Locality Optimization on a NUMA Architecture for Hybrid LU Factorization

Adrien RÉMY [a], Marc BABOULIN [a], Masha SOSONKINA [b,1] and Brigitte ROZOY [a]

[a] *INRIA and University of Paris-Sud*
[b] *Old Dominion University (USA)*

Abstract We study the impact of non-uniform memory accesses (NUMA) on the solution of dense general linear systems using an LU factorization algorithm. In particular we illustrate how an appropriate placement of the threads and memory on a NUMA architecture can improve the performance of the panel factorization and consequently accelerate the global LU factorization. We apply these placement strategies and present performance results for a hybrid multicore/GPU LU algorithm as it is implemented in the public domain library MAGMA.

Keywords. ccNUMA, thread placement, dense linear systems, LU factorization, MAGMA library.

Introduction

On modern parallel systems, the main memory bandwidth plays a crucial role in high-performance computing (HPC) applications. On shared memory parallel computers, a large number of processors work on a common, shared physical address space. There are two types of shared memory systems that propose similar functionalities to the programmer but have different performance in terms of main memory access.

Unified Memory Access (UMA) systems consist of a single memory bank for which the latency and bandwidth are the same for all threads, regardless of the memory location. The downside of UMA systems is that, when many application threads are trying to access the main memory simultaneously, bandwidth bottlenecks can occur. To overcome this problem of scalability, architectures referred to as ccNUMA (cache coherent Non Uniform Memory Access) are commonly used in clusters of nodes. Recently, the ccNUMA has been adapted inside multicore nodes (see, e.g., [1] and the references therein). On ccNUMA systems, the memory is physically distributed but logically shared. The mechanism is transparent from the programmer point of view, the required protocols being handled by the hardware (*e.g.* HyperTransport for AMD and QuickPath for Intel). Each bank of

[1]The work of this author was supported in part by the AFOSR award FA9550-12-1-0476, NSF/OCI—0941434, 0904782, 1047772, and by the ODU award 533561.

memory is associated with a set of cores and this association forms a NUMA node. Due to this physical distribution, the performance of memory accesses varies depending on the mutual location of a given thread and the memory bank that the thread accesses. Accessing the remote memory banks may become slow and, if a lot of threads are used, this will affect the application scalability [2]. When using HPC applications on ccNUMA systems, we face two main difficulties. The first one is the locality problem. It happens when a thread located on a node accesses data stored in the memory bank of another node. This kind of nonlocal transfer can hurt performance. The second problem is contention, which occurs when two threads located on different nodes access memory in another node, and thus, fight for memory bandwidth. For each thread, the access to data should be restricted to its own node to avoid these two problems. If no particular data placement is proposed, the default memory affinity policy of the operating system is used. In most Linux-type operating systems, the default policy (called *first touch*) places the data in the memory node that is local to the thread that is writing the data first. This ensures fast access for the thread inside the node regardless of the other threads accessing the data [3]. In a multithreaded application, the fact that the master thread usually initializes multiple shared data structure can exacerbate the problem (all these shared data structures will be allocated in the same node as the master thread). This problem can be approached by using the software tools, such as the `libnuma` library [4] or the `likwid` software [5], that provide user interfaces to allocate memory into the desired nodes [4] or by initializing the data by multiple (possibly all) threads. The Servet Benchmark Suite [6] also provides an API to handle threads mappings based on communication or memory performance. Even if data locality is respected, the thread scheduling is important. If the scheduler ignores the locality information, the effect of caches is reduced. Switches into uncached process contexts will cause cache and TLB misses and cache line invalidations for the other processes [7]. The cost of thread scheduling can be reduced by moving thread management and synchronization to the user level [8].

