memory, but reduces the bandwidth cost of the algorithm. A block-cyclic layout is again necessary to achieve load-balance and minimize latency. Though, we note that the 2.5D parallelization actually slightly increases latency cost, which is necessary to reduce bandwidth [3].

III. Experimental platform

Before starting the discussion on applying communication overlapping techniques to the aforementioned linear algebra algorithms, we describe the hardware and software environments for our implementation because they are helpful to understand some of the design decisions we choose. While our ideas are generally applicable to all systems, we take a step further to optimize the tunable parameters specifically for our experimental platform.

A. Hardware

Our target system is a Cray XE6 supercomputer with 153,216 compute cores and 217 TB of memory in total. Each Cray XE6 node has 24 cores, grouped by 6 in 4 Non-Uniform Memory Access (NUMA) domains. CPU cores have faster access speed to local memory within the same NUMA domain but have slower access speed to remote memory in other NUMA domains. Inter-node communication is done through the custom Cray Gemini Network, which is a high-bandwidth and low-latency 3-D torus interconnect with hardware RDMA support. Table I lists the specifications of our experimental system. Cray XK-6, a sister model of Cray XE6 with GPUs, also uses the same Gemini interconnect. Therefore, we expect our techniques and software developed to be beneficial to a large class of many-petaflop systems important to the broad supercomputing community.

B. Software

We choose Unified Parallel C (UPC) [4] , a PGAS extension of C99, to implement our applications. Modern networks, such as the Gemini interconnect in Cray XE6, have special hardware support for RDMA offloading that facilitates communication overlapping. Shan et al. [15] demonstrated that PGAS languages, such as UPC and CAF, can deliver substantially better performance for non-blocking point-to-point communication over MPI on Cray XE6 because Gemini allows remote-node references to be pipelined in these programming models.

System	Cray XE6
Processor	AMD Opteron “Magny-Cours”
Clock rate	2.1 GHz
Peak performance per core	8.4 Gflops
Cores per NUMA domain	6
NUMA domains per node	4 (packaged in 2 sockets)
Total cores per node	24
Private L1 data cache per core	64 KB
Private L2 data cache per core	512 KB
Shared L3 cache per NUMA domain	6 MB
Memory bandwidth	25.6 GB/s
Memory per node	32 GB DDR3-1066 ECC
Compiler	Cray Compiler
Interconnect	Gemini 3-D Torus
Peak Bandwidth (per direction)	7 GB/s

TABLE I
SPECIFICATIONS OF THE CRAY XE6 USED FOR OUR EXPERIMENTS

In addition, the global address space of UPC improves our programming productivity for implementing global matrices with complex blocked data-layouts.

As in many scientific applications, the communication patterns in the algorithms that we studied can be conveniently expressed by collective communication, which is usually optimized for the hardware platform by the vendor. The collectives in the current UPC 1.2 specification don’t meet our needs because: 1) they don’t support collectives within a subset of threads (i.e., no equivalent of MPI communicators); 2) they lack some of the necessary collective operations such as `reduce` and `allreduce` for arrays. Thus we use MPI collectives to supplement UPC in our implementation.

Though hardware RDMA can transfer data without CPU intervention, it usually requires non-trivial amount of CPU cycles for initialization. Therefore, hardware RDMA is more suitable for long messages, whose cutoff size is system-specific. For short messages, the best latency is attained by attentive CPU polling. Collective communication often requires even more CPU resources to proceed because they usually use more complicated algorithms that cannot be simply offloaded to the network. Thus we use a dedicated Pthread to handle communication progress when overlapping collective communication with computation.

We use a hybrid (process/thread) parallelization model for all our benchmark runs. Specifically, we run one process per NUMA domain and one thread per core within the NUMA domain. Each UPC thread is mapped an OS process and each process uses 6 OS threads, 1 thread per core, via Pthreads and multi-threaded BLAS/LAPACK libraries. We implement our parallel algorithms using local linear algebra routines provided by vendor-optimized libraries.

C. Performance characteristics

To guide our optimization with performance modeling, we develop a suite of micro-benchmarks to measure the system parameters of the target computer. In addition, these performance numbers can also tell us the practical efficiency upper bound of the respective operations.

1) Computation efficiency: Figure 2 shows the efficiency of the linear algebra routines used in our implementations for different matrix sizes. For example, `dgemm` attains peak efficiency at 8MB matrix size (a 1024-by-1024 matrix of `double` type).

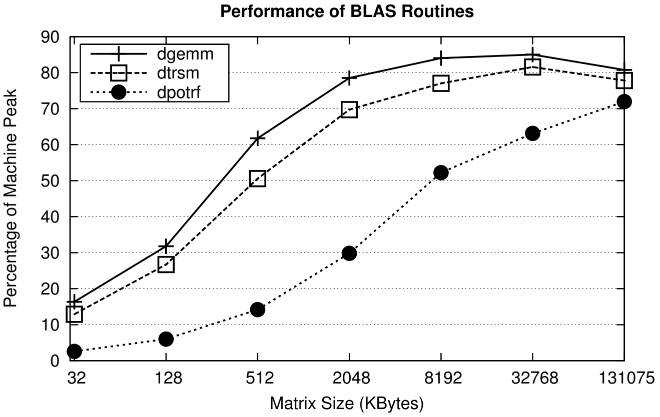

Fig. 2. Efficiency of the BLAS routines used in our algorithms. Run with 6 cores in a NUMA domain.

