A Sparse Direct Solver for Distributed Memory Xeon Phi-accelerated Systems

Piyush Sao*, Xing Liu†, Richard Vuduc*, Xiaoye Li‡

*Georgia Institute of Technology
Email: {piyush3, richie} @gatech.edu
†IBM T. J. Watson Research Center
Email: xliu@us.ibm.com
‡Lawrence Berkeley National Laboratory
Email: xsli@lbl.gov

***Abstract*—This paper presents the first sparse direct solver for distributed memory systems comprising hybrid multicore CPU and Intel Xeon Phi co-processors. It builds on the algorithmic approach of SUPERLU_DIST, which is right-looking and statically pivoted. Our contribution is a novel algorithm, called the HALO. The name is shorthand for *highly asynchronous lazy offload*; it refers to the way the algorithm combines highly aggressive use of asynchrony with accelerated offload, lazy updates, and data shadowing (*a la* halo or ghost zones), all of which serve to hide and reduce communication, whether to local memory, across the network, or over PCIe. We further augment HALO with a model-driven autotuning heuristic that chooses the intra-node division of labor among CPU and Xeon Phi co-processor components.**

When integrated into SUPERLU_DIST and evaluated on a variety of realistic test problems in both single-node and multi-node configurations, the resulting implementation achieves speedups of up to $2.5\times$ over an already efficient multicore CPU implementation, and achieves up to 83% of a machine-specific upper-bound that we have estimated. Our analysis quantifies how well our implementation performs and allows us to speculate on the potential speedups that might come from a variety of future improvements to the algorithm and system.

***Index Terms*—Sparse Direct Solver; Xeon-Phi acceleration; GPU; MPI; OpenMP; Communication-avoiding algorithm; Heterogeneous computing;**

I. INTRODUCTION

The goal of this study is to significantly improve the state-of-the-art in sparse direct solvers for distributed memory machines, focusing on how to best exploit on-node accelerators. Sparse direct solvers compute the solution x of a linear system, $Ax = b$, where A is a sparse matrix, by Gaussian elimination, also known as sparse LU factorization. In a preliminary study, we enhanced SUPERLU_DIST—an existing state-of-the-art solver for multicore CPU-based clusters [1]—to use NVIDIA graphics accerlators [2]; however, our approach was limited in scope, concentrating primarily on how to effectively offload certain dense Basic Linear Algebra Subprograms (BLAS) subproblems, such as dense matrix-matrix multiply (GEMM). Such an approach is a natural first step. Indeed, it mirrors much of the existing work, which—until our GPU cluster work—considered only the *single-node* case [3]–[7]. However, it also falls short of what is ultimately possible. The present study overcomes this shortcoming.

For instance, we estimated the best-case speedup of our prior approach on one of the test problems and platforms considered in this study. If GEMM cost *zero* time units, that speedup would be *at most* $1.4\times$. This fares poorly against the method we propose herein, which by contrast achieves a speedup of $1.7\times$ on the same test problem.

By the way of explanation, the idea of offloading BLAS calls is not unreasonable. The most computationally expensive step in sparse LU factorization is the so-called "Schur-complement update" computation, which consists of two steps: multiplying two dense matrices (the GEMM step) and scattering the output of GEMM back into sparse format (the SCATTER step). Even if one only offloads large GEMMs, as we did previously so that Peripheral Component Interface Express (PCIe) transfer costs would not dominate, a non-offloaded SCATTER either becomes a bottleneck or—as multicore CPU memory bandwidth improves—becomes fast enough that overlapping with PCIe transfer for GEMM is no longer effective.

What we propose instead is a new approach, based on the high-level idea of using asynchronous execution as aggressively as possible. Our proposed algorithm shares structural similarities with communication optimal 2.5D LU factorization, which uses redundancy across processes to reduce communication across the network [8]. More specifically, we use redundancy between the CPU and the co-processor to reduce PCIe communication. However, translating the same high-level idea to sparse LU factorization is much harder, due to the irregular parallelism, irregular dependencies, and irregular data structures. To do this, we need to break some of the usual algorithmic abstraction boundaries, fusing distinct steps, such as GEMM and SCATTER, and using asynchrony to do so across iterations. In addition, we combine asynchrony with accelerated offload, lazy updates, and data shadowing (*a la* halo or ghost zones). This combination hides and reduces communication, whether to local memory, across the

During period of this research, Xing Liu was affiliated with Georgia Institute of Technology.

network, or over PCIe. We refer to this combined technique as the HALO algorithm, where the term HALO evokes *highly asynchronous lazy offload*.

We further enhance the basic HALO framework in two ways, to make it more effective in practice. First, we develop an empirical model-driven autotuning scheme to load balance within the node. This balancing occurs among *both* CPU cores *and* co-processor accelerators. The scheme overcomes limitations of both static load balancing, which can fail to accommodate the intrinsic dynamic and irregular nature of a sparse direct solver; and dynamic load balancing, which may incur high latency overheads due to PCIe. Secondly, we address the memory requirement problem of sparse direct solvers, by implementing a scheme that graceful degrades when offloading to an accelerator whose memory is much smaller than the host's memory. This is done by a heuristic that exploits the structure of a sparse direct solver's *elimination tree*. These enhancements to HALO make it practical.

Although HALO specifically concerns intra-node performance, it is easy to add our single-node implementation into a distributed memory code—namely, SUPERLU_DIST—and thereby accelerate the distributed case. The asynchronous nature of our approach naturally accommodates overlapping network communication with various on-node tasks. Additionally, although our experimental platform uses Intel Manycore Xeon Phi (MIC) co-processors, the technique is generic and could in principle apply to GPU-based platforms. Our hybrid MIC-accelerated SUPERLU_DIST achieves speedups of up to 2.5× on practical problems of interest (§ VI), relative to a highly scalable hybrid MPI+OpenMP baseline. To better understand performance, we analyze our code's performance issues and quantify the *potential* improvements. Together with our scaling experiments, this analysis helps us estimate the potential for future improvements in hardware, software, and runtime systems. Such findings may be of interest beyond the specific case of a sparse direct solver.

II. OVERVIEW OF SUPERLU_DIST

Solving a system of linear equations in SUPERLU_DIST involves three major steps: preprocessing of the input matrix, sparse LU factorization, and triangular solve. This paper focuses on the sparse LU factorization step, which can account for 75–99% of the total solve time. For brevity, we only sketch the sparse LU factorization step; the interested reader may find details about SUPERLU_DIST elsewhere [1].

SUPERLU_DIST performs a sparse LU factorization using the so-called *supernodal approach*. A *supernode* is a set of strongly connected vertices in the graph representation of the sparse matrix. During pre-processing step, SUPERLU_DIST extracts the supernodal structure from the input matrix, allowing it to store the sparse matrix as a collection of dense sub-matrices. This dense sub-matrix representation becomes the basis for exploiting fast level-3 BLAS operations, such as GEMM. However, unlike the case of factoring purely dense matrices, these dense subproblems have widely varying sizes.

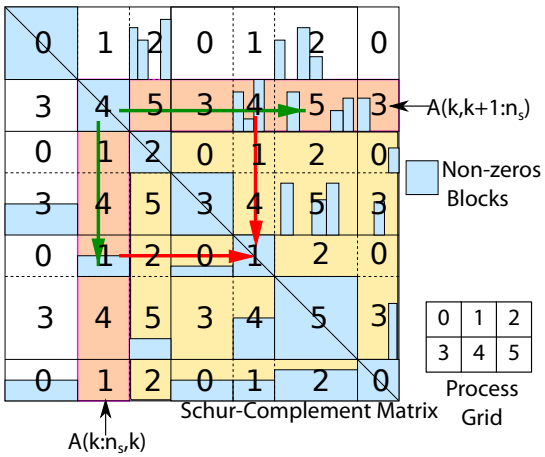


Figure 1: Data distribution in SUPERLU_DIST [1]: a sparse matrix distributed is among 6 MPI processes arranged in a 2×3 grid. Regions colored in blue denote the non-zero entries of the matrix. For k=2, $A(k{:}n_s, k)$ and $A(k, k{+}1{:}n_s)$ (orange background) form the k-th panel matrices, which after the k-th panel-factorization are overwritten by $L(k)$ and $U(k)$ panels, respectively. We also show the k-th Schur-Complement matrix $A(k{+}1{:}n_s, k{+}1{:}n_s)$ (yellow background).