In this paper, we study the effect of NUMA on the solution of dense general linear systems. To solve square linear systems $Ax = b$, the method commonly used is Gaussian Elimination with partial pivoting (GEPP). GEPP is performed as an LU factorization that decomposes the input matrix A into $L \times U$, where L is a unit lower triangular matrix and U an upper triangular matrix. Libraries, such as LAPACK [9], provide a block algorithm version of GEPP where the factorization is performed by iterating over blocks of columns (panels). LAPACK has been redesigned to use heterogeneous systems of multi/manycore CPUs and accelerators, such as GPUs. Examples of the redesign are PLASMA [10] and MAGMA [11], which take advantage of current multicore and hybrid multicore/GPU architectures [12]. In a classical LU factorization, the panel is first factored and then the trailing submatrix is updated using level 3 BLAS routines for high performance [13]. The update consisting of matrix-matrix products performed by the GPU is very efficient, making the panel factorization the bottleneck of the performance [14]. Indeed, due to its data access pattern the panel factorization is widely affected by memory access performance. By reducing the time of the panel factorization, we can improve the performance of the overall computation.

In the following we show how a proper placement of the threads and memory on a NUMA architecture can improve the performance of the panel factorization and consequently accelerate the global LU factorization as it is implemented in the MAGMA library for multicore/GPU systems. The paper is organized as follows. In Section 2.1, we describe the LU factorization algorithm for hybrid CPU/GPU architectures as implemented in MAGMA. Then in Section 2.2, we describe different strategies for thread pinning and data placement on NUMA architectures. Section 3 presents the experimental setup and performance results of LU factorization on a given platform using the different placement strategies given in Section 2.2. Concluding remarks are given in Section 4.

1. Locality optimization for LU algorithm

1.1. LU factorization for hybrid architecture

We consider here the hybrid LU factorization (right looking factorization [15, p. 85]) as implemented in the MAGMA library. The factorization is illustrated in Figure 1 where the computation is split so that the panel is factored on the CPU (black tasks) while the updates are performed on the GPU (gray tasks). The initial matrix has been downloaded to the GPU and we describe here a current iteration:

1. The current panel is offloaded to the CPU.
2. The panel is factored by the CPU and the result is sent back to the GPU.
3. The GPU updates in priority the column block that corresponds to the next panel in the trailing submatrix.
4. The updated panel is sent to the CPU and asynchronously factored on the CPU while the GPU updates the rest of the matrix.

The technique that consists of factoring the next panel while still updating the rest of the trailing submatrix is often referred to as *look-ahead* [16]. Depending on the problem size and on the hardware used, MAGMA proposes a default value for the width of the panel. Note that the communication between GPU and CPU is limited to the transfer of the successive panels.

In the remainder, we will consider two MAGMA implementations for the LU factorization. These two versions differ mainly in the way the panel is factored. In the first version, the panel is factored using GEPP (partial pivoting) while the second version does not pivot since it also uses randomization as a preprocessing to avoid pivoting [17]. We point out that in both MAGMA implementations the panel is factorized as a BLAS 3 algorithm where we consider an inner panel (factored using BLAS 2) inside the global panel. The size of this inner panel is set to 128 for the no pivoting version, and cannot be tuned for the partial pivoting when we use an MKL [18] implementation. Note that larger size of the panel results in more BLAS 3 operations, and thus, increasing the computation-to-memory access ratio in the panel factorization.

Due to the search for the pivot and to the subsequent row interchange, GEPP performs a lot of memory accesses, whereas they are minimal for the version

without pivoting. In the following we focus on the panel factorization, because its memory-bound characteristics make it particularly dependent on NUMA.

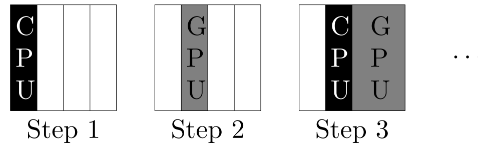


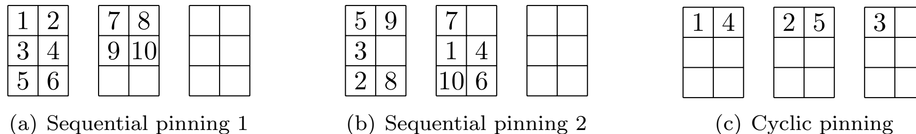
Figure 1. Hybrid LU factorization (4 panels).