2) Network performance: We use the methodology in [16], [17], [18] to measure the LogGP parameters of the interconnect. Figure 3 shows the inter-node network bandwidth for different message sizes. The network bandwidth peaks at 512KB messages.

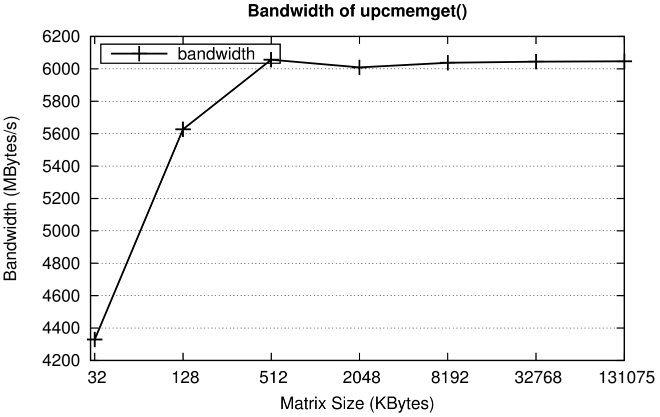

Fig. 3. Inter-node network bandwidth using UPC one-sided communication

IV. SUMMA

A. Overlapping communication and computation

As explained in Section II-A, the SUMMA algorithm performs the matrix product based on broadcasts and outer products. The main part of the 2D and 2.5D algorithm consists of a loop where each thread takes part in two broadcasts (related to the grid row and column, respectively) and performs one BLAS matrix product (*dgemm*) per iteration. We develop new versions of both the 2D and 2.5D algorithms that overlap communication and computation by starting the broadcast of the next iteration while the *dgemm* of the current iteration is being performed. Most of UPC compilers provide support for overlapping through asynchronous *upc_memget*, *upc_memput* and *upc_memcpy* functions. However, they are limited to point-to-point communications. In order to overlap the broadcast we create one Pthread that performs the part of this broadcast that corresponds to that UPC thread. Thus, one core of the NUMA region is only used for communication and, at the same time, the other five cores are used for computation by calling the multithreaded BLAS routines.

B. Implementation details

Despite the fact that UPC supports distributed arrays with blocked and cyclic layouts, it does not support blocking in multiple dimensions which is necessary for implementing the 2D block and block-cyclic layouts. Therefore, instead of using the UPC distributed arrays directly, we exploit the global address space to build a distributed data structure called *directory*. This data structure uses global pointers, which can refer to memory associated with other UPC threads. After initializing this data structure with the appropriate pointers, each UPC thread makes a local copy of the structure. Then, every time it needs to access memory belonging to another UPC thread, it looks up the local directory and finds the corresponding global pointer. This lookup is fast since it involves local data. Finally, the UPC thread uses the obtained global pointer to access the data. We found these directories convenient for a few reasons. First, they facilitate the indexing over layers in the 2.5D algorithms. To incorporate the notion of layers into the dictionary, it is sufficient to add an auxiliary dimension to the data structure. Second, adding blocking levels into our data distributions degenerates into modifying some tunable parameters in the data structure. This was helpful for the 2.5D algorithms where we deploy a two-level 2D block cyclic layout and this additional level of blocking is substantial in controlling load balance and latency.

The memory requirements of each algorithm and the overheads included by their optimizations should be taken into account in order to find the most suitable option for each scenario. In 2D SUMMA, each UPC thread must keep one square block of size $n/\sqrt{p}$-by-$n/\sqrt{p}$ per matrix. Additionally, two buffers of the same size are necessary to keep the data of the blocks of the input matrices that must be multiplied during each iteration. Thus, they need $5n^2/p$ elements. The number of blocks that must be kept at the same time in the 2.5D algorithm are the same (3 blocks for the original matrices and 2 for the current iteration). The only difference is that their dimension is $n/\sqrt{p/c}$ and the memory requirements $5n^2c/p$ elements (overhead of c).

When applying overlapping, we need to double the buffer size in order to keep, at the same time, the blocks of the current iteration that are being multiplied and the blocks needed in the next iteration whose broadcast is being forwarded. Thus the memory requirements are increased in $2n^2/p$ and $2n^2c/p$ elements because of using overlapping in the 2D and 2.5D algorithms, respectively.

C. Experimental results

The percentage of peak flops of the target system described in Section III obtained by each SUMMA algorithm are illustrated in the graphs of Figure 4. These results were taken using double precision from 1,536 to 24,576 cores (from 256 to 4,096 UPC threads). Results for the 2.5D algorithms were obtained using different values for the replication factor c, showing always the best (in these scenarios $c = 4$). On 1,536 cores, for a matrix size of 32,768-by-32,768 doubles, replication (2.5D algorithms) could not be done due to insufficient memory.

First, these results prove that communication avoiding and overlapping techniques are complementary as the version that combines them obtains the best performance for a large number of cores (even double of performance than the basic 2D version for 24,576 cores). Comparing both techniques, the more cores we use the more significant is the improvement obtained by the 2.5D algorithms. However, they are not very beneficial for 1,536 cores, and the overlapped 2D version is the best option. The reason is that, although the time due to the broadcasts of the loop decreases, the overhead introduced by the initial replication and the final reduction of the matrices across layers offsets the reduced intra-layer communication. Thus, we see that the overlapping technique is more beneficial for a low/medium number of cores while communication avoidance has more influence while we increase this number, although the two techniques can be combined.