Algorithm 1 SUPERLU_DIST Sparse LU Factorization

Input: Distributed sparse matrix A; n_s: number of supernodes; p_{id} : my process rank; $P_r(k)$: k-th process row; $P_c(k)$: k-th process column.
On each MPI process p_{id} do in parallel:
for $k = 1,\ 2,\ 3 \ldots\ n_s$ **do**
  Synchronize all processes
  Panel Factorization
  if p_{id} owns $A(k,k)$ **then**
    Factor $A(k,k)$ and send $L(k,k)$ to $P_r(k)$ who need it
    Send $U(k,k)$ to $P_c(k)$
  if $p_{id} \in P_c(k)$ **then**
    Wait for $U(k,k)$
    Factor the block column $L(k)$
    Send $L(k)$ blocks to needed processes in $P_r(:)$
  else
    Receive $L(k)$ blocks if needed
  if $p_{id} \in P_r(k)$ **then**
    Wait for $L(k,k)$
    Compute the block row $U(k)$
    Send $U(k)$ blocks to required processes in $P_c(:)$
  else
    Receive $U(k)$ blocks if required
  Schur-complement update
  if $L(:,k)$ and $U(k,:)$ are locally non-empty **then**
    for $j = k{+}1,\ k{+}2,\ k{+}3 \ldots\ n_s$ **do**
      for $i = k{+}1,\ k{+}2,\ k{+}3 \ldots\ n_s$ **do**
        if $p_{id} \in P_r(i) \cap P_c(j)$ **then**
          $A(i,j) \leftarrow A(i,j) - L(i,k)U(k,j)$

SUPERLU_DIST uses the Message Passing Interface (MPI) to express its distributed memory parallelism. The MPI processes are logically arranged in a two-dimensional (2D) process grid. On this 2D process grid, SUPERLU_DIST distributes the input matrix A in a block cyclic fashion. For example, fig. 1 shows sparse matrix distributed among 6 MPI processes arranged in a 2×3 grid.

SUPERLU_DIST's numerical factorization algorithm appears in alg. 1. The outer loop iterates over all the supernodes and factors them in sequence. Each iteration consists of two phases known as the panel-factorization phase and the Schur-complement update phase.

In the k-th iteration's panel-factorization phase, the algorithm factors the k-th panel matrices, $A(k:n_s, k)$ and $A(k, k{+}1:n_s)$, producing the factored panels $L(k)$ and $U(k)$. Additionally, the MPI process, p_{id}, that owns the $A(k,k)$ block factors it as $A(k,k) = L(k,k)U(k,k)$. It then sends $U(k,k)$ to the processes in the same process column group, denoted $P_c(k)$, to which p_{id} belongs. Similarly, it sends the block $L(k,k)$ to its process row group, $P_r(k)$. Each column process $p \in P_c(k)$ will, if it owns a non-empty block $A(i,k)$, calculates the local L factor[1] as $L(i,k) \leftarrow U(k,k)^{-1}A(i,k)$, and then sends $L(i,k)$ to its own process row. Similarly, each row process $p \in P_r(k)$ calculates a local U factor as $U(k,j) \leftarrow A(k,j)L(k,k)^{-1}$ for each non-zero block $A(k,j)$ that it owns, and then sends this block to its process column.

Once a process receives all its $L(i,k)$ and $U(k,j)$ blocks, it starts the Schur-complement update step. In this step, each process updates all the blocks $A(i,j)$ it owns, which form parts of the Schur-complement matrix, $A(k{+}1{:}n_s, k{+}1{:}n_s)$. More specifically, the update is

$$A(i,j) \leftarrow A(i,j) - L(i,k)U(k,j).$$

The sparsity patterns in $L(i,k)$ and $U(k,j)$ blocks may differ from the sparsity pattern in $A(i,j)$. Therefore, SUPERLU_DIST first calculates the product,

$$V \leftarrow -L(i,k)U(k,j),$$

by calling a presumably highly-optimized implementation of GEMM. Then, it maps elements of V to elements of $A(i,j)$ and then updates the mapped elements. We call this update SCATTER and denote it using the binary operator, $\oplus$, so that

$$A(i,j) \leftarrow A(i,j) \oplus V.$$

Since SCATTER involves indirect addressing, it can be expensive. The GEMM and SCATTER kernels together constitute the Schur-complement update. This is typically the most expensive sub-step of the sparse LU factorization phase.

III. THE DESIGN SPACE OF MIC-BASED SUPERLU_DIST

In contrast to a GPU, which today may only used as a co-processor for offloading, MIC has two possible execution modes. The first is similar to GPU offloading, and so is referred to as *offload mode*. For SUPERLU_DIST, one could offload compute intensive steps like GEMM to MIC. The second option is to use the MIC as an independent multicore node, launching multiple MPI processes onto MIC to run SUPERLU_DIST; this mode is called *native* mode.

In SUPERLU_DIST, where many calculations are sequential or lack enough parallelism, native mode is not likely to perform well. A single MIC core is slower than a typical high-end CPU core, as it executes in-order, at a lower operating frequency, and with a higher cache miss penalty. On the other hand, if we spawn heterogeneous MPI processes on both the CPUs and the MICs, load balancing among the CPUs and the MICs becomes difficult. Moreover, the MIC has a relatively low memory capacity, which limits the use of the MIC in native mode to matrices of relatively small sizes (table III), compared to what is possible on the CPU-based host. Thus, our approach focuses on using offload mode.

So what should be offloaded? Recall that there are two main phases, the panel-factorization and the Schur-complement update. Panel-factorization typically has insufficient parallelism for the MIC, and for a small number of MPI processes, it is also not usually the performance bottleneck. At relatively larger numbers of MPI processes, MPI communication costs dominate panel-factorization, which MIC acceleration cannot improve. Therefore, we do not consider this phase for offload.

By contrast, the Schur-complement update phase has a large number of independent GEMM and SCATTER calls that can account for more than 70-80% of the factorization time. Thus, this phase is a good offload candidate.

Our prior approach offloaded the Schur-complement update's GEMM calls to the GPU [2]. In each iteration, it offloaded a large GEMM call that multiplies L and U panel matrices, as $V^{m_t \times n_t} = -L^{m_t \times k_t} U^{k_t \times n_t}$, where typically $m_t, n_t \gg k_t$. Doing so required first sending the L and the U panel matrices to the GPU, then calling CUBLAS to compute GEMM, and then sending the product matrix V back to the CPU via PCIe. The SCATTER of V would occur on the CPU. To hide the data transfer costs, our prior approach pipelined the transfer of V and execution of the SCATTER.

This approach has two critical limitations. First, the bandwidth-bound SCATTER calls, since they remain on the CPU, cannot benefit from high GPU memory bandwidth. In the best test case of this paper, leaving SCATTER unaccelerated on a 20-core Intel *Ivybridge* system yields a maximum possible speedup of $1.4\times$, even if we assume the GEMM cost to be zero. Secondly, modern CPU bandwidth is significantly higher than the PCIe bandwidth. Therefore, efficiently pipelining PCIe transfer and SCATTER of V is not possible, since the SCATTER time is dwarfed by PCIe transfer time. Thus, our proposed algorithm will try to extend this prior approach by also finding a way to effectively offload SCATTER calls to MIC.

IV. THE HIGHLY ASYNCHRONOUS LAZY OFFLOAD ALGORITHM

To understand our new HALO algorithm, it helps to start with a natural and simpler method. For additional simplicity, first consider the single node case, which we will subsequently extend for the distributed memory case.