1.2. Placement strategies

In this section we describe how threads may be bound to cores and how data may be placed in memory, which may be achieved using the following tools.

- Before each execution the data are placed in the node using the `mbind()` function from the `libnuma` [4] library.
- The threads may be pinned to the cores using the `sched_setaffinity()` Unix function, the `likwid` [5] or `numactl` [4] tools.
- Before each execution, using the same tools, the data may be placed in the nodes to which the threads are bound.

For the thread pinning, we consider the following strategies, which are illustrated in Figure 2 by considering three nodes of six cores. For all the different strategies the data are interleaved over the memory banks of the NUMA nodes used (i.e., the data are spread in a round-robin fashion in the memory pages across all the nodes).



(a) Sequential pinning 1 (b) Sequential pinning 2 (c) Cyclic pinning

Figure 2. Examples of pinning methods

No pinning: The threads and data are assigned automatically by the system, without manual intervention. We refer to this strategy as `noPin`.

Sequential pinning 1: The number of threads assigned to a node corresponds to its number of cores. When a given node is full while more threads need to be placed, the next thread will be “spilled” to the next node. In a sense, nodes are provided one by one for the thread placement purposes. An example is given in Figure 2(a), where we use 10 threads. Note that the thread placement *within* a node is not fixed explicitly and may be governed by the application. For example, the MKL Intel mathematical library may assign threads to cores dynamically. We refer to this strategy as `seqPin1`.

Sequential pinning 2: To avoid the problem of load balancing among the nodes when the number of threads is not a multiple of the number of cores per node, we allow the application to place threads evenly on several nodes at once. Specifically, these "occupied" nodes are taken as the first nodes that can accommodate all the threads. In Figure 2(b), the application will place them on the first two nodes, such that each node may have a free core. We refer to this strategy as `seqPin2`.

Cyclic pinning: The threads are cyclically placed onto all the nodes allotted to the application in a round-robin manner. For example in Figure 2(c), we use five threads in three nodes. Note that, although this approach is a rather load-balanced (the number of threads in each node may differ by one at most), it is not compact in terms of processing power. On the other hand, as will be shown later, its memory availability may be attractive. We refer to this strategy as `cycPin`.

Since the data are interleaved only among the nodes that are effectively used, sequential pinnings result in mostly local memory accesses. The memory accesses may incur extra latency for the cyclic pinning. However, for the cyclic pinning, there will be less competition among threads inside a node to access L3 cache. Moreover, the global amount of L3 cache available for all the threads will be larger for `cycPin` because this strategy uses more nodes. Note that if the number of threads used is the same as the number of cores available, the sequential and the cyclic pinning are equivalent in terms of data locality. In [19] and [20], we have investigated more sophisticated strategies for memory binding to NUMA nodes, which improved performance on sparse matrix computations with irregular matrix and vector access patterns. For the dense matrices and dynamic thread placement within a node, interleaving data in memory is sufficient mainly due to the dynamic nature of the thread placement by MKL.

2. Experiments

2.1. Experimental framework

Our experiments have been carried out using a MagnyCours-48 system. This machine is composed of four AMD Opteron 6172 processors running at 2.1GHz with twelve cores each (48 cores total) and 128GB of memory. Each processor contains two NUMA nodes with 6MB of L3 cache per node. Thus, we have 8 NUMA nodes of 6 CPU cores and 16 GB of main memory each. The GPU device is an NVIDIA Fermi Tesla S2050 with 448 CUDA cores running at 1.15 GHz and 2687 MB memory.