V. Cannon's algorithm

A. Overlapping communication and computation

The structure of Cannon's algorithm is quite similar to that of SUMMA. Apart from the initial skew of the matrices, the main difference is that in each iteration the threads only need to shift the blocks of the input matrices by rows/columns of the grid. Similarly to SUMMA, the Cannon's 2D and 2.5D algorithms have been optimized by forwarding the shift of the blocks needed in the next iteration and overlapping it with the BLAS matrix product of the current iteration. Two options arise for this overlapping: using Pthreads as in SUMMA or the asynchronous *upc_memget* functions available in the compilers. We have developed a benchmark that compares the overlapping capabilities of each approach and determined that the usage of asynchronous copies is better than Pthreads, especially as the computation costs can also be divided among the 6 cores of the NUMA region. Therefore, overlapping in Cannon's is implemented using asynchronous *upc_memget* functions.

B. Implementation details

In the 2D algorithm, instead of synchronizing all threads at each iteration so that each of them takes the data from the next one, we use one buffer so that each UPC thread can store two blocks of each matrix at the same time. The blocks used in the iteration $i + 1$ will be kept in the part of the memory where the blocks used in $i - 1$ were stored. Therefore, the blocks that must be shifted to another UPC thread (the ones used

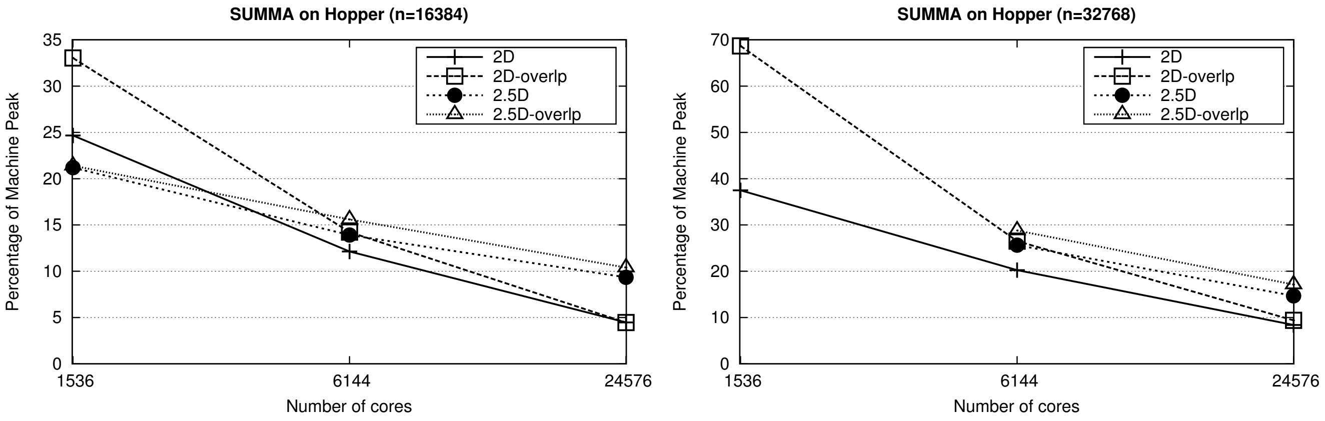

Fig. 4. Performance of the SUMMA algorithm

for computation in iteration i) are not corrupted because of copying the next ones. The shifts are made by the the receiving UPC thread using *upc_memget*.

The memory requirements of Cannon's algorithm are also quite similar to those of SUMMA. The block sizes are again $n/\sqrt{p}$ and $n/\sqrt{p/c}$ in the 2D and 2.5D cases, respectively. Since one buffer is needed to shift the blocks of the input matrices the memory requirements are $5n^2/p$ and $5n^2c/p$ too. Thus, again the memory overhead because of using the 2.5D version is indicated by c. The overlapping technique only needs memory to store two blocks of each input matrix so it does not lead to any memory overhead.

C. Experimental results

A study of the performance of the different versions of Cannon provides two insights. On the one hand, the observed performance is better than SUMMA (for instance, around 25% for 24,576 cores). The reason is that the latency cost is reduced because of using point-to-point communications with *upc_memget* instead of collectives. This issue is even more significant in PGAS languages with one-sided communications because the source and destination threads do not need to be synchronized. On the other hand, the conclusions of the comparison among the different approaches are quite similar to SUMMA, in spite of using a different technique to apply overlapping (asynchronous *upc_memget* instead of Pthreads): communication avoidance seems to be more beneficial for a large number of cores while overlap help more at a smaller scale.

Figure 5 shows the performance breakdown using 1,536 and 24,576 cores for a 16,384x16,384 matrix in order to help us to explain the reasons for this behavior. The communication label represents the time of the point-to-point copies within the loop. The initial shift also includes the broadcast along the layers used in the 2.5D approaches. When using 1,536 cores, the communication and computation times of the 2D algorithm are very similar, so the overlapping technique improves the performance significantly. The 2.5D approach significantly reduces the communication time, but not enough to compensate for the overhead introduced by the broadcast of the initial shift and the final reduction. Furthermore, its combination with overlap is not especially worthy as communication and computation times are not balanced. However, for 24,576 cores, the difference between the communication and computation times in the 2D algorithm limits the influence of the overlapping technique and the application of the communication avoidance is worthy even with the overheads (especially combined to overlapping communication and computation as their execution times are similar).