A primitive offload algorithm: Recall the k-th iteration of alg. 1. It factors the k-th *panel matrices*, $A(k{:}n_s, k)$ and $A(k, k{+}1{:}n_s)$, during the panel-factorization phase, producing the factored panels, $L(k)$ and $U(k)$. In the

[1]...overwriting A with the L and U factors

Algorithm 2 SUPERLU_DIST with MIC-offloading

1: Initialize $A_\phi \leftarrow 0$
2: **for** $k = 1,\ 2,\ 3 \ldots\ n_s$ **do**
Panel Factorization
same as alg. 1
.
.
.
Fetch and Assemble matrix on the CPU
3: (†) the MIC sends $A_\phi(k{+}1{:}n_s, k{+}1)$ and $A_\phi(k{+}1, k{+}1{:}n_s)$ blocks to the CPU
Hybrid Schur-complement Update
4: **if** $L(k)$ and $U(k)$ are locally non-empty **then**
5: find n_ϕ such that $k{<}n_\phi \leq n_s$
6: (‡) send $L(k)$ and $U(k, n_\phi : n_s)$ to the MIC
Schur Complement Update on the CPU
7: **for** $j = k{+}1,\ k{+}2,\ k{+}3 \ldots\ n_\phi - 1$ **do**
8: **for** $i = k{+}1,\ k{+}2,\ k{+}3 \ldots\ n_s$ **do**
9: **if** $p_{id} \in P_r(i) \cap P_c(j)$ **then**
10: $A(i,j) \leftarrow A(i,j) - L(i,k)U(k,j)$
Schur-complement Update on the MIC Asynchronously
11: Wait for (‡) to finish
12: **for** $j = n_\phi,\ n_\phi + 1,\ n_\phi + 2 \ldots\ n_s$ **do**
13: **for** $i = k{+}2,\ k{+}3,\ k{+}4 \ldots\ n_s$ **do**
14: **if** $p_{id} \in P_r(i) \cap P_c(j)$ **then**
15: $A_\phi(i,j) \leftarrow A_\phi(i,j) - L(i,k)U(k,j)$
Reduce the MIC updates with the CPU
16: the CPU waits for (†) to finish
17: $A(k{+}1{:}n_s, k{+}1) \leftarrow A(k{+}1{:}n_s, k{+}1){+}A_\phi(k{+}1{:}n_s, k{+}1)$
18: $A(k{+}1, k{+}2{:}n_s) \leftarrow A(k{+}1, k{+}2{:}n_s){+}A_\phi(k{+}1, k{+}2{:}n_s)$

Schur-complement update phase, it updates the k-th Schur-complement $A(k{+}1{:}n_s, k{+}1{:}n_s)$ by,

$$A(k{+}1{:}n_s, k{+}1{:}n_s) \leftarrow A(k{+}1{:}n_s, k{+}1{:}n_s){-}L(k)U(k).$$

Here is a primitive algorithm to offload the Schur-complement update phase to the MIC. This algorithm keeps a copy of the matrix A on the MIC. In the k-th iteration, it transfers the k-th panel matrices from the MIC to the CPU. Using the k-th panels, it calculates the factored panels $L(k)$ and $U(k)$ on the CPU. It then sends the $L(k)$ and $U(k)$ panels to the MIC and updates the k-th Schur-complement on the MIC. Thus, in each iteration this algorithm transfers a pair of panel matrices in each direction; these matrices are considerably smaller than the k-th Schur-complement. Consequently, the PCIe communication volume in each iteration can be relatively small. However, many iterations will not have enough parallelism to utilize the MIC well; therefore, those iterations may be significantly slower on the MIC than on the CPU. To avoid such slowdown, we instead consider *selectively offloading* the Schur-complement update to the MIC.

The HALO *algorithm:* The HALO algorithm enables selective offloading of the Schur-complement update to the MIC by extending the primitive algorithm as shown in alg. 2, summarized as follows.

HALO keeps the matrix A on the CPU and keeps a *structural* copy of the matrix A on the MIC, which it initializes with *zeros*. In other words, the matrix on the MIC has the same sparse data structure as the matrix A, but all the stored non-zero entries are initialized to zero. We denote the matrix on the MIC as A_ϕ to distinguish it from the matrix A on the CPU. In the k-th iteration, HALO transfers the k-th panel matrices from the MIC to the CPU. And, on the CPU, it *reduces* the k-th panels from the CPU and the MIC, via

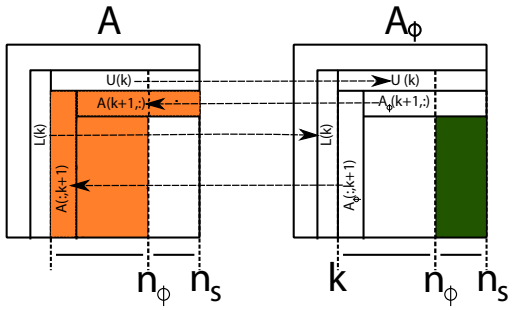


Figure 2: HALO: Schur-complement update in the k-th iteration. The $L(k)$ and $U(k)$ panels—calculated in k-th panel-factorization on the CPU—are sent to the MIC. The MIC sends $(k+1)st$ A-panels to the CPU. The CPU and MIC update parts of the k-th Schur-complement, shown in orange for the CPU and in green for MIC. The CPU merges the received MIC's $(k+1)st$ A-panels with its own $(k+1)st$ A-panels, before $(k+1)st$ iteration starts.

$$A(k{:}n_s, k) \leftarrow A(k{:}n_s, k){+}A_\phi(k{:}n_s, k); \quad (1)$$
$$A(k, k{+}1{:}n_s) \leftarrow A(k, k{+}1{:}n_s){+}A_\phi(k, k{+}1{:}n_s). \quad (2)$$

HALO then factors the reduced k-th panel matrices, yielding $L(k)$ and $U(k)$ on the CPU. If it offloads the Schur-complement update to the MIC, then it also sends the $L(k)$ and $U(k)$ panels to the MIC and updates the k-th Schur-complement there; otherwise it does the update on the CPU.

In general, HALO divides the k-th iteration's Schur-complement update between the CPU and the MIC. For some value n_ϕ, it updates the submatrix $A(k{+}1{:}n_s, k{+}1{:}n_\phi)$ on the CPU and $A(k{+}1{:}n_s, n_\phi + 1{:}n_s)$ on the MIC. We discuss how to choose n_ϕ in § V-B.

Figure 2 illustrates the regions of the matrix updated on and transferred from both the CPU and the MIC.

Note that in the k-th iteration, HALO does not update the $k{+}1$-th panels on the MIC. This way the MIC can start the transfer of the $k{+}1$-th panel before it executes the k-th Schur-complement update. Thus, HALO transfers the $k{+}1$-th panels and updates the k-th Schur-complement on the MIC in parallel. Figure 3 shows a sample execution timeline.

To see how this method works, consider any block $A(i,j)$ and its corresponding $A_\phi(i,j)$ on the MIC. Let $A^0(i,j)$ denote the initial value of $A(i,j)$ and recall that $A_\phi(i,j)$ is initially zero. In some iteration $k < \min(i,j)$, HALO either updates $A_\phi(i,j) \leftarrow A_\phi(i,j) - L(i,k)U(k,j)$ on the MIC or it updates $A(i,j) \leftarrow A(i,j) - L(i,k)U(k,j)$ on the CPU. Let $\mathcal{K}_1$ denote the set of iterations in which $A_\phi(i,j)$ is updated on the MIC, and let $\mathcal{K}_2$ denote those iterations in which $A(i,j)$ is updated on the CPU. Then, the snapshots of $A(i,j)$ and $A_\phi(i,j)$ are

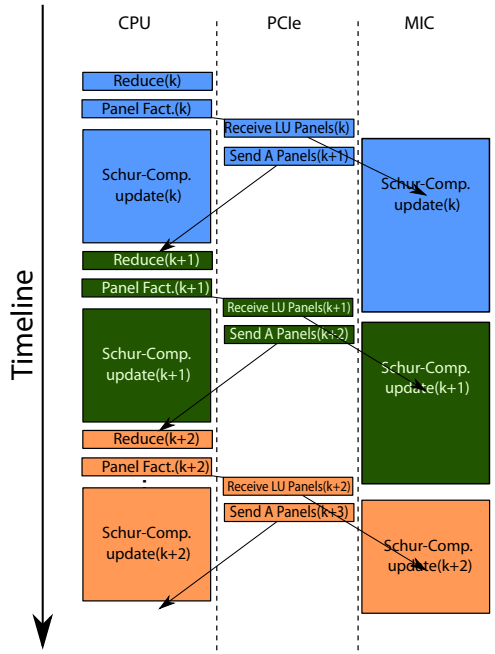


Figure 3: Concurrent execution of the Schur-complement update on the CPU and the MIC. *Send A panels(k)* denotes transfers of k-th panels of A_ϕ from MIC to CPU. *Reduce(k)* denotes reductions of k-th panels from the CPU and the MIC. *Receive LU panels(k)* denotes transfers of panels $L(k)$ and $U(k)$ from CPU to MIC. In general, the Schur-complement update is much longer than both other steps and data transfer.

