A distributed CPU-GPU sparse direct solver

Piyush Sao[1], Richard Vuduc[1], and Xiaoye Li[2]

[1] Georgia Institute of Technology, {piyush3,richie}@gatech.edu
[2] Lawrence Berkeley National Laboratory, xsli@lbl.gov

Abstract. This paper presents the first hybrid MPI+OpenMP+CUDA implementation of a distributed memory right-looking unsymmetric sparse direct solver (i.e., sparse LU factorization) that uses static pivoting. While BLAS calls can account for more than 40% of the overall factorization time, the difficulty is that small problem sizes dominate the workload, making efficient GPU utilization challenging. This fact motivates our approach, which is to find ways to aggregate collections of small BLAS operations into larger ones; to schedule operations to achieve load balance and hide long-latency operations, such as PCIe transfer; and to exploit simultaneously all of a node's available CPU cores and GPUs.

1 Introduction

Given a sparse matrix A, we consider the problem of factoring it into the product $A = L \cdot U$, where L is a unit lower triangular matrix and U is an upper triangular matrix. This problem ("sparse LU") is usually the most expensive step in a *sparse direct solver*, the use of which appears in a variety of computational science and engineering applications. It typically needs a lot of memory, thereby benefiting from the use of a distributed memory system. A natural question is, given the increased reliance on some form of GPU-like acceleration for such systems, how to exploit all forms of available parallelism, whether distributed memory, shared memory, or "accelerated."

The challenge is that sparse LU factorization is, computationally, neither strictly dominated by arithmetic, like high-performance LINPACK is when A is dense, nor is it strictly dominated by communication, as is often the case with iterative linear solvers. Thus, it is an open question whether or by how much we should expect to speed up sparse LU factorization using distributed CPU+GPU machines [9]. Additionally, the facts of indirect irregular memory access, irregular parallelism, and a strong dependence on the input matrix's structure—known only at runtime—further complicate its implementation. These complications require carefully designed data structures and dynamic approaches to scheduling and load balancing. Indeed, perhaps due to these myriad issues, there are many studies offering distributed algorithms and hybrid single-node CPU+GPU implementations but, to date, no fully distributed hybrid CPU+GPU sparse direct solver of which we are aware (§ 2).

This paper presents the first such algorithm and implementation that can run scalably on a cluster comprising hybrid CPU+GPU nodes. We extend an existing distributed memory sparse direct solver, SuperLU_DIST [5], by adding

CPU multithreading and GPU acceleration during the LU factorization step. To effectively exploit intranode CPU and GPU parallelism, we use a variety of techniques (§ 4). These include aggregating small computations to increase the amount of compute-bound work; asynchronously assigning compute-bound work to the GPU and memory-bound work to the CPU, thereby minimizing CPU-GPU communication and improving system utilization; and careful scheduling to hide various long-latency operations. We evaluate this implementation on test problems derived from applications (§ 5). We show speedups of over 2× (§ 5) over a highly scalable MPI-only code; and, provide the required explanation when our approach does not yield speedups.

2 Related work

The last five years have seen several research developments on accelerating sparse factorization algorithms using GPUs. Most of these efforts rely on the GPU for solving large dense matrix subproblems, performing any other processing on the host CPU with data transfer as needed. There exist methods for multifrontal Cholesky [4, 12, 9, 6]; and, in single-precision, left-looking sparse LU [8]. In essence, all of these methods use the GPU as a BLAS accelerator.

George et al. go beyond BLAS acceleration for their single-node multifrontal sparse Cholesky algorithm, implemented in WSMP [3]. They examine three compute-intensive kernels associated with each frontal matrix: factoring the diagonal block, triangular solution, and Schur complement update. These computations are selectively offloaded to the GPU depending on the workload distribution of the flops, which in turn depends on the input matrix. Their method achieves 10-25× speedups over a single-core.

Yeralan et al. developed a sparse multifrontal QR factorization algorithm using one CPU-GPU combination [11]. Since sparse QR has intrinsically higher arithmetic intensity than sparse LU, the pay-off of GPU acceleration should be higher.

Our approach also offloads the most arithmetic-intensive part of the workload to GPUs. However, one distinction of our work is that we aim to exploit the maximum available parallelism of a distributed memory system, namely, distributed memory parallelism via MPI combined with intranode parallelism through multithreading and GPU acceleration. While our implementation is specific to SuperLU_DIST, we believe techniques discussed in this paper can be extended to other direct solvers.