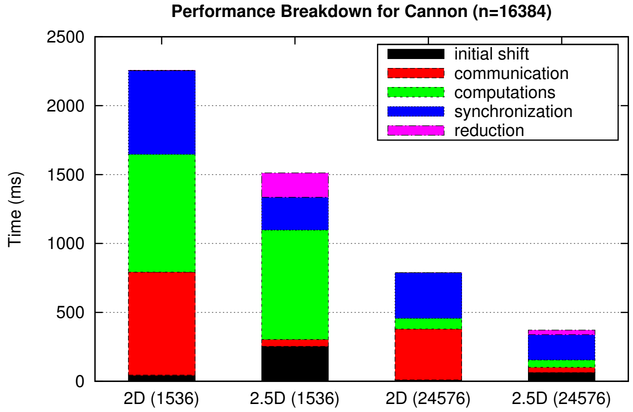

Fig. 5. Performance breakdown of the Cannon algorithm using 1,536 and 24,576 cores for a 16,384x16,384 matrix

VI. Triangular solve

A. Overlapping communication and computation

The main bottleneck of the 2D triangular solve is the update of the matrix using *dgemm* (matrix multiplication): $B_2 = B_2 - X_1 \cdot U_{12}$. Since, in this case, the block U_{12} is always available to be broadcast along the columns of the grid, we opted to overlap this update with the broadcast of the blocks of U needed in the *dgemm* call of the following iteration, using the Pthreads mechanism also applied in SUMMA.

B. Implementation details

The 2D TRSM has been described in Section II-B. However, we develop a different 2.5D approach to the one explained in [3]. This new approach follows the same scheme as 2.5D algorithms (using c layers if there is enough memory) but without minimizing the total amount of communication. The communication times just decrease because we reduce the number of cores involved in each broadcast and, thus, the contention of the network. This algorithm takes advantage of the rows of X that can be computed independently to avoid the reductions and, in general, minimizes the communication and synchronization among layers. The initial distribution of the matrices in one layer is the same as for the 2D algorithm, a block-cyclic distribution along the $\sqrt{p/c}$ threads in each dimension. Thus, if each UPC thread has r blocks in each dimension, the block size is $n/r\sqrt{p/c}$-by-$n/r\sqrt{p/c}$. In the beginning, we replicate the triangular matrix along layers (as input matrices in the 2.5D algorithms for the matrix product) but distribute the rows of each block of X among them. We remark that there are two levels of blocking for X, each UPC thread has r^2 rectangular blocks of $n/cr\sqrt{p/c}$-by-$n/r\sqrt{p/c}$ elements. With this distribution, each layer computes a subset of the rows of X via a 2D TRSM with its $\sqrt{p/c}$-by-$\sqrt{p/c}$ grid of threads. The distributed rows must be gathered once the layers finish their computation.

In this new 2.5D TRSM, most of the computation is completely independent among layers because only one synchronization is necessary to know the moment when all them have finished and the final gather must be performed. The main advantage of this approach over the algorithm shown in [3] is that it replaces the c reductions across layers with only one final gather. Besides, it keeps the advantage of reducing the communication times with respect to the 2D approach because the broadcasts within one layer involve less UPC threads and are faster. The 2.5D algorithm can also overlap the matrix update of the current iteration with the broadcast of the blocks of U needed in the next one.

With regard to the memory requirements we remark that in this problem there are two matrices of different type: X, which is a dense matrix and U, which is a triangular matrix where only the elements above the main diagonal are stored. In the 2D algorithm, where r is the factor that indicates the number of blocks per thread in each dimension in the block-cyclic distribution, the block size is $n/r\sqrt{p}$-by-$n/r\sqrt{p}$. In the dense matrix X, each thread owns $n/\sqrt{p}$ rows and columns so they must store n^2/p elements of this matrix. Nevertheless, only blocks above or in the main diagonal of matrix U are stored so UPC threads do not need the same amount of memory. We consider the UPC threads with the largest requirements which are $r(r+1)/2$ blocks. It means that $n^2(r+1)/2rp$ elements of the initial triangular matrix must be stored in the worst case. Furthermore, additional buffers are needed to perform the broadcasts (one column block of X and one row block of U). Simplifying, the memory requirements of the 2D TRSM are $n^2(r^2+5r+2)/2pr^2$ elements.

In the 2.5D algorithm the initial matrices are distributed in a 2D block-cyclic way among the UPC threads within the first layer. Thus, these threads will need the same amount of blocks but with size $n/r\sqrt{p/c}$-by-$n/r\sqrt{p/c}$. As the number of blocks to store in the buffers is also the same as in the 2D algorithm, the memory requirements are multiplied by c.

The overlapping technique for TRSM only increases the memory requirements because of an additional buffer needed to store the data of U that will be used in the next iteration update. Thus, the 2D and 2.5D algorithms need n^2/rp and n^2/rpc more elements, respectively.