given by:

$$A_\phi(i,j) \leftarrow -\sum_{k\in\mathcal{K}_1} L(i,k)U(k,j); \quad (3)$$

$$A(i,j) \leftarrow A^0(i,j) - \sum_{k\in\mathcal{K}_2} L(i,k)U(k,j). \quad (4)$$

Were we to add $A_\phi(i,j)$ to $A(i,j)$, that would be the same as updating $A(i,j)$ on $\mathcal{K}_1$ iterations, i.e.,

$$\begin{aligned} A(i,j) &\leftarrow A(i,j) + A_\phi(i,j) \\ &= A^0(i,j) - \sum_{k\in\mathcal{K}_1\cup\mathcal{K}_2} L(i,k)U(k,j). \end{aligned}$$

Thus, before the $k = \min(i,j)$-th iteration begins, we can fetch the block $A_\phi(i,j)$ and add it to $A(i,j)$. This reduced $A(i,j)$ block contains updates from all of the $\mathcal{K}_1 \cup \mathcal{K}_2$ iterations. Hence, when participating in the $k = \min(i,j)$-th panel-factorization, the $A(i,j)$ block in the MIC offload case is the same as in non-offloaded case. This argument holds for all the blocks participating in the k-th panel-factorization. Consequently, the factored panels $L(k)$ and $U(k)$ are the same in the case of MIC offload as they would have been otherwise.

Distributed HALO*:* In SUPERLU_DIST, each process owns a subset of the blocks of A following a 2D-cyclic data distribution. In the distributed HALO, we assign one MIC to each MPI process. Thus, we can conveniently assume that the shadow matrix, A_ϕ, has the same distribution the MICs as A.

Like the single node case, in each iteration's distributed panel-factorization, each process calculates or receives from other processes $L(k)$ and $U(k)$ blocks. It then transfers the $L(k)$ and $U(k)$ blocks to the MIC, and the CPU and the MIC can now update respective Schur-complement in parallel.

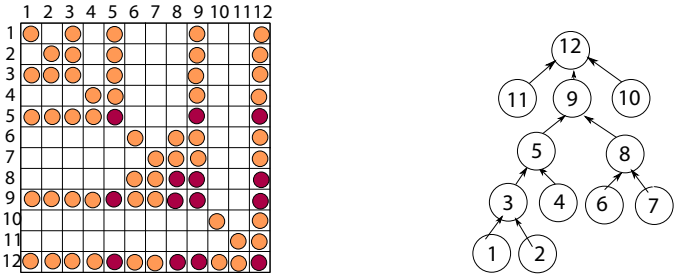
In contrast to single node case, for a given k only a subset of processes will own the blocks of $k+1$-th panels, following from the 2D cyclic distribution of the matrix. Therefore, in the k-th iteration, a process only needs to fetch the $k+1$-th panels from the MIC if it owns the $k+1$-th panel blocks, thereby further reducing the overall PCIe transfer volume.

V. ALGORITHMIC OPTIMIZATIONS FOR HALO

To make HALO practical for wide range of matrices, we augment HALO with several more performance optimizations. These optimizations expose parallelism at all levels of SIMD, multithreading, accelerator, and MPI, and reduces PCIe transfer volume and intra-node synchronizations. Due to a lack of space, here we describe the two key *algorithmic* enhancements for HALO.

A. Limited device memory considerations

Storing large matrices may require larger than the MIC's 8 GiB of memory. Using multiple MICs is not always possible or economical. Therefore, we augment the HALO algorithm with a kind of "out-of-core" strategy that keeps a submatrix on the MIC and offloads updates of this submatrix.



(a) (b)

Figure 4: (a) The non-zero structure of some sparse matrix; (b) the elimination tree of the sparse matrix [9]. If only 4 panels fit on the MIC, then our heuristic keeps the panels corresponding to nodes with largest number of descendants—here, 5, 8, 9 and 12—on the MIC.

Why might such an approach work in practice? In a sparse LU factorization, a small number of blocks are updated in many more iterations than other blocks. Therefore, the Schur-complement update of a small number of blocks can account for a large fraction of all Schur-complement update computations. For example, consider the 12×12 sparse matrix in fig. 4a. The $(12,12)$ block is updated during the Schur-complement updates of iterations 1 through 11. On the other hand, the blocks in the 1,2,4,6,7,10, and 11-th panels are not updated in any iteration's Schur-complement update. Therefore, it should be possible to offload a large fraction of the Schur-complement update operations keeping only a small fraction of matrix blocks on the MIC.

To choose these blocks, we use a heuristic based on the *elimination tree* of the sparse matrix [9]. The elimination tree, computed in any sparse LU factorization, shows which blocks will be updated in the k-th iteration's Schur-complement update. More specifically, we need only update the panels corresponding to the ancestors of the k-th node on elimination tree. For example, fig. 4b shows the elimination tree for the matrix of fig. 4a. In first iteration's Schur-complement update, only panels 3, 5, 9, and 12 need to be updated.

Conversely, the k-th panels are updated in all the iterations corresponding to descendants of k-th node in the elimination tree. Therefore, panels having the largest number of descendants are updated in largest number of iterations. Thus, our heuristic is to choose such panels. For example, in the elimination tree of fig. 4b, the nodes with the largest number of descendants are 5, 8, 9 and 12. Therefore, we would keep the 5, 8, 9 and 12-th panel matrices, shown in red in fig. 4a, on the MIC.

The best-case scenario for the above scheme is if the MIC has infinite memory, so that we would not need to consider by how much to restrict offload. Relative to this ideal scenario, § VI shows the above scheme can obtain speedups very close to the infinite-memory ideal.

B. Model-driven autotuning of intra-node load balance

Intra-node load balance i.e. choosing the optimal value of n_ϕ in fig. 2 is vital to achieving good load balance. However, the relative performance of the GEMM and the SCATTER kernel depends strongly on their input operand sizes. This motivates a model-driven scheme for choosing n_ϕ.

By way of motivation, consider fig. 5. It shows, for GEMM operations at various problem sizes, the speedup of MIC relative to a dual-socket 10-core Ivy Bridge system. While the theoretical *peak* performance of MIC is twice that of the aggregate peak of the Ivybridge system, fig. 5 clarifies that for a wide range of input sizes, the CPU can be much faster than MIC. As a function of problem size, the relative performance varies widely and nonlinearly.

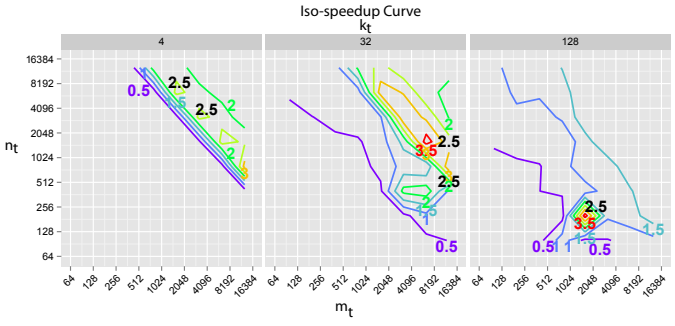


Figure 5: The speedup of MIC over a 20-core Ivy Bridge EP server varies widely and nonlinearly, for a GEMM that multiplies a $m_t \times k_t$ by a $k_t \times n_t$ matrix. (Speedups are shown as contour lines.)

Similarly, the performance of SCATTER on MIC also depends on the input block size, as fig. 6 shows. Due to MIC's in-order execution, it is crucial to use SIMD and software prefetching; however, for small blocks, it is hard to use SIMD and prefetching effectively. From one iteration to another, the distribution of block sizes in the Schur-complement update varies a lot, and small blocks are common.

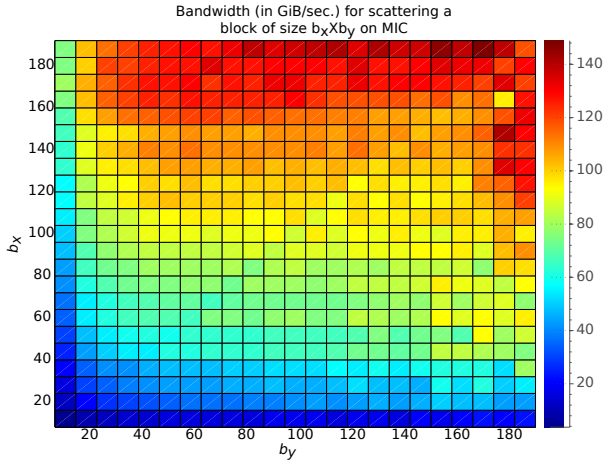


Figure 6: When scattering small blocks ($b_x \times b_y$), performance suffers due to poor SIMD and prefetch efficiency.