For a thread accessing memory in the same NUMA node the relative distance to the memory (latency) is taken as 10 in the ACPI specification [21]. The relative distances between the nodes are reproduced in Table 1 as obtained by the `numactl` tool [4]. For example, if node 0 accesses data stored in node 3, the cost of this access is 2.2 times larger than if the data were stored in node 0. The 8 nodes are linked by HyperTransport links. When a thread pinned on a core accesses data, if the data is located inside the same NUMA node as the core, the relative cost to

access the memory will be 10. If the data is located in the memory from a node directly connected by an HyperTransport link the cost will be 16. If the memory is in a node that is not directly linked to the current then the cost will be 22 (the data have to pass through 2 links).

Table 1. Node distances with respect to NUMA accesses.

node	0	1	2	3	4	5	6	7
0:	10	16	16	22	16	22	16	22
1:	16	10	22	16	22	16	22	16
2:	16	22	10	16	16	22	16	22
3:	22	16	16	10	22	16	22	16
4:	16	22	16	22	10	16	16	22
5:	22	16	22	16	16	10	22	16
6:	16	22	16	22	16	22	10	16
7:	22	16	22	16	22	16	16	10

We suppose now that the threads are scattered on all the nodes and that data are interleaved on the nodes. Then, assuming that each thread performs the same number of memory accesses on each node, we can compute the average memory access cost as $(10 + 16 + 16 + 22 + 16 + 22 + 16 + 22)/8 = 17.5$ in each of the eight nodes since all the rows in Table 1 have the same entries but in a different order. Therefore, the average memory access cost is 1.75 times larger than in the case of only local accesses.

2.2. Performance for the panel factorization

We test the performance of an LU panel factorization. This performance is expressed in Gflops/s. We measure it by summing the total number of flops executed in factoring successively each panel throughout the factorization and dividing it by the time spent in all the panel factorizations. The algorithms that we consider in our experiments are LU with partial pivoting and LU with no pivoting. The former uses the LAPACK implementation of GEPP (routine `dgetrf`) while the latter is a panel factorization with no pivoting (used in e.g., [22]), both linked with the multi-threaded BLAS from MKL.

In Figure 3, we compare the performance resulting from the different strategies of thread placements. The sequential pinning shown in the legend corresponds to the `seqPin2` strategy as described in Section 1.2, which provides a better load balance than `seqPin1`. For each type of placement, we measure the performance using a number of threads varying from 1 to 48.

When comparing the different types of pinning in Figure 3, we are interested in the peak performance obtained by each strategy and by the number of threads that enables us to obtain this rate. Indeed, since the scalability of the panel factorization is limited to a certain number of threads, it is not always recommended to use all the CPU cores available. In particular, using the 48 cores available in our experiments is never the most efficient solution. A first comment is that, as expected, the pivoting LU algorithms outperforms the nonpivoting one with at least a factor 4. This is consistent with the results obtained and analyzed in [22,23]