C. Experimental results

The experimental evaluation of TRSM is shown in Figure 6. The matrix size is up to 65,536-by-65,536 doubles because of needing only two matrices. As in the previous algorithms, the 2.5D algorithm with overlap is the best choice for almost all the scenarios. Studying both techniques independently, the minimization of contention reduces the communication time more significantly than the overlapped 2D approach when using a large number of cores. Nevertheless, the best option for the smallest scenario (1,536 cores and matrices of 32,768-by-32,768 doubles) is again the overlapped 2D algorithm.

VII. Cholesky Factorization

A. Overlapping communication and computation in Cholesky

As in the triangular solve, the bottleneck of the 2D Cholesky factorization is the update of the trailing matrix $A_{22} = A_{22} - L_{21} \cdot L_{21}^T$ (described in Section II-C). This update can be decomposed in two phases: first, the required sub-blocks of L_{21} and L_{21}^T are received via two consecutive broadcasts and then *dgemm* operations follow to update the corresponding blocks of A_{22}. However we should emphasize that the dependencies of Cholesky differ substantially from those of triangular solve. In this case the dependencies are across columns and rows of the original matrix. Furthermore, the critical path includes the factorization of the diagonal matrix blocks in a sequential way.

Due to the aforementioned dependencies, we opted to overlap computation and communication in the update of trailing matrix. These operations are illustrated in Figure 7. Initially we factorize the first column of blocks and we execute two back to back broadcasts to transfer it to the involved UPC threads. Note that we store the received blocks in auxiliary buffers. Then, instead of updating the whole trailing matrix, we update only the second column of blocks with *dgemms* and continue with its factorization. This slight modification creates a pipeline and provides an overlapping opportunity at the next step. While we broadcast the second column of factorized blocks, we can simultaneously update the rest of the trailing matrix using the buffered first column of factorized blocks. This part of the trailing matrix is depicted with yellow color in Figure 7 and the two back to back broadcasts are depicted with blue arrows. As soon as the two overlapped operations synchronize, we update with *dgemms* only the third column of blocks in respect to the second factorized column (which is

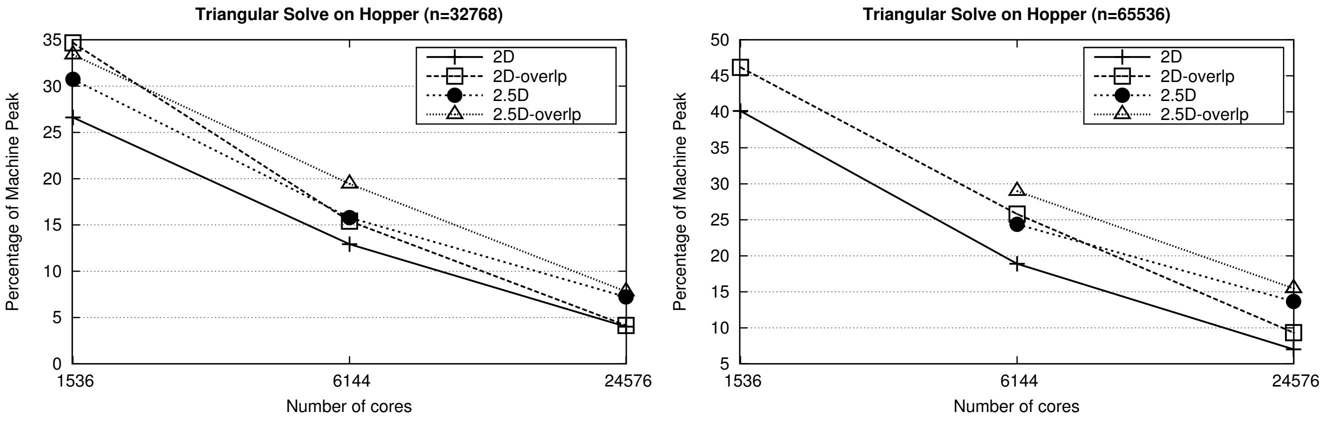

Fig. 6. Performance of the triangular solver

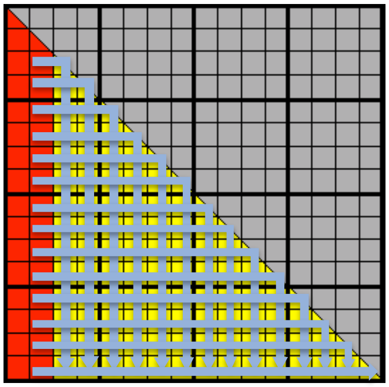

Fig. 7. Overlapping computations and communication in 2D Cholesky

just received), then we factorize it and the same overlapping schedule remains active until the whole matrix is factorized.

The memory overhead for this overlapping schedule consists of the auxiliary buffers for two columns of blocks (one buffer for the blocks received from the row broadcast and one buffer for the blocks received during the column broadcast). In order to fill the pipeline and to initiate the overlapping schema, only two broadcasts are required. In other words, all except two of the consecutive broadcasts are overlapped with computations. This kind of overlapping hides both the bandwidth/latency costs of the two broadcasts and also the idle times resulting from them.