MDWIN*:* Among conventional generic approaches to load balancing, static load balancing is hard to do well under such workload variability, and dynamic load balancing over PCIe may incur high latency overheads. Instead, we propose a model-driven autotuning scheme, which we call MDWIN.

MDWIN is a static approach driven by an empirical performance model. At a high level, it tries to predict the execution time of the Schur-complement update using a simple analytical model, whose parameters derive from offline benchmarks. The model is calibrated on both the CPU and MIC, and is used to predict the value of n_ϕ at which the CPU and MIC are approximately balanced in time.

Let t_{GEMM}, t_{SCATTER}, $t^{(\phi)}_{\text{GEMM}}$, $t^{(\phi)}_{\text{SCATTER}}$ be the GEMM and SCATTER times on the CPU and MIC, respectively. In each iteration k, these are functions of n_ϕ. MDWIN seeks n_ϕ such that

$$t_{\text{GEMM}} + t_{\text{SCATTER}} \approx t^{(\phi)}_{\text{GEMM}} + t^{(\phi)}_{\text{SCATTER}}. \tag{5}$$

The component times of this model are determined as follows.

Modeling GEMM *costs:* To determine t_{GEMM} and $t^{(\phi)}_{\text{GEMM}}$, MDWIN maintains a lookup table of flop rates for $V \leftarrow L \cdot U$, where V is $m_t \times n_t$, L is $m_t \times k_t$, and U is $k_t \times n_t$. The sizes may be taken at a sample of points, the number of which may be used to tradeoff the table size and construction time. Let $F(m_t, n_t, k_t)$ denote this table of flop rates for the CPU. We simply estimate $t_{\text{GEMM}} = 2m_t n_t k_t / F(m_t, n_t, k_t)$. A similar table and formula for $t^{(\phi)}_{\text{GEMM}}$ may be constructed on MIC.

Modeling SCATTER *costs:* SCATTER is a memory bandwidth-bound kernel. Scattering a block of size $b_x \times b_y$ requires $3b_x \times b_y$ memory operations.[2] On the CPU, we observed that in most cases, only a few threads were sufficient to achieve

[2]For scatter operation $A(i,j) \leftarrow A(i,j) - V(i,j)$, we assume two reads and one write for each element

Matrix	n	$\frac{nnz(A)}{n}$	Fill-in ratio	# Flops in factorization
atmosmodd	1,270,432	6.93	244.00	1.12E+13
audikw_1	943,695	82.28	35.01	1.13E+13
dielFilterV3real	1,102,824	80.97	14.57	1.94E+12
Ga19As19H42	133,123	66.74	180.20	1.59E+13
Geo_1438	1,437,960	41.89	85.71	3.28E+13
H2O	67,024	33.07	210.98	2.28E+12
nd24k	72,000	398.82	23.08	3.98E+12
nlpkkt80	1,062,400	26.53	141.63	3.03E+13
RM07R	381,689	98.15	74.09	2.71E+13
torso3	259,156	17.09	63.80	3.11E+11

Table I: List of Matrices used for performance evaluation.

	Test-bed	IVB20C	BABBAGE
Host	CPU Micro-architecture	Ivy Bridge-EP	Sandy Bridge-EP
	Sockets/Cores/Threads	2/20/40	2/16/32
	Clock rate	2.80GHz	2.60GHz
	DRAM capacity	128 GB	128 GB
	Stream bandwidth	95 GB/s	72 GB/s
	Peak DP floating point performance	448 GF/s	332 GF/s
	PCIe type-Bandwidth	PCIe 2.0-8 GB/s	PCIe 2.0-8 GB/s
MIC	#MIC per node	1	2
	Clock rate	1.09 GHz	1.05 GHz
	Cores/Threads	61/244	60/240
	Stream bandwidth	163 GB/s	150 GB/s
	Peak DP floating point performance	1063 GF/s	2×1008 GF/s

Table II: Testbeds used for performance evaluation.

close to *stream bandwidth*, denoted B_{stream}. Therefore, on the CPU, we estimate t_{SCATTER} as sum of all the memory operations divided by the stream bandwidth.

On MIC, even if we use all of the cores, the achieved bandwidth for SCATTER operations depends more sensitively on the input operands sizes and their distribution. Therefore, MDWIN takes the *distribution* of sizes of blocks into account. Similar to the case of GEMM, we create a lookup table for the bandwidth achieved when scattering blocks of different sizes. This lookup table comes from running a microbenchmark. Such a table appears graphically in fig. 6. When we SCATTER the (i, j)-th block of size $b_x \times b_y$, we estimate the time spent as $3b_xb_y/B(b_x, b_y)$, where $B(b_x, b_y)$ is the value obtained from the lookup table. Thus, $t^{(\phi)}_{\text{SCATTER}}$ is estimated as,

$$t^{(\phi)}_{\text{SCATTER}} = \sum_{i,\ j} \frac{3b_x(i)b_y(j)}{B(b_x(i), b_y(j))}. \tag{6}$$

MDWIN is carefully implemented to reduce any overheads. Empirically, MDWIN's overhead is less than 2% of the total factorization time across our experiments (not shown here).

VI. EXPERIMENTS AND RESULTS

A. Experimental setup

Test Matrices: The matrices used in our tests are listed in table I. These matrices, taken from the University of Florida Sparse Matrix Collection, come from various real applications [10]. These matrices vary in sparsity structure,

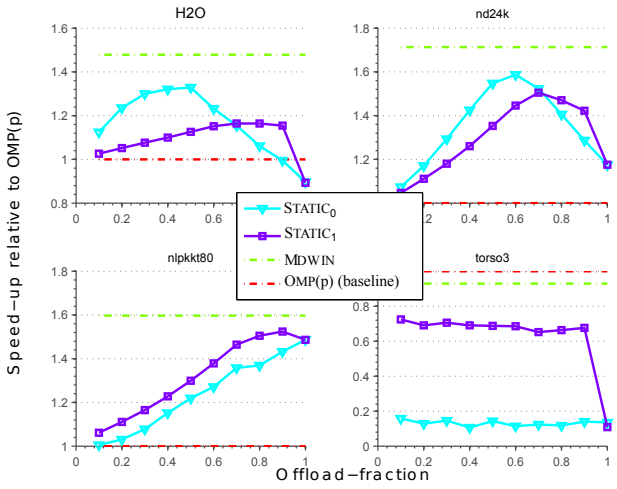


Figure 7: Comparison of model-driven work partitioning scheme to two static work partitioning scheme

which in turn affects the sparsity of the L and U factors, the factorization time, and the overall flop rates for the factorization.

Testbeds: We used two systems in the performance evaluation: IVB20C, which is a single-node 2×10-core Ivy Bridge-EP machine with a Intel Xeon Phi co-processors; and BABBAGE, which is a 45-node 2×8-core Sandy Bridge-EP with two Xeon-Phi cards machine, located at NERSC. The key machine parameters for the two systems are listed in table II.

We used the Intel C Compiler (ICC 15.0.0) with Intel MPI Runtime Library (IMPI 5.0) and Intel Math Kernel Library (MKL) version 11.1.

In all the experiments, we used the default settings for SUPERLU_DIST: ordering via *Metis* on $|A| + |A|^T$ and static pivoting and equilibration via *MC64*. The maximum size for any supernode was set to 192. Typically, a small supernode size eases load balance among different MPI processes; therefore, we chose a small supernode size where both the GEMM and SCATTER kernels obtain reasonable performance on both CPU and MIC.

For large matrices, we limited the user allocated memory on MIC to 7 GB. For the distributed experiments, for a given number of MPI processes, we tried different combinations of $P_r \times P_c$ on unaccelerated SUPERLU_DIST, and used the best configuration when running either with and without MIC acceleration.