3 Overview of SuperLU_DIST

Solving a linear system $Ax = b$ using SuperLU_DIST involves a number of steps [10, 7]. However, the most expensive step is numerical factorization, which is the focus of this paper. For test matrices in our study, numerical factorization accounts for at least 75% of the total solve time, and in fact more often accounts

Algorithm 1 SuperLU_DIST Numerical Factorization

```
1: for $k = 1,\ 2,\ 3 \ldots\ n_s$ do
       Panel Factorization
2:     Column computation of $L_{:,k}$.
3:     if $p_{id} \in P_c(k)$ then
4:         compute the block column $L_{k:n_s,k}$
5:         (communicate $U_{k,k}$ among $P_c(k)$)
6:         send $L_{k:n_s,k}$ to required processes in $P_r(:)$
7:     else
8:         receive $L_{k:n_s,k}$ if required
       Row computation of $U_{k,:}$.
9:     if $p_{id} \in P_r(k)$ then
10:        wait for $U_{k,k}$
11:        compute the block row $U_{k,k+1:n_s}$
12:        send $U_{k,k+1:n_s}$ to required processes in $P_c(:)$
13:    else
14:        receive $U_{k,k+1:n_s}$ if required
       Schur Complement Update
15:    if $L_{:,k}$ and $U_{k,:}$ are locally non-empty then
16:        for $j = k+1,\ k+2,\ k+3 \ldots\ n_s$ do
17:            for $i = k+1,\ k+2,\ k+3 \ldots\ n_s$ do
18:                if $p_{id} \in P_r(i) \cap P_c(j)$ then
19:                    $A_{i,j} \leftarrow A_{i,j} - L_{i,k}U_{k,j}$
```

for 90% or more. Therefore, we focus on just the numerical factorization phase. Accelerating the remaining steps is a good avenue for future research.

SuperLU_DIST uses a fan-out (right-looking, outer-product) *supernodal* algorithm. A *supernode* is a set of consecutive columns of L with a dense triangular block just below the diagonal and with the same nonzero structure below the triangular block. To achieve good parallelism and load balance, the MPI processes are assigned to the supernodal blocks in a 2D cyclic layout.

Algorithm 1 shows the pseudocode of the factorization algorithm, where n_s is the number of supernodes, p_{id} is the ID of this process, and $P_c(k)$ and $P_r(k)$ are the groups of processes assigned to the k-th supernodal column and the k-th supernodal row, respectively. Step 1 is the k-th panel factorization, where the k-th supernodal column of L and the k-th supernodal row of U are computed. Subsequently, each process in $P_c(k)$ and $P_r(k)$ sends its local blocks of the factors to the processes assigned to the same row and column, respectively. Consequently, Step 2 updates the trailing submatrix using the k-th supernodal column and row of the LU factors. The block $A_{i,j}$ is updated only if both blocks $L_{i,k}$ and $U_{k,j}$ are not empty. A more detailed description appears elsewhere [10].

4 New Intranode Enhancements

Our work enhances the intranode performance and scaling of alg. 1. The panel factorization and row computation phases primarily are concerned with communication. By contrast, the Schur complement update phase (lines 15–19) is the local computation that dominates intranode performance. Thus, it is our main target for optimization.

Baseline Schur complement update. The Schur complement update step at iteration k of alg. 1 computes $A_{k+1,n_s:k+1,n_s}$ as

$$A_{k+1,n_s:k+1,n_s} = A_{k+1,n_s:k+1,n_s} - L_{k+1:n_s,k} U_{k,:k+1:n_s}. \qquad (1)$$

SuperLU_DIST uses an owner-computes strategy, where each process updates the set of blocks, $\{A_{i,j}\}$, which it owns once it has received the required blocks $L_{:,k}$ and $U_{k,:}$.

Each GEMM subproblem computes one $A_{i,j}$, which is line 19 of alg. 1. In the baseline SuperLU_DIST implementation, a process updates each of its $A_{i,j}$ blocks in turn, traversing the matrix in a columnwise manner (outermost j-loop at line 18 of alg. 1). The update takes place in three steps: packing the U block, calling BLAS GEMM, and unpacking the result. We refer to the first two steps as the *GEMM* phase, and the last step as the *Scatter* phase.

Packing allows the computation to use a highly optimized BLAS implementation of GEMM. Packing converts the $U_{k,j}$, which is stored in a sparse format, into a dense BLAS-compliant column major format, $\tilde{U}_{k,j}$. This packing takes place once for each $U_{k,j}$. The $L_{i,k}$ operand need not be packed, as it is already stored in a column major form as part of a rectangular supernode.