The 2.5D algorithm decomposes the updates of the trailing matrix on different layers as it is depicted in Figure 1. So each layer is responsible for updating its matrix using a sub-panel of columns (the red ones in Figure 1). Since we are implementing a two level blocking as previously described, each red sub-panel consists of multiple columns of blocks. Thus we can adopt a similar overlapping schema as in the 2D algorithm. Initially we broadcast the first of these columns and we store the received blocks in auxiliary buffers. At the next step, we update the trailing matrix with *dgemms* using the buffered blocks and simultaneously broadcast the second of the columns. Again the memory overhead due to overlapping is the space needed for two auxiliary column-buffers.

B. *Implementation details*

Since the input matrix A is symmetric, we must store only the elements lying on and below the diagonal. So, in Figure 1 the gray blocks are not stored, instead their values can be obtained explicitly by transposing their symmetric blocks. Thus, the transfer of L_{21} across rows and L_{21}^T across columns that are mentioned in Section II-C, are implemented as two consecutive team broadcasts: During the second broadcast, the UPC threads lying on the diagonal send to their column-team the blocks received from their row-sender during the first broadcast. Finally, we emphasize here that the 2.5D algorithm deploys a two level blocking: On a first level, we block the matrix into "fat panels" on each layer and the result of this blocking is shown in Figure 1. At a next level we further block each "fat panel". This two level blocking is crucial for achieving load-balancing and minimizing latency. Moreover, 2.5D Cholesky proceeds in logical steps where it first factorizes a column of "fat panels" and then updates the trailing matrix. To take significant advantage from the 2.5D algorithm we should have as many "fat panels" as possible in our matrix decomposition. In this case, we will spend more time on the updates of trailing matrices, which can be done at high computational efficiency.

Regarding the exact memory requirements, the 2D algorithm has to store $n^2/2$ elements (i.e. half of the matrix A) and since we utilize p UPC threads, each thread stores $n^2/(2p)$ elements. If we apply the 2D block-cyclic layout r times in each dimension, each thread allocates two communication buffers with total size $2n^2/(pr)$ elements. This means that the aggregate memory overhead for the 2D algorithm with overlapping consists of $2n^2/r$ elements. Finally, in the 2.5D algorithm we store c replicas of the initial matrix and on each replica we assign p/c threads. Hence, all the memory requirements are multiplied by a factor of c for 2.5D and 2.5D with overlapping respectively.

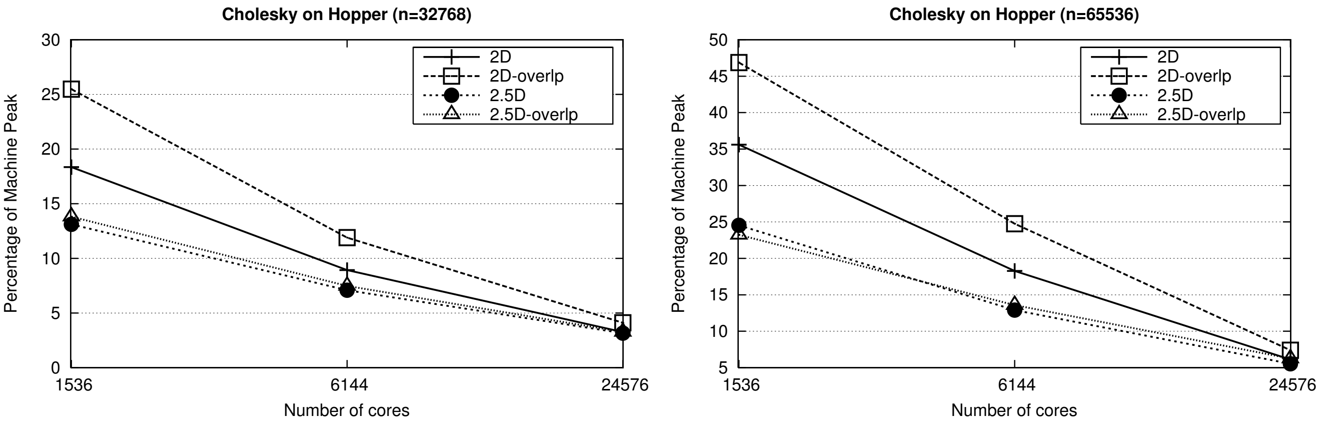

Fig. 8. Performance of Cholesky

C. Experimental results

The experimental results for Cholesky factorization are shown in Figure 8. We conclude that overlap helps improve the performance of both 2D and 2.5D algorithms. However, 2.5D does not incur as much benefit from overlap as 2D does because of the "fat panel" factorization phase, where no overlap is exploited by our implementation. If we examine the two optimization techniques independently, we observe that overlap gives significant boost to the 2D Cholesky factorization. On the other hand, communication avoidance is not beneficial up to 6,144 cores, while on 24,576 cores the 2.5D version meets the performance of the 2D algorithm. As explained in the previous section, we want as many "fat panels" as possible in the 2.5D case and this can lead to very small block sizes. The latter fact implies reduced BLAS efficiency which in turn increases both computation and idle times[1]. For this reason, we expect the 2.5D algorithm to give a significant performance boost when the input matrix is even larger. Finally, from the given strong scaling graphs one can infer that overlapping is beneficial in weak scaling too. In theory, the ratio of computation/communication remains the same as we weak-scale, so overlapping should provide equivalent gains. For 1,536 cores and n=32,768 overlapping decreases the execution time of the 2D version by 24% and similarly for 6,144 cores and n=65,536 (i.e. $4\times$ cores and $4\times$ problem size) the decrease in execution time via overlapping is 26%.