We studied various single node characteristics of HALO on IVB20C. For comparison, we used multithreaded SUPERLU_DIST as our baseline. This baseline uses OpenMP threads across 20 Ivy Bridge cores, and is denoted by OMP(p).[3]

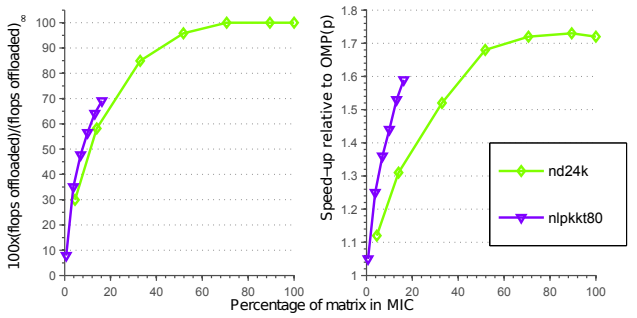


Figure 8: (Left) Flops offloaded to MIC as we vary the matrix kept in the device, shown as a percentage of (flops offloaded)$_\infty$. flops offloaded to MIC if it has *infinite* memory). It increases steeply with fraction of matrix kept in the MIC. (Right) The fraction of flops offloaded is directly correlated to the speed-up obtained relative to OMP(p).

B. Performance of model-driven work partitioning MDWIN

We evaluated the efficacy of MDWIN (see § V-B) by comparing its performance against two static work partitioning schemes, denoted STATIC$_0$ and STATIC$_1$. Both STATIC$_0$ and STATIC$_1$ assign a fixed fraction, called the *offload-fraction*, of columns of $U(k)$ to MIC, to divide work between the CPU and the MIC in each iteration. In addition, STATIC$_1$ does not offload any work to MIC in some iteration, if operand sizes are smaller than a fixed cut-off.[4]

We used four matrices for comparison. For each matrix, we vary the offload-fraction to find its optimal value (fig. 7). For both STATIC$_0$ and STATIC$_1$, for different matrices, the optimal offload-fraction occurred at different values. This illustrates the main limitation of such a fixed static partitioning scheme, which is that we cannot *tune* the offload-fraction for one matrix and use it for other matrices. In addition, a bad choice of the offload-fraction may slow down the computation by $10\times$, e.g., in case of STATIC$_0$ and *torso3* combination.

For all the matrices, MDWIN outperformed STATIC$_0$ and STATIC$_1$. Even in an especially difficult case (e.g., *torso3*), MDWIN incurred only a small slow down ($1.1\times$) versus 1.4–$10\times$ of STATIC$_0$ and STATIC$_1$.

C. Effect of limited MIC memory

We wish to see how well HALO performs on an accelerator whose memory is much smaller than the host's memory. To do so, we vary the fraction of the matrix in the MIC, to simulate acceleration with a small memory constraint, and see its effect on the number of flops offloaded. For this experiment, we use two matrices: (1) *nd24k* that can completely fit in the MIC's memory, and (2) *nlpkkt80* cannot.

In fig. 8 (left), the fraction of offloaded flops increases steeply as the matrix fraction stored on the MIC increases. For both the matrices, by only keeping 17% of the matrix in the MIC, we are able to offload more than 70% of the number of flops that we offload if the MIC has infinite memory. Thus, HALO can gracefully handle the relatively low memory capacity of the MIC.

In the fig. 8, we also show the speedup obtained. For large matrices, such as *nlpkkt80*, MIC-acceleration becomes more critical. Thanks to HALO, speedups for such a large matrix is close (within 10%) to the best small matrices case.

D. Single node performance on IVB20C

In our single node experiment, we seek to understand gains of MIC acceleration for different matrices. We compare the following two configurations of SUPERLU_DIST on IVB20C:

- OMP(p) (Baseline)
- OMP(p)+MIC: OMP(p) added with MIC acceleration.

For these two configurations, table III breaks down the factorization time and the obtained speedup. Overall, offloading the Schur-complement update to the MIC makes it faster by η^{sch}=0.9−1.8×, which results in an overall speed-up (η^{net}) of 0.9-1.7×. In addition to η^{sch}, overall speedup (η^{net}) also depends on the time spent in panel-factorization computations (t_{pf}), also shown in table III. In 8 out of 10 test cases, t_{pf}—being less than 20% of the baseline—is not the bottleneck.

Overall, the gains from HALO vary for different matrices. In our analysis, we treat total the factorization time (t_{omp}), the panel-factorization time (t_{pf}), and η^{sch} as independent quantities. However, they are related to the sparsity pattern of the input matrix, and can be inter-related. For example, the matrices *torso3* and *dielFilterV3real* are among the smallest in factorization time; they both also show a large t_{pf} and a small η^{sch}. Therefore, the study of the effects of sparsity-pattern on the performance of HALO is warranted.

E. Offload efficiency

We estimate the performance of HALO in the absence of any load imbalance. Load imbalance can be due to limited MIC memory, exposed PCIe communication and latency costs, or the limitations of MDWIN. Furthermore, it can cause the CPU and the MIC to idle, which would manifest as nonzero t_{cpu_idle} and t_{mic_idle} values in table III. In a hypothetical case where there are no load imbalances and PCIe communication has *zero* cost (due to either hardware or software improvements), if time for factorization is t_{ideal}, then offload efficiency ξ is given by $\frac{t_{ideal}}{t_{mic}}$. To calculate offload efficiency, we estimate t_{ideal} by making a simplifying assumption, which is that the incurred load imbalance can be shared *equally* between the CPU and the MIC. In other words, in the absence of any load imbalance, the factorization time of the HALO can be reduced further by $(t_{cpu\_idle}+t_{mic\_idle})/2$. Thus,

$$\xi = 1 - \frac{t_{mic\_idle} + t_{cpu\_idle}}{2t_{mic}}. \quad (7)$$

This value ranges from between 0.5 (only one resource, CPU or MIC, is working and the other is completely idle) and 1.0 (both CPU and MIC are working and perfectly load-balanced). We evaluate eq. 7 and show it in the last column

[3]It is the strongest of all possible baselines to which the MIC-accelerated version can be directly compared.

[4]We use $m_t = n_t = 512$, $k_t = 16$ as cut-offs, selected based on the relative performance of GEMM (fig. 5).

		Factorization time (in sec.)			Speed-up		Miscellaneous time (OMP(p)+MIC)			
	Matrix	OMP(p) (t_{omp})	OMP(p)+MIC (t_{mic})	Panel-fact. (t_{pf})‡	Schur-Comp. Update (η^{sch})	Overall (η^{net})	t_{cpu_idle}†	t_{mic_idle}†	t_{pcie}†	Offload efficiency(ξ)
Fits in MIC memory	H2O	41.9	28.3	4.3%	1.5	1.5	6.12%	32.4%	9.7%	80.7%
	nd24k	28.2	16.4	7.3%	1.8	1.7	4.9%	29.4%	7.6%	82.85%
	torso3	4.2	4.5	35.2%	0.9	0.9	7.9%	72.6%	4.8 %	59.7%
Does not fit in MIC memory	atmosmodd	64.2	43.4	14.1%	1.6	1.5	7.35%	50.8%	5.7%	70.3%
	audikw_1	50.3	33.7	16.1%	1.6	1.5	6.37%	49.5%	5.7%	72.4%
	dielFilterV3real	15.5	14.3	39.5%	1.1	1.1	2.7%	74.8%	6.4%	62.3%
	Ga19As19H42	224.3	165.8	2.9%	1.4	1.4	1.8%	59.6%	2.1%	69.3%
	Geo_1438	136.6	96.1	10.8%	1.5	1.4	1.34%	67.6%	2.7%	65.4%
	nlpkkt80	123.9	77.6	9.5%	1.7	1.6	0.44%	64.0%	2.9%	67.8%
	RM07R	136.3	87.6	5.7%	1.6	1.6	5.0%	54.9%	6.1%	70.0%

Table III: Factorization time of OMP(p) and OMP(p)+MIC for different matrices on the single node IVB20Csystem. ‡: Time shown is a percentage of t_{omp}. †: Time shown is a percentage of the t_{mic}. Overall speedup η^{net} ranges from 0.9 to 1.7×, and depends on the unaccelerated fraction (t_{pf}) and speedup obtained in MIC-accelerated Schur-complement update (η^{sch}). The last four columns show the idle time of the CPU and the MIC, the PCIe transfer time, and the estimated offload efficiency.

of table III. For many matrices, our implementation already achieves within 30% of the upper bound, and achieves close to 83% for *nd24k*. For bigger matrices, due to limited MIC memory, the offload efficiency hovers around 70%.