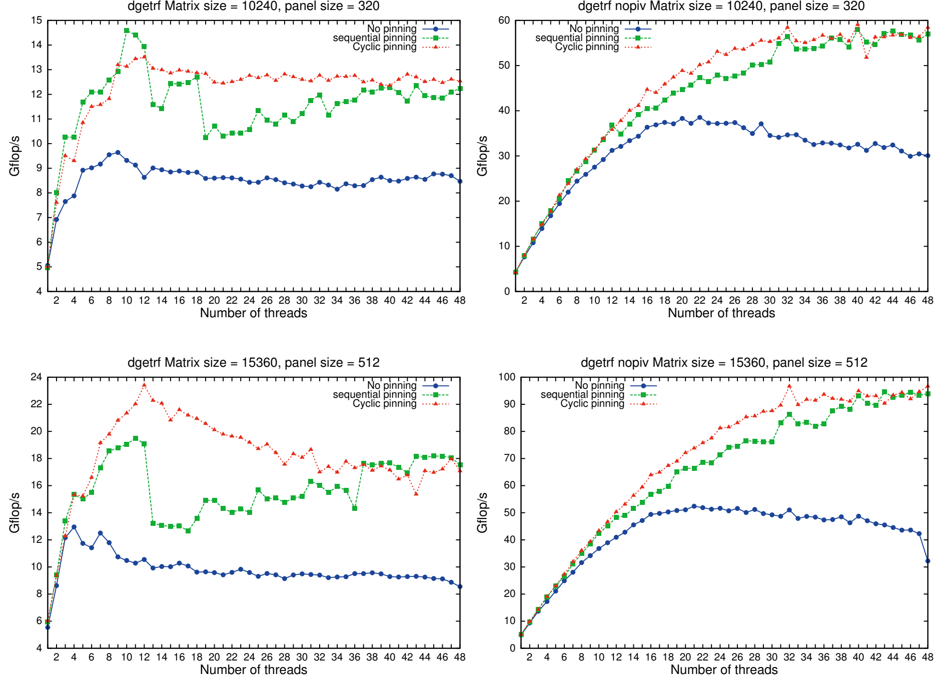


Figure 3. Performance of thread pinning strategies for LU panel factorization with pivoting (left) and no pivoting (right). Panel sizes: 10240×320 and 15360×512.

and confirms that the communication overhead due to pivoting is critical for factorizing the panel [17,24]. We observe that the sequential and cyclic pinnings give better results than `noPin`, and they are similar for larger numbers of threads (in the range 40-48).

For the partial pivoting case (`dgetrf`), the sequential pinning applied to a problem of size 10240 gives better performance than the cyclic pinnings for small thread counts due to a better data locality (i.e., because fewer NUMA nodes are involved). In our experiments, the best performance for size 10240 is obtained using 10 threads and consequently two nodes. We use nodes 0 and 1 which gives, using Table 1, an average memory access cost of 13 ($(10 + 16)/2$). The cyclic pinning gives better performance for the problem of size 15360, since there are more BLAS 3 operations that take better advantage of the cache and require fewer main memory accesses, possibly in remote NUMA nodes. We observe that for the sequential pinning, we have a performance drop for some number of threads, due to the addition of a new NUMA domain, namely when the number of threads is a multiple of 6, reducing then the data locality.

For the no pivoting case (`dgetrf_nopiv`), `cycPin` provides the best performance for all problem sizes. As expected, `dgetrf_nopiv` is less affected by data locality than `dgetrf` since there is no search for pivots, and thus, fewer memory access. Thereby, cache is used more efficiently, which is favored by the `cycPin` strategy that may make more cache available to threads due to the use of more nodes than for `seqPin2` in general. For example, if only one node is used, the amount of L3 cache available for the threads will be, on our architecture, 6MB and all the threads on the node will have to share it. If all of the 8 nodes are used then the memory accesses will be more expensive but the cache memory available will be $8 \times 6 = 48$ MB. Moreover, on this system the latency of the L3 cache is 20 ns, whereas the latency of the memory (inside a same node) is 60 ns. We also

mention that these behaviors (ratio of cache misses, number of memory accesses) have been confirmed by measurements using the PAPI [25] library.

2.3. Performance for the hybrid code

Let us evaluate the impact of the thread/data placement on a hybrid CPU/GPU LU factorization. In this case, as explained in Section 1.1, the panel is factored by the CPU while the updates are performed by the GPU. In these experiments, the CPU uses a fixed number of threads and the matrix size varies.

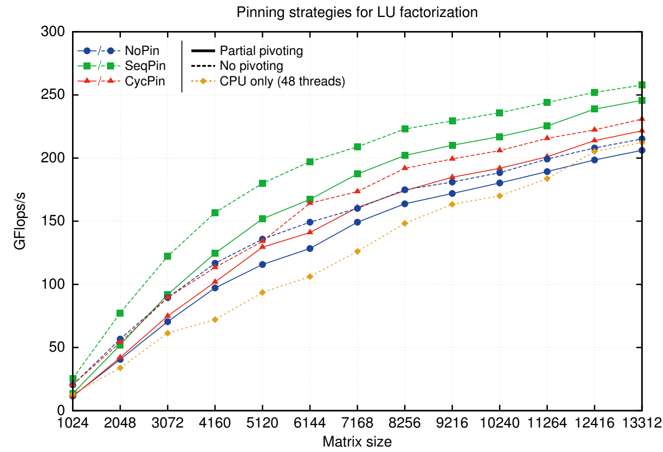


Figure 4. Performance for hybrid LU factorization with partial pivoting and no pivoting (12 threads).