VIII. Methodology for constructing performance models

In this section we describe the general methodology of developing detailed performance models for the aforementioned algorithms. Our ultimate objective is to maximize the performance for any given problem, so we opt to design detailed performance models that indicate the algorithm to execute and, additionally, provide an auto-tuning framework for the tunable parameters among the various algorithms.

Our performance models track the execution flow of each algorithm and estimate the completion time for every encountered operation, whether it is a BLAS call or a communication operation. In regard to the overlapped operations, the models predict the execution time as the maximum expected completion time of each individual operation. So, the models measure the execution time in the critical path taking into account the computation time, possible idle time due to load imbalance, and the communication time. To make accurate estimates for the BLAS and communication calls, the models employ input arguments which are machine and algorithm dependent.

First of all, a problem set-up is defined by the matrix dimension n and the available number of processors p. The algorithms that leverage a 2D block cyclic distribution for the matrix partitioning have an extra parameter r that defines how many times we apply this 2D block cyclic distribution on the input matrix (i.e. if the distribution is simply pure block 2D then $r = 1$). Finally, the 2.5D algorithms have a parameter c that specifies the replication factor. For Cholesky and triangular solve, a 2D block cyclic distribution is applied on each replicated layer since we have implemented a two level blocking as it is described in Sections VI and VII. So, given the parameters n, p, r, c we can find out the block size that each BLAS call operates on and we can specify the exact number of processors and words involved in each communication operation.

All the previous parameters are platform independent. We should somehow support the computation and communication estimations with some additional machine dependent parameters. Specifically, we utilize the following multithreaded BLAS routines: *dgemm*, *dpotrf*, and *dtrsm*. We run micro-benchmarks on the target platform to get the efficiency of each routine as a function of the input matrix size. The corresponding results for the multithreaded BLAS routines with six threads on Hopper are illustrated in Figure 2. From these results we can make an accurate prediction about the execution time of each BLAS call since we know the exact block size of the inputs.

[1] An optimization we are working on is to aggregate the BLAS operations into larger rectangular blocks so that we can avoid the side-effect of reduced BLAS efficiency

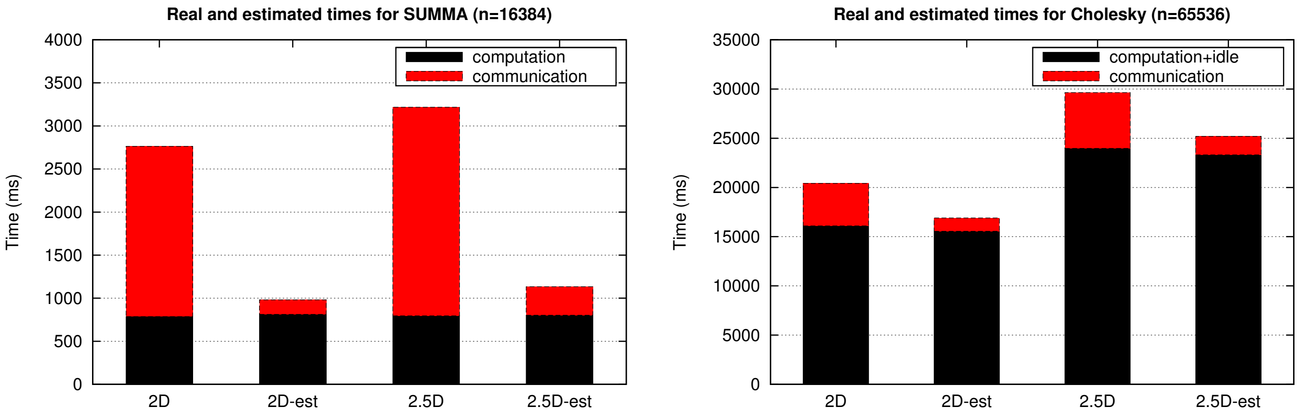

Fig. 9. Comparison of the real and the estimated performance for SUMMA and Cholesky in experiments with 1,536 cores

Furthermore, we model each point-to-point communication based on the number of the transferred words, the network bandwidth and the latency. The last two parameters depend on the message size, thus we benchmark the network performance using the LogGP model as explained in Section III-C. The model for the collective operations uses the same parameters, the number of participating processes and a binomial function.

An overview of the accuracy of our performance models is shown in Figure 9. We use SUMMA and Cholesky with 1,536 cores as examples. Our methodology provides models that predict correctly the computation and idle times. However, their estimations of the communication times are very optimistic. The reason is that the contention of the network significantly influences the performance of the algorithms but it is not included in the LogGP model. Via the inclusion of the contention in the models, they could potentially estimate correctly the performance of the algorithms. We refer the reader to [19] for the exact equations resulting from this methodology.

IX. Conclusion

Minimizing communication cost in parallel algorithms is key to improving performance for current and future supercomputers. Through the case studies of four numerical algorithms, we have demonstrated that combining communication avoiding and overlapping techniques is effective and can significantly reduce execution time in most cases. The priority of applying these two techniques depends on the algorithm, the problem size and the machine configuration. The trade-offs between communication avoidance and overlap include computation cost, communication cost, and memory usage. We developed performance models that could be used to predict the benefits of these optimization techniques but we also found that estimating communication time precisely was very difficult in the presence of unpredictable network contention.