F. Single node performance on BABBAGE

Each node on BABBAGE has 2×8-core Sandy-Bridge sockets and two MICs. To use the both MICs, we spawn two MPI processes on each node and assign each MPI process a MIC. In addition to the shared memory configurations OMP(p)and OMP(p)+MIC, we also compare the following distributed memory configurations on BABBAGE:

- MPI(p)+OMP(q): One MPI process on each socket and uses OpenMP multi-threading at socket level; and
- MPI(p)+OMP(q)+MIC: MPI(p)+OMP(q)added MIC acceleration for each MPI process.

The factorization time on BABBAGE for the different configurations and matrices appears in fig. 9. The time for each configuration is split into the panel-factorization and the Schur-complement update phases. Note that the panel-factorization phase in MPI(p)+OMP(q) and MPI(p)+OMP(q)+MIC now include the time for MPI_Send, MPI_Recv and MPI_Wait. Therefore, t_{pf} increases when we go from shared memory to distributed memory configuration. On the other hand, the Schur-complement update phase shows better scalability in MPI(p)+OMP(q) configuration due to absence of any NUMA overheads.

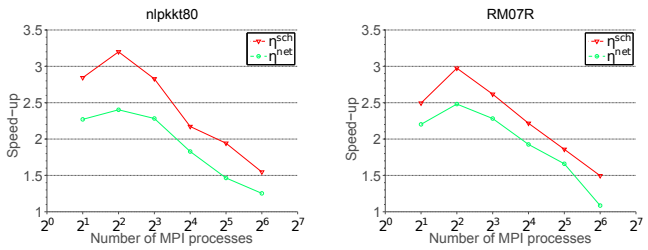
Overall, when we include another MIC, we can improve the performance by additional 1.1-1.8×. This performance improvement comes from two sources. First, we offload more computation to the MIC since relative performance of the MIC with respect to the host CPU has increased. Secondly, for large matrices such as RM07R and Ga19As9H42, where only a small fraction of the matrix can fit in one MIC, the increased fraction of the matrix fits in two MICs. Thus, more computation can be offloaded to the MIC.

G. Strong scaling on BABBAGE

We study strong scaling of HALO to understand the effects of MIC-acceleration when we scale SUPERLU_DIST to a large number of nodes. For this experiment, we use two matrices, *RM07R* and *nlpkkt80*, and we scale up to 32 nodes in two MPI processes per node configuration.

Figure 10 shows the scaling of the panel-factorization and Schur-complement update phases of computation. As a reference, we also give the scaling results for the baseline SUPERLU_DIST.

The Schur-complement update phase for both matrices and both configurations scales almost linearly with number of MPI processes, while the panel-factorization phase does not. Thus, at 64 MPI processes, the cost of panel-factorization dominates. HALO's scalability is affected even more than the baseline, since its Schur-complement update cost is smaller.



(a) (b)

Figure 11: Strong scaling on BABBAGE: speedup of Schur-complement update η^{sch} and overall speedup η^{sch} of MPI(p)+OMP(q)+MIC with respect to MPI(p)+OMP(q) for different number of MPI processes.

Figure 11 shows the speedup of HALO over the baseline. The speedup in the Schur-complement update phase, η^{sch}, increases when we go from two to four MPI processes. This increased speedup is due to an increased fraction of the matrix residing on the MICs. When we go beyond four processes,

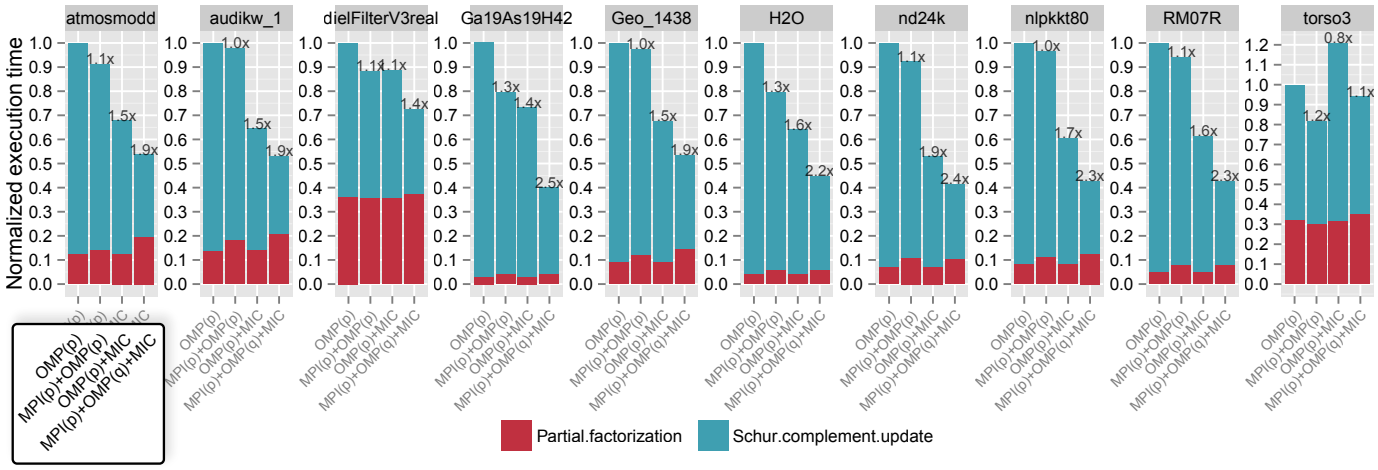
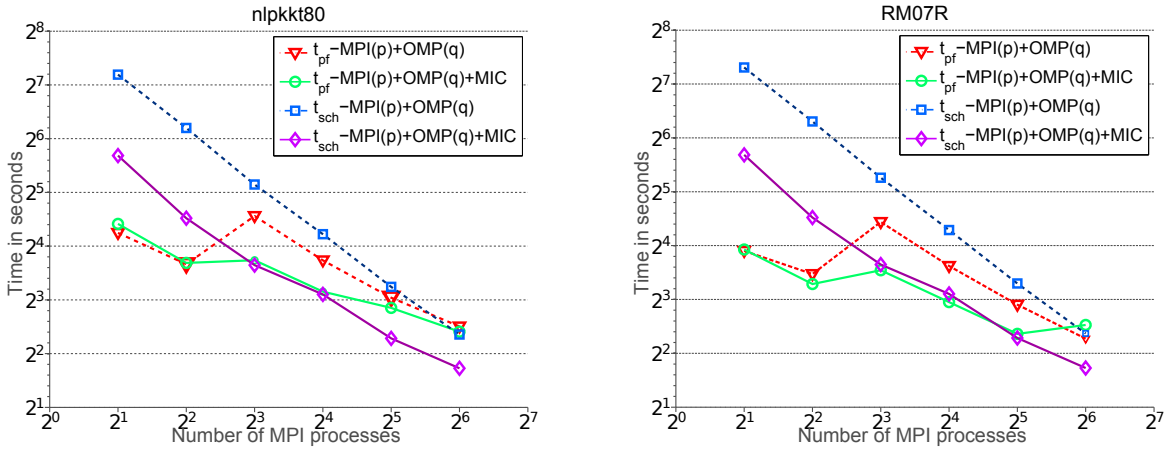


Figure 9: Performance for different matrices and different configuration of SUPERLU_DIST on the single node of the BABBAGE cluster. The configurations OMP(p) and MPI(p)+OMP(q) use only CPU cores, while OMP(p)+MIC and MPI(p)+OMP(q)+MIC, in addition to CPU cores, use one and two MICs, respectively. On top of each matrix×configuration bar, we show the speedup with respect to OMP(p). Overall, we obtain an additional 1.1-1.8× speedup when we use an additional MIC.



(a) (b)

Figure 10: Strong scaling of the baseline (MPI(p)+OMP(q)) and MIC-accelerated (MPI(p)+OMP(q)+MIC) configurations on BABBAGE. For both matrices, the panel factorization phase (t_{pf}) does not scale as well as the Schur-complement update. Therefore, at a large number of processes, panel factorization will become bottleneck.

the per iteration work of the Schur-complement update phase decreases; thus, η^{sch} also gracefully decreases, reducing to about 1.5× at 64 MPI processes. This η^{sch}, however, results in an overall speedup (η^{net}) of only 1 to 1.25×, as the panel-factorization phase dominates the total time at 64 MPI processes.