Figure 4 compares the performance of the `seqPin2` and `cycPin` strategies with the `noPin` for the partial pivoting and no pivoting factorizations. The GPU was used along with the 12 CPU threads. For sake of comparison, we included in Figure 4 the performance for a CPU-only LU factorization using 48 threads (MKL implementation), the data being interleaved on all the nodes using the numactl tool. The NoPin curves represent the performance of the MAGMA codes without any modification. The SeqPin curves represent the performance of the MAGMA codes modified to have the threads pinned via the `seqPin2` strategy and the data placed only on the nodes that are actually used. The CycPin curves represent the performance with the threads pinned with the `cycPin` strategy with the data interleaved on the nodes.

We observe that the pinning methods outperform the `noPin` version and that for partial and no pivoting, the `seqPin2` strategy gives the best performance. Note that the difference of performance between the pivoting and no pivoting routines is smaller that for the panel factorization as depicted in Figure 3. Indeed, the update phase represents most of the computation and is performed by the GPU and the cost of the panel factorization has less impact on the global performance [22,26]. Note also that asymptotically, when the matrix size increases and as mentioned in [17], the performance of the pivoting and no pivoting LU should be close

because communication involved in pivoting becomes negligible compared to the $\mathcal{O}(n^3)$ computations for large dimensions.

3. Conclusion

In this work we studied different methods (referred to as sequential and cyclic pinning) to place threads and data for an LU factorization algorithm executed on NUMA architecture using GPU accelerator. The two methods of placement improve the performance up to 2x compared with the default memory and thread placement. The choice of the most efficient method depends on the data pattern access of the algorithm and on the ability of the implementation to take advantage of cache memory. This choice is also influenced by the size of the problem. Consequently to this work we will develop a heuristic to choose automatically the best placement strategy. This technique has been implemented as an external function that can be easily applied to other algorithms with panel blocking and we are currently applying a similar approach to the QR factorization algorithm.

Acknowledgements

We are grateful to Jack Dongarra and Stanimire Tomov (Innovative Computing Laboratory, University of Tennessee) for allowing us to perform experimentation on their multicore/GPU machines.

References

[1] G. Hager and G. Wellein. *Introduction to High Performance Computing for Scientists and Engineers.* CRC Press, 2011.

[2] C. Lameter. Local and remote memory: Memory in a linux/numa system. In *Linux Symposium (OLS2006), Ottawa, Canada*, 2006. `ftp://ftp.tlk-l.net/pub/linux/kernel/people/christoph/pmig/numamemory.pdf`.

[3] R. Iyer, H. Wang, and L.N. Bhuyan. Design and analysis of static memory management policies for cc-numa multiprocessors. *Journal of systems architecture*, 48(1):59–80, 2002.

[4] A. Kleen. A numa api for linux. Technical report, Novel Inc, 2004. `http://www.halobates.de/numaapi3.pdf`.

[5] J. Treibig, G. Hager, and G. Wellein. LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures*, San Diego CA, 2010.

[6] J. Gonzàlez-Domìnguez, G. L. Taboada, B. B. Fraguela, M. J. Martín, and J. Touriño. Automatic mapping of parallel applications on multicore architectures using the servet benchmark suite. *Computers & Electrical Engineering*, 38(2):258 – 269, 2012.

[7] F. Bellosa and M. Steckermeier. The performance implications of locality information usage in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 37(1):113 – 121, 1996.