In addition, we have a couple of interesting observations about parallel programming models in our research: 1) Hardware RDMA support in modern interconnects enables very efficient communication overlapping with PGAS languages such as UPC in our case. 2) Our implementations employed three different parallel programming models (UPC, MPI and Pthreads) to meet our algorithmic needs and fully realize the hardware potential. However, this is unfavorable in terms of programming productivity and indicates that more research in parallel programming is still much needed.

While we have successfully showed how to use communication avoiding and overlapping techniques to improve performance for a few linear algebra problems, many research questions remain, for example: 1) Can we automate the process of applying these optimizations and selecting the right parameters by using performance models and empirical search? 2) Can we extend communication avoidance and overlap to other interesting problems, especially those of irregular parallelism? and 3) What new hardware features can assist these two techniques? We hope our work will inspire more future work in the quest of minimizing communication cost.

Acknowledgment

This research was supported in part by the Office of Science of the U.S. Department of Energy (DE-AC02-05CH11231), DARPA (HR0011-10-9-0008), the Ministry of Science and Innovation of Spain (TIN2010-16735) and the Ministry of Education of Spain under the FPU research grant AP2008-01578. The third author was supported by a Krell Department of Energy Computational Science Graduate Fellowship, grant number DE-FG02-97ER25308. This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

References

[1] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick, “Optimizing bandwidth limited problems using one-sided communication and overlap,” in *Proc. 20th Intl. Parallel and Distributed Processing Symposium (IPDPS’06)*, 2006.

[2] R. Nishtala, P. H. Hargrove, D. O. Bonachea, and K. Yelick, “Scaling communication-intensive applications on BlueGene/P using one-sided communication and overlap,” in *Proc. 23rd Intl. Parallel and Distributed Processing Symposium (IPDPS’09)*, 2009.

[3] E. Solomonik and J. Demmel, “2.5D algorithms for parallel dense linear algebra,” *Concurrency and Computation: Practice and Experience*, 2012, to appear.

[4] “UPC language specifications, v1.2,” Lawrence Berkeley National Lab, Tech. Rep. LBNL-59208, 2005.

[5] L. E. Cannon, “A cellular computer to implement the kalman filter algorithm,” Ph.D. dissertation, Montana State University, 1969.

[6] R. van de Geijn and J. Watts, “SUMMA: scalable universal matrix multiplication algorithm,” *Concurrency and Computation: Practice and Experience*, vol. 9, no. 4, pp. 255–274, 1997.

[7] V. Strassen, “Gaussian elimination is not optimal,” *Numerische Mathematik*, vol. 13, pp. 354–356, 1969.

[8] G. Ballard, J. Demmel, O. Holtz, B. Lipshitz, and O. Schwartz, “Communication-optimal parallel algorithm for Strassen’s matrix multiplication,” UC Berkeley, Tech. Rep. EECS-2012-32, 2012.

[9] J. Poulson, B. Maker, J. R. Hammond, N. A. Romero, and R. van de Geijn, “Elemental: A new framework for distributed memory dense matrix computations,” *ACM Transactions on Mathematical Software*, 2012, to appear.

[10] E. Dekel, D. Nassimi, and S. Sahni, “Parallel matrix and graph algorithms,” *SIAM Journal on Computing*, vol. 10, no. 4, pp. 657–675, 1981.

[11] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar, “A three-dimensional approach to parallel matrix multiplication,” *IBM J. Res. Dev.*, vol. 39, pp. 575–582, 1995.

[12] A. Aggarwal, A. K. Chandra, and M. Snir, “Communication complexity of PRAMs,” *Theoretical Computer Science*, vol. 71, no. 1, pp. 3–28, 1990.

[13] S. L. Johnsson, “Minimizing the communication time for matrix multiplication on multiprocessors,” *Parallel Computing*, vol. 19, pp. 1235–1257, 1993.

[14] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz, “Minimizing communication in linear algebra,” *SIAM J. Mat. Anal. Appl.*, vol. 32, no. 3, 2011.

[15] H. Shan, N. J. Wright, J. Shalf, K. Yelick, M. Wagner, and N. Wichmann, “A preliminary evaluation of the hardware acceleration of the Cray Gemini interconnect for PGAS languages and comparison with MPI,” in *Proc. 2nd Intl. Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems (PMBS’11)*, 2011, pp. 13–14.

[16] C. Bell, D. Bonachea, Y. Cote, J. Duell, P. Hargrove, P. Husbands, C. Iancu, M. Welcome, and K. Yelick, “An evaluation of current high-performance networks,” in *Proc. 17th Intl. Parallel and Distributed Processing Symposium (IPDPS’03)*, 2003.

[17] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken, “LogP: Towards a realistic model of parallel computation,” in *Proc. 4th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 1993, pp. 1–12.

[18] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman, “LogGP: Incorporating long messages into the LogP model,” *Journal of Parallel and Distributed Computing*, vol. 44, no. 1, pp. 71–79, 1997.

[19] E. Georganas, J. González-Domínguez, E. Solomonik, Y. Zheng, J. Touriño, and K. Yelick, “Communication avoiding and overlapping for numerical linear algebra,” UC Berkeley, Tech. Rep., 2012, to appear. [Online]. Available: http://bebop.cs.berkeley.edu