

Lawrence Berkeley National Laboratory

LBL Publications

Title

Cross-scale efficient tensor contractions for coupled cluster computations through multiple programming model backends

Permalink

<https://escholarship.org/uc/item/2q73t2t4>

Authors

Ibrahim, Khaled Z
Epifanovsky, Evgeny
Williams, Samuel
et al.

Publication Date

2017-08-01

DOI

10.1016/j.jpdc.2017.02.010

Peer reviewed

Cross-scale Efficient Tensor Contractions for Coupled Cluster Computations Through Multiple Programming Model Backends

Khaled Z. Ibrahim^{1a}, Evgeny Epifanovsky^b, Samuel Williams^a, Anna I. Krylov^c

^a*Computational Research Division, Lawrence Berkeley National Laboratory
Berkeley, CA, USA*

^b*Q-Chem, Inc*

6601 Owens Drive, Suite 105, Pleasanton, CA 94588, USA

^c*Department of Chemistry*

University of Southern California, Los Angeles, CA, USA

{kzibrahim, swilliams}@lbl.gov, {epifanov, krylov}@usc.edu

Abstract

Coupled-cluster methods provide highly accurate models of molecular structure through explicit numerical calculation of tensors representing the correlation between electrons. These calculations are dominated by a sequence of tensor contractions, motivating the development of numerical libraries for such operations. While based on matrix-matrix multiplication, these libraries are specialized to exploit symmetries in the molecular structure and in electronic interactions, and thus reduce the size of the tensor representation and the complexity of contractions. The resulting algorithms are irregular and their parallelization has been previously achieved via the use of dynamic scheduling or specialized data decompositions. We introduce our efforts to extend the Libtensor framework to work in the distributed memory environment in a scalable and energy-efficient manner. We achieve up to $240\times$ speedup compared with the optimized shared memory implementation of Libtensor. We attain scalability to hundreds of thousands of compute cores on three distributed-memory architectures, (Cray XC30 and XC40, and IBM Blue Gene/Q), and on a heterogeneous GPU-CPU system (Cray XK7). As the bottlenecks shift from being compute-bound DGEMM's to communication-bound collectives as the size of the molecular system scales, we adopt two radically different parallelization approaches for handling load-imbalance, tasking and bulk synchronous models. Nevertheless, we preserve a unified interface to both programming models to maintain the productivity of computational quantum chemists.

Keywords: Tensor Contraction Engines; Quantum Chemistry; Libtensor; Cyclops. High Performance Computing; Distributed Memory Programming Models; Energy Efficiency

1. Introduction

Tensor contraction routines are important kernels for quantum chemistry computation. The tuning of these kernels is notoriously complex because of inherent computational characteristics such as load imbalance, complex locality, memory requirements, etc. Multiple frameworks provide

¹Corresponding author: Khaled Ibrahim, kzibrahim@lbl.gov, voice: (510)486-7467, fax: (510)486-6900

interfaces to easily express the required computation by domain scientists. Mapping and executing these expressions on the underlying architecture is the responsibility of the runtime.

Choosing the programming model abstractions for the most efficient use of the system is a common challenge for many computational disciplines. Often, runtimes choose one abstraction (e.g. distributed memory) and apply it to different architectures and concurrencies. The computational scientist often selects one computational environment because of familiarity or develops multiple variants of the computations to maximize efficiency across differing architectures. Obviously, this makes efficient utilization of computational resources and code maintenance difficult.

Recent architectural developments targeting power efficiency motivates the use of lightweight manycore- or accelerator-based architectures. Depending on the available memory capacity per node, the computation dataset may fit in the shared memory or distributed memory. These architectural variants influence the efficiency of programming abstractions and the optimization strategies.

In this paper, we introduce our effort to extend Libtensor framework to work in distributed memory environment. Although, we could have extended the tasking model used in Libtensor to work in a distributed environment, our analysis shows that the associated data movement could impact the efficiency at scale. Instead, we leverage the Cyclops Tensor Framework (CTF) to achieve efficient data movement. We complement the CTF support