[8] T. E. Anderson, E. D. Lazowska, and H. M. Levy. The performance implications of thread management alternatives for shared-memory multiprocessors. *IEEE Transactions on Computers*, 38(12):1631–1644, 1989.

[9] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide.* SIAM, Philadelphia, 3 edition, 1999.

[10] University of Tennessee. *PLASMA Users' Guide, Parallel Linear Algebra Software for Multicore Architectures, Version 2.3.* 2010.

[11] M. Baboulin, J. Demmel, J. Dongarra, S. Tomov, and V. Volkov. Enhancing the performance of dense linear algebra solvers on GPUs in the MAGMA project. *Poster at Supercomputing*, 8, 2008.

[12] M. Baboulin, J. Dongarra, and S. Tomov. Some issues in dense linear algebra for multicore and special purpose architectures. In *9th International Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA'08)*, volume 6126-6127 of *Lecture Notes in Computer Science*. Springer-Verlag, 2008.

[13] E. Anderson and J. Dongarra. Evaluating block algorithm variants in LAPACK. Technical report, April 1990. (LAPACK Working Note #19).

[14] S. Tomov, J. Dongarra, and M. Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing*, 36(5&6):232–240, 2010.

[15] J. Dongarra, I. Duff, D. Sorensen, and H. van der Vorst. *Numerical Linear Algebra for High-Performance Computers.* SIAM, Philadelphia, 1998.

[16] J. Kurzak and J. Dongarra. Implementing linear algebra routines on multi-core processors with pipelining and a look ahead. *LAPACK Working Note 178*, September 2006.

[17] M. Baboulin, J. Dongarra, J. Herrmann, and S. Tomov. Accelerating linear system solutions using randomization techniques. *ACM Trans. Math. Softw.*, 39(2), 2013.

[18] Intel. *Math Kernel Library (MKL).* `http://www.intel.com/software/products/mkl/`.

[19] A. Srinivasa and M. Sosonkina. Nonuniform memory affinity strategy in multithreaded sparse matrix computations. In *Proceedings of the 2012 Symposium on High Performance Computing*, HPC '12, pages 9:1–9:8, San Diego, CA, USA, 2012.

[20] A. Srinivasa, M. Sosonkina, P. Maris, and J.P. Vary. Effcient shared-array accesses in ab initio nuclear structure calculations on multicore architectures. *Procedia CS*, 9:256–265, 2012.

[21] Hewlett-Packard Corporation, Intel Corporation, Microsoft Corporation, Phoenix Technologies Ltd., Toshiba Corporation. *ADVANCED CONFIGURATION AND POWER INTERFACE SPECIFICATION 4.0a*, April 2010. `http://www.acpi.info/DOWNLOADS/ACPIspec40a.pdf`.

[22] M. Baboulin, S. Donfack, J. Dongarra, L. Grigori, A. Rémy, and S. Tomov. A class of communication-avoiding algorithms for solving general dense linear systems on CPU/GPU parallel machines. In *International Conference on Computational Science (ICCS 2012)*, volume 9 of *Procedia Computer Science*, pages 17–26. Elsevier, 2012.

[23] S. Donfack, L. Grigori, and A. K. Gupta. Adapting communication-avoiding LU and QR factorizations to multicore architectures. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–10. IEEE, 2010.

[24] L. Grigori, J. W. Demmel, and H. Xiang. Communication avoiding Gaussian elimination. 2008. In Proceedings of the IEEE/ACM SuperComputing SC08 Conference.

[25] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *International Journal of High Performance Computing Applications*, 14(3):189–204, 2000.

[26] S. Donfack, J. Dongarra, M. Faverge, M. Gates, J. Kurzak, P. Luszczek, and I. Yamazaki. On algorithmic variants of parallel gaussian elimination: Comparison of implementations in terms of performance and numerical properties. Technical report, Innovative Computing Laboratory, University of Tennessee, jul 2013. University of Tennessee Computer Science Technical Report (also LAWN 280).