VII. RELATED WORK

A number of studies have reported GPU-accelerated multifrontal [11] sparse direct solvers in the case of a single node [3]–[7], [12], [13]. In the case of a distributed heterogeneous cluster, we have developed for the first time, a distributed *and* GPU-accelerated implementation that extends SUPERLU_DIST [2]. All of these approaches, including our own GPU implementation, accelerate by only offloading dense BLAS subproblems. Considering a larger domain of linear system solvers, there has been additional work involving GPUs on dense direct solvers [14], [15] and sparse iterative solvers [16].

HALO is distinct from this prior art. First, our choice of SUPERLU_DIST is algorithmically different from multifrontal methods, which other efforts address. Secondly, to our knowledge HALO is the first published attempt to use MIC-based acceleration for a sparse direct solver. Thirdly, we offload not only dense BLAS subproblems, but also SCATTER (see § IV). As SCATTER can quickly become the bottleneck [2], this enhancement is significant.

Our interest in MIC stems from its emergence as a viable GPU alternative, as is evident by its presence on, for instance, the Top500 and Green500 lists [17], [18]. There are numer-

ous reports of successful MIC-acceleration for a variety of scientific problems [19]–[22]. A sparse direct solver presents unique challenges of moderate and highly-variable arithmetic intensity, which makes it neither purely compute-bound nor purely memory bandwidth-bound.

VIII. Conclusions and Future Work

None of the technical components of HALO are specific to MIC, meaning the same ideas should extend naturally to GPU-based clusters or other heterogeneous node architectures. However, architectural difference between GPU and MIC may result in different relative performance profiles for different operand sizes. We are working on a combined software infrastructure that can exploit either or both types of accelerators.

At a sufficiently large number of MPI processes, one should expect the time spent in panel-factorization phase to begin to dominate. This would happen primarily because of increasing load imbalance among the processes, which leads to increases in MPI_Wait and MPI_Recv times. This problem could be avoided if the bottleneck process could assign more work to its accelerator, as a way of reducing the apparent load imbalance. However, knowing precisely when to do so would require a scheme to estimate time at a *global* level, which would have to include modeling of potential load imbalance and MPI communication costs. This topic is an excellent one for future work.

Part of the proposed scheme targets the relatively smaller memory capacities of accelerators compared to their hosts. However, an intrinsic problem is that the per-process memory requirement tends to increase as the number of processes increases. In this sense, accelerators can help by decreasing the need for more MPI processes. Nevertheless, a more formal and precise understanding of this issue would be needed to scale sparse direct solvers to the next level.

IX. Acknowledgment

We thank Edmond Chow for local access to a MIC platform.

This work was supported in part by the National Science Foundation (NSF) under NSF CAREER award number 0953100 and NSF SI2-SSI Award 1339745. Additional support provided by the U.S. Dept. of Energy (DOE), Office of Science, Advanced Scientific Computing Research (and Basic Energy Sciences/Biological and Environmental Research/High Energy Physics/Fusion Energy Sciences/Nuclear Physics), through the Scientific Discovery through Advanced Computing (SciDAC) program; and X-Stack 1.0 under DE-FC02-10ER26006/DE-SC0004915. We also used resources of the National Energy Research Scientific Computing Center, which is supported by the DOE Office of Science under Contract No. DE-AC02-05CH11231.

Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of NSF or DOE.

References

[1] X. S. Li and J. W. Demmel, "SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems," *ACM Trans. Mathematical Software*, vol. 29, no. 2, pp. 110–140, June 2003.

[2] P. Sao, R. Vuduc, and X. S. Li, "A distributed cpu-gpu sparse direct solver," in *Euro-Par 2014 Parallel Processing*. Springer International Publishing, 2014, pp. 487–498.

[3] R. Vuduc, A. Chandramowlishwaran, J. Choi, M. Guney, and A. Shringarpure, "On the limits of GPU acceleration," in *Proc. of the 2nd USENIX conference on Hot topics in parallelism, HotPar'10*, Berkeley, CA, 2010.

[4] G. Krawezik and G. Poole, "Accelerating the ANSYS direct sparse solver with GPUs," in *Proc. Symposium on Application Accelerators in High Performance Computing (SAAHPC)*, Urbana-Champaign, IL, NCSA, 2009.

[5] T. George, V. Saxena, A. Gupta, A. Singh, and A. Choudjury, "Multifrontal factorization of sparse spd matrices on GPUs," in *Proc. of IEEE International Parallel and Distributed Processing Symposium (IPDPS 2011)*, Anchorage, Alaska, May 16-20 2011.

[6] R. Lucas, G. Wagenbreth, D. Davis, and R. Grimes, "Multifrontal computations on GPUs and their multi-core hosts," in *VECPAR'10: Proc. 9th Intl. Meeting for High Performance Computing for Computational Science*, Berkeley, CA, 2010.

[7] C. Yu, W. Wang, and D. Pierce, "A CPU-GPU hybrid approach for the unsymmetric multifrontal method," *Parallel Computing*, vol. 37, pp. 759–770, 2011.

[8] E. Solomonik and J. Demmel, "Communication-optimal parallel 2.5 d matrix multiplication and lu factorization algorithms," in *Euro-Par 2011 Parallel Processing*. Springer, 2011, pp. 90–109.

[9] I. S. Duff, A. M. Erisman, and J. K. Reid, *Direct methods for sparse matrices*. Clarendon press Oxford, 1986.

[10] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, p. 1, 2011.

[11] I. S. Duff and J. K. Reid, "The multifrontal solution of indefinite sparse symmetric linear," *ACM Transactions on Mathematical Software (TOMS)*, vol. 9, no. 3, pp. 302–325, 1983.

[12] X. Lacoste, M. Faverge, G. Bosilca, P. Ramet, and S. Thibault, "Taking advantage of hybrid systems for sparse direct solvers via task-based runtimes," in *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*. IEEE, 2014, pp. 29–38.

[13] K. Kim and V. Eijkhout, "Scheduling a parallel sparse direct solver to multiple gpus," in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*. IEEE, 2013, pp. 1401–1408.

[14] N. Galoppo, N. K. Govindaraju, M. Henson, and D. Manocha, "Lu-gpu: Efficient algorithms for solving dense linear systems on graphics hardware," in *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, 2005, p. 3.

[15] H. Ltaief, S. Tomov, R. Nath, P. Du, and J. Dongarra, "A scalable high performant cholesky factorization for multicore with gpu accelerators," in *High Performance Computing for Computational Science–VECPAR 2010*. Springer, 2011, pp. 93–101.

[16] J. Bolz, I. Farmer, E. Grinspun, and P. Schröoder, "Sparse matrix solvers on the gpu: conjugate gradients and multigrid," in *ACM Transactions on Graphics (TOG)*, vol. 22, no. 3. ACM, 2003, pp. 917–924.

[17] "Green500 list," http://www.green500.org/, 2013.

[18] "Top500 list," http://www.top500.org/, 2013.

[19] A. Heinecke, K. Vaidyanathan, M. Smelyanskiy, A. Kobotov, R. Dubtsov, G. Henry, A. G. Shet, G. Chrysos, and P. Dubey, "Design and implementation of the linpack benchmark for single and multi-node systems based on intel® xeon phi coprocessor," in *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*. IEEE, 2013, pp. 126–137.

[20] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey, "Efficient sparse matrix-vector multiplication on x86-based many-core processors," in *Proceedings of the 27th international ACM conference on International conference on supercomputing*. ACM, 2013, pp. 273–282.

[21] J. Park, G. Bikshandi, K. Vaidyanathan, P. T. P. Tang, P. Dubey, and D. Kim, "Tera-scale 1d fft with low-communication algorithm and intel® xeon phi? coprocessors," in *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2013, p. 34.

[22] J. Park, M. Smelyanskiy, K. Vaidyanathan, A. Heinecke, D. D. Kalamkar, X. Liu, M. M. A. Patwary, Y. Lu, and P. Dubey, "Efficient shared-memory implementation of high-performance conjugate gradient benchmark and its application to unstructured matrices," 2014.