The second step is the BLAS GEMM call, which computes $V \leftarrow L_{i,k}\tilde{U}_{k,j}$, where V is a temporary buffer.

The final *Scatter* step updates $A_{i,j}$ by subtracting V from it. Since only the nonzero rows of L and U are stored, the destination block $A_{i,j}$ usually has more nonzero rows and columns, than $L_{i,k}$ and $U_{k,j}$. Thus, this step must also map the rows and columns of V to the rows and columns of $A_{i,j}$ before the elements of $A_{i,j}$ can be updated, which involves indirect addressing. This final unpacking step is what we refer to as the *Scatter* phase.

Aggregating small GEMM subproblems. Relative to the baseline (above), we may increase the intensity of the GEMM phase by aggregating small GEMM subproblems into a single, larger GEMM. This aggregated computation then becomes a better target for GPU offload, though it also works well even in the multicore CPU-only case.

Our approach to aggregation, illustrated in fig. 1, has two aspects. First, we process an entire block column at once. That is, instead of calling GEMM for every block multiply $L_{i,k}\tilde{U}_{k,j}$, we aggregate the L-blocks in column k into a single GEMM call that effectively computes $V \leftarrow L_{k+1:n_s,k}\tilde{U}_{k,j}$, thereby reusing $\tilde{U}_{k,j}$. Secondly, the packed block $\tilde{U}_{k,j}$ may still have only a few nonzero columns. Thus, we group multiple consecutive U-blocks to form a larger $\tilde{U}_{k,j_{st}:j_{end}}$ block, where j_{st} and j_{end} are the starting and the ending block indices. This large block has some minimum number of columns N_b, a tuning parameter. We schedule

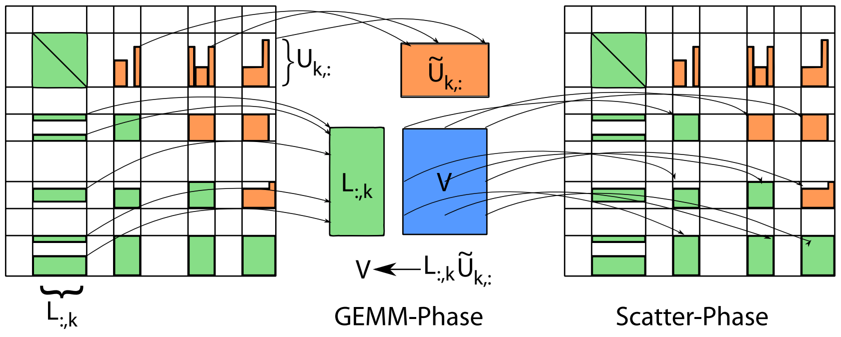


Fig. 1: Aggregating Small GEMM subproblems

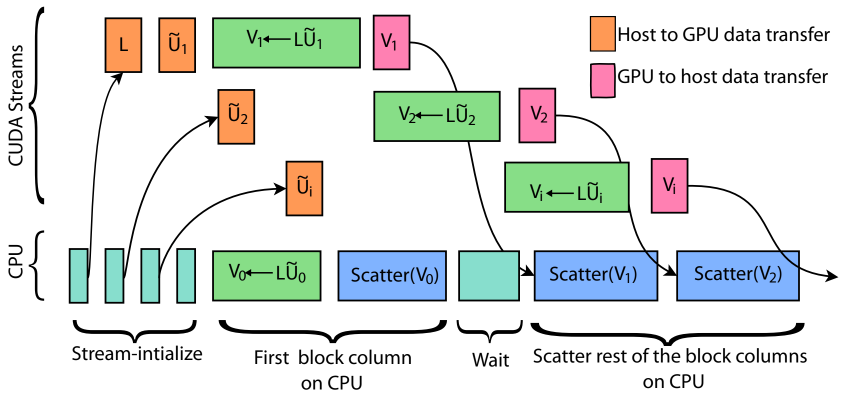

CUDA Streams
CPU
L
Ũ1
V1 ← LŨ1
V1
Ũ2
V2 ← LŨ2
V2
Ũi
Vi ← LŨi
Vi
V0 ← LŨ0
Scatter(V0)
Scatter(V1)
Scatter(V2)
Host to GPU data transfer
GPU to host data transfer
Stream-intialize
First block column on CPU
Wait
Scatter rest of the block columns on CPU

Fig. 2: Overlapping GEMM with Scatter

the computation of $L_{k+1:n_s,k}\tilde{U}_{k,j_{st}:j_{end}}$ onto the GPU, using CUDA streams as explained below.

Aggregation may increase the memory footprint relative to the baseline. In particular, we may need to store a large U-block, $\tilde{U}$, and a large intermediate output, V. Our implementation preallocates these buffers, using N_b as a tunable parameter to constrain their sizes.

Pipelined execution. Given aggregated GEMMs, we use a software pipelining scheduling scheme to overlap copying the GEMM operands to the GPU with execution of both the GEMMs themselves as well as the CPU Scatter.

Our pipelining scheme, illustrated in fig. 2, uses CUDA's streams facility. Our scheme divides $\tilde{U}$ into n_s partitions, where n_s is the number of desired CUDA streams, a tuning parameter. To perform this division, our scheme first ensures that each partition has a minimum of N_b columns. It also ensures that the number of columns in each partition does not cross the boundary of the block columns. It then uses a greedy algorithm to ensure that each partition has a number of columns of at most the average number of columns, except for the last partition which has all the remaining columns.

The pipelining begins with the transfer of L to the GPU. Now each CUDA stream asynchronously initializes transfer of i-th partition, $\tilde{U}_i$, and a CUDA BLAS GEMM call to perform $V_i \leftarrow L\tilde{U}_i$, and transfer of V_i to the host. Once V_i is copied back to the host, this V_i is scattered as soon as possible. We schedule the GEMM and scatter of the first block column on CPU. This is done to minimize idle time of CPU, while it waits for the first CUDA stream to finish transferring the V_1. Note that CUDA streams mainly facilitates overlap of CPU, GPU, and PCIe transfer. The streams themselves may, but do not necessarily, overlap

CUDA streams facility carries a nontrivial setup overhead. Suppose asynchronous CUDA calls take time t_s to initialize, and the effective floating-point throughput of the CPU is F_{cpu} operations per unit time. Then, offloading fewer than $t_s F_{CPU}$ would be slower than executing on the host. Our implementation uses such a heuristic to decide whether offloading a particular GEMM phase to the GPU will pay off, or otherwise executes on the CPU.

OpenMP parallelization of Scatter. We parallelized Scatter using OpenMP. There are a number of ways to assign blocks to be scattered to threads. Prior work on SuperLU_DIST used a block cyclic assignment [10]. However, we discovered by experiment that particular static assignment can lead to severe load imbalance. In addition, assigning one block to a thread can be inefficient since many blocks may have very little work in each, leading to an overly fine grain size.

We address these issues as follows. When there are a sufficient number of block columns, we schedule the Scatter of the entire block column to one thread using OpenMP's guided scheduling option. We also tried dynamic scheduling options, but for our test cases, there was no significant difference in performance. When there are fewer block columns than the number of threads, we switch from parallelizing across block columns to parallelizing across block rows.

In addition, we also use OpenMP to parallelize the local work at the lookahead phase and the panel factorization phase. However, doing so does not affect performance by much because these phases are dominated by MPI communication.

5 Experiments and Results

We used two GPU clusters in our evaluation (table 2). We tested our implementations on the input matrices in table 2, which derive from real applications [2].

We evaluated 6 implementation variants. (All variants use double-precision arithmetic, including on the GPU.) The baseline is SuperLU_DIST. We *modified* this baseline to include the BLAS aggregation technique of § 4. Since all variants derive from SuperLU_DIST, they all *include* distributed memory parallelism via MPI. Their mnemonic names describe what each variant adds to the MPI-enabled baseline.

Parameter	Jinx-Cluster	Dirac-GPU test bed
# GPUs per node	2	1
Type of GPU	Tesla M2090 "Fermi"	Tesla C2050 "Fermi"
GPU double-precision peak	665 GF/sec	515 GF/sec
GPU DRAM / Bandwidth	6 GB / 177 GBytes/sec	3 GB / 144 GBytes/sec
Host	Intel Xeon X5650 @2.67 GHz	Intel Xeon X5530 @2.4 GHz
PCIe / Bandwidth	PCIe x16 /8GB/s	PCIe x16 /8GB/s
Sockets $\times$ Cores / socket	2×6	2×4
CPU double-precision peak	128 GF/sec	76.8 GF/sec
L3 Cache	$2 \times$ 12M	$2\times$ 8M
Memory	24GB	24GB
Network /Bandwidth	InfiniBand/ 40 Gbit/s	InfiniBand/ 32 Gbit/s

Table 1: Evaluation testbeds for our experiments

Name	n	nnz	$\frac{nnz}{n}$	symm	Fill-in Ratio	Application
audikw_1*	943695	77651847	82.28	yes	31.43	structural problem
bone010*	986703	47851783	48.49	yes	43.52	model reduction problem
nd24k*	72000	28715634	398.82	yes	22.49	2D/3D problem
RM07R*	381689	37464962	98.15	no	78.00	computational fluid dynamics
dds.quad†	380698	15844364	41.61	no	20.18	cavity
matrix211†	801378	129413052	161.48	no	9.68	Nuclear Fusion
tdr190k†	1100242	43318292	39.37	no	20.43	Accelerator
Ga19As19H42*	133123	8884839	66.74	yes	182.16	quantum chemistry problem
TSOPF_RS_b2383_c1*	38120	16171169	424.21	no	3.44	power network problem
dielFilterV2real*	1157456	48538952	41.93	yes	22.39	electromagnetics problem

Table 2: Different test problems used for testing solvers. * See the University of Florida Sparse Matrix Collection [2]; † from NERSC users

- `MKL`$_1$ is the baseline, based on SuperLU_DIST Version 3.3 "out-of-the-box." It uses MPI-only within a node and uses Intel's MKL, a vendor BLAS library, running in single-threaded mode. This implementation is what we hope to improve by exploiting intranode parallelism. Unless otherwise noted, we try all numbers of MPI processes within a node up to 1 MPI process per physical core, and report the performance of the best configuration.
- `MKL`$_p$ is the same as `MKL`$_1$, but with multithreaded MKL instead. It uses 1 MPI process per socket; within each socket, it uses multithreaded MKL with the number of threads equal to the physical cores per socket.
- `{cuBLAS,Scatter}` is `MKL`$_p$ but with most GEMM calls replaced by their NVIDIA GPU counterpart, via the CUDA BLAS (or "cuBLAS") library. (Any other BLAS call uses `MKL`$_p$.) Additionally, cuBLAS may execute asynchronously; therefore, there may be an additional performance benefit from partial overlap between cuBLAS and Scatter, as the mnemonic name suggests. Like `MKL`$_1$, we try various numbers of MPI processes per node and re-

port results for the best configuration. (When there are more MPI processes than physical GPUs, the cuBLAS calls are automatically multiplexed.)

- `OpenMP+MKL`$_1$ exploits intranode parallelism *explicitly* using OpenMP. It parallelizes all phases using OpenMP. For phases that use the BLAS, we use explicit OpenMP parallelization and with single-threaded MKL. Scatter and GEMM phases run in sequence, i.e., they do not overlap.
- `OpenMP+{MKL`$_p$`,cuBLAS}` shares the work of the GEMM phase between *both* the CPU and GPU, running them concurrently. This tends to reduce the time spent in GEMM compared to `OpenMP+MKL`$_1$ implementation, but may not hide the cost completely.
- `OpenMP+{MKL`$_p$`,cuBLAS,Scatter}+pipeline` adds pipelining to `OpenMP+{MKL`$_p$`,cuBLAS}`. We use $n_s = 16$ CUDA streams and $N_b = 128$.

The first three implementations use implicit parallelism via multithreaded or GPU-accelerated BLAS; the last three involve explicit parallelism. We used $X_s = 144$ as maximum supernode size. To profile the computation's execution time, we use TAU. When we evaluate memory usage, we use the IPM tool [1].

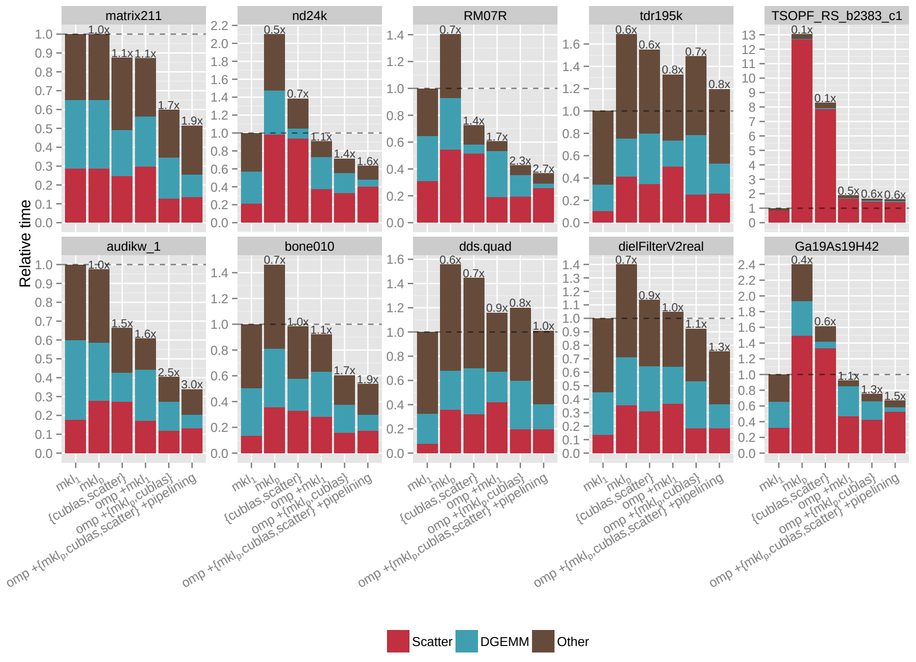


Fig. 3: Performance of different implementations for different test problems on Jinx cluster. Each bar is labeled by its speedup relative to the baseline (`MKL`$_1$).

Overall impact of intranode optimization. Our first analysis answers the question, by how much can *explicit* intranode optimization techniques improve performance above and beyond having a highly tuned multicore and/or GPU-accelerated BLAS? These experiments use just two nodes of the cluster. The results show best-case improvements of up to 3× using our techniques, and highlight scenarios in which our methods may yield a slowdown.

We show results for the Jinx system in fig. 3. (Dirac results are similar [7], and so omitted for space.) It shows time (y-axis) versus implementation variant (x-axis) for a given matrix. The time is normalized to the baseline, with actual baseline execution times in the range of 10 to 1,000 seconds (not shown). Each bar breaks down the execution time into its components, which correspond to different phases of SuperLU. The **GEMM** phase and **Scatter** phase are as described in § 4. The **Scatter** phase includes any CUDA stream setup and wait time. The "**Other**" phase has three major components: `MPI_Wait`, `MPI_Recv`, and triangular solve. When phases may overlap, the bar shows only the *visible* execution time, i.e., the part of the execution time that does *not* overlap. Thus, the total height of the bar is the visible wall-clock time.

Both the $\texttt{MKL}_p$ and `{cuBLAS,Scatter}` variants are slower or just comparable to $\texttt{MKL}_1$ in many cases. Though they may improve **GEMM**, **Scatter** and **Other** may slowdown since they tend to improve with more MPI processes. Thus, only relying on accelerating BLAS calls—whether by multithreading or offload—tends not to yield a significant overall speedup, and can in fact decrease performance.

The $\texttt{OpenMP+MKL}_1$ variant reduces the cost of **Scatter** and **Other** phases compared to $\texttt{MKL}_p$ and `{cuBLAS,Scatter}`. While **Other** for $\texttt{OpenMP+MKL}_1$ is better than with $\texttt{MKL}_1$, **Scatter** is worse. $\texttt{OpenMP+MKL}_1$ often matches the baseline $\texttt{MKL}_1$. The $\texttt{OpenMP+\{MKL}_p\texttt{,cuBLAS\}}$ variant reduces the time spent in **GEMM** compared to $\texttt{OpenMP+MKL}_1$ implementation, but cannot hide the cost of **GEMM** completely.

Our combined $\texttt{OpenMP+\{MKL}_p\texttt{,cuBLAS,Scatter\}+pipeline}$ implementation outperforms $\texttt{MKL}_1$ on 7 of the 10 test matrices on either platform, yielding speedups of up to 3× (fig. 3, `audikw_1`). Compared to $\texttt{MKL}_1$, this variant hides the cost of **GEMM** very well. However, **Scatter** still cannot achieve the same parallel efficiency as with $\texttt{MKL}_1$. The worst case occurs with `TSOPF_RS_bs2383_c1`, which derives from a power network analysis application. On Jinx, it is nearly 2× slower than $\texttt{MKL}_1$ (fig. 3). However, even with a slowdown our implementation can reduce the memory requirement of this problem; see below.

Strong Scaling. Part of the benefit of intranode parallelism is to enhance strong scaling. We consider this scenario, for configurations of up to 8 nodes and 64 cores (Dirac) or 96 cores (Jinx), showing results for Jinx in fig. 4. (Dirac results are better than Jinx [7].) We present results for just two "extremes": Matrix `nd24k`, on which our implementation does well, and `TSOPF_RS_b2383_c1`, on which it fairs somewhat poorly.

We focus on three of our implementation variants: the baseline $\texttt{MKL}_1$, $\texttt{OpenMP+MKL}_1$, and $\texttt{OpenMP+\{MKL}_p\texttt{,cuBLAS,Scatter\}+pipeline}$. For the $\texttt{MKL}_1$ variant, we use 1 MPI process per core. For $\texttt{OpenMP+MKL}_1$ and $\texttt{OpenMP+\{MKL}_p\texttt{,cuBLAS,Scatter\}+pipeline}$ cases, we use 1 MPI process per socket and one OpenMP thread per core.

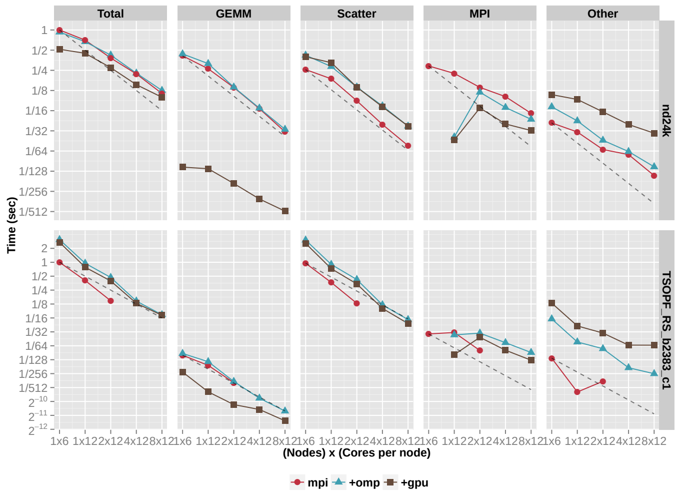


Fig. 4: Strong scaling on up to 8 nodes (96 cores and 16 GPUs) on Jinx

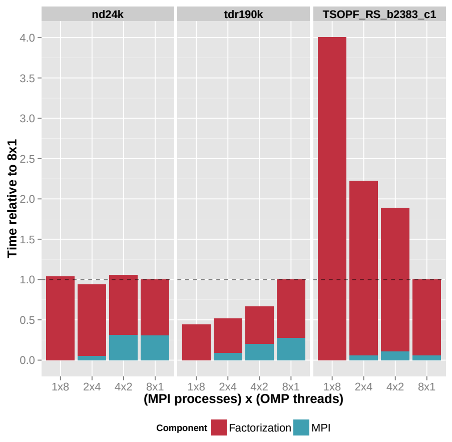
Figure 4 shows scalability as a log-log plot of time (y-axis) versus configuration as measured by the total number of cores (x-axis). Each series shows one of the three implementation variants. Each column is a phase, with the leftmost column, **Total**, showing scalability of the overall computation, inclusive of all phases. Time is always normalized by the *total* $\texttt{MKL}_1$ time when running on the smallest configuration (1 node and 1 socket), to reveal the relative time spent in each phase. Dashed lines indicate ideal linear speedup for $\texttt{MKL}_1$; perfect scaling would be parallel to this line, while sublinear scaling would have a less steep slope and superlinear scaling would have a more steep slope.

On Dirac (not shown), both test matrices exhibit good scaling behavior for nearly all the phases; by contrast, Jinx scaling (fig. 4) exhibits sublinear behavior. At 96 cores and 16 GPUs (2 GPUs per node), all three implementations differ by only a little on `nd24k`. This is due largely to the relatively poor scaling of the **Other** phase, which eventually becomes the bottleneck for `OpenMP+{MKL`$_p$`,cuBLAS,Scatter}+pipeline`.

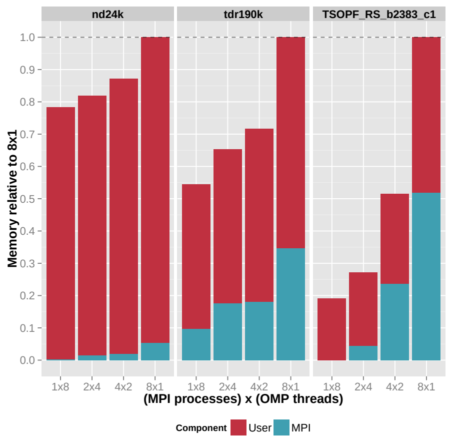
On `TSOPF_RS_b2383_c1`, the baseline $\texttt{MKL}_1$ is always fastest on both clusters, when it ran successfully. On Jinx, there was not enough memory per node to accommodate the 48 and 96 MPI processes cases, due to the fundamental memory scaling requirement of SuperLU_DIST; for more analysis, see below.

Matrix `TSOPF_RS_b2383_c1` case shows superlinear scaling. The **Scatter** phase is a major contributing factor in cost. As noted previously, the **Scatter** phase scales with increasing MPI processes, primarily due to better locality.

Overall, `OpenMP+MKL`$_1$ shows good strong scaling. By contrast, the scaling of `OpenMP+{MKL`$_p$`,cuBLAS,Scatter}+pipeline` can be worse, as observed on Jinx. How-



(a) Time in MPI vs. compute



(b) User vs. MPI-runtime memory

Fig. 5: Effect of intranode threading on memory and time

ever, this owes largely to Amdahl's Law effects due to **Other**. That component is primarily a triangular solve step, which our work has not yet addressed.

Time and memory requirements. Sparse direct solvers like SuperLU_DIST may exhibit a *time-memory tradeoff*. We show an example on three representative problems in fig. 5. This example includes `nd24k`, which shows common-case behavior; as well as `TSOPF_RS_b2383_c1`, which was a worst-case in execution time for our approach. The experiment tests the `OpenMP+MKL`$_1$ variant on one node of Dirac, which has 8 cores per node, under all configurations of (# of MPI processes) $\times$ (# of OpenMP threads) $= 8$.

Matrix `TSOPF_RS_b2383_c1` exhibits the time-memory tradeoff. The **Scatter** phase dominates execution time, as observed above; since **Scatter** scales with MPI processes, the all-MPI configuration wins. However, memory usage actually *increases* with increasing numbers of MPI processes. Among user allocated memory, it turns out that the memory required by the L and U factors remains fairly constant, whereas the buffers used for `MPI_Send` and `MPI_Recv` increase. Memory allocated by MPI runtime also increases. Thus, even if our intranode threading approach is slower than the all-MPI case, there can be a large reduction in the memory requirement.

6 Conclusions and Future Work

The high-level question this paper considers is how to exploit intranode parallelism in emerging CPU+GPU systems for distributed memory sparse direct solvers. At the outset, one expects a highly tuned multicore and/or GPU BLAS will yield much of the potential performance benefits. The real question, then, is how much *additional* performance gain is possible from explicit parallelization. Our results for SuperLU_DIST suggest that on today's systems, there may be up to a factor of 2× more to gain above and beyond BLAS-only parallelization.

Other avenues to pursue would include alternative accelerator platforms (e.g., Intel Xeon Phi, near-memory processing solutions); accelerating the Scatter phase, which requires extensive data structure changes; deeper architecture-dependent performance analysis; and evaluation of time-energy tradeoffs, which we believe are present intrinsically in the SuperLU_DIST algorithm.

References

1. IPM : Integrated performance monitoring. `http://ipm-hpc.sourceforge.net/`. Accessed: 2014-01-26.
2. Timothy A Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1, 2011.
3. 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.
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. `http://saahpc.ncsa.illinois.edu/09`.
5. Xiaoye S. Li and James W. Demmel. SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Trans. Mathematical Software*, 29(2):110–140, June 2003.
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. `http://vecpar.fe.up.pt/2010/papers/5.php`.
7. Piyush Sao, Richard Vuduc, and Xiaoye Li. A distributed CPU-GPU sparse direct solver. Technical report, Georgia Institute of technology, 2014.
8. O. Schenk, M. Christen, and H. Burkhart. Algorithmic performance studies on graphics processing units. *J. Parallel and Distributed Computing*, 68(10):1360–1369, 2008.
9. 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.
10. Ichitaro Yamazaki and Xiaoye S Li. New scheduling strategies and hybrid programming for a parallel right-looking sparse LU factorization algorithm on multicore cluster systems. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 619–630. IEEE, 2012.
11. S.N. Yeralan, T. Davis, and S. Ranka. Sparse QR factorization on gpu architectures. Technical report, University of Florida, November 2013.
12. C.D. Yu, W. Wang, and D. Pierce. A CPU-GPU hybrid approach for the unsymmetric multifrontal method. *Parallel Computing*, 37:759–770, 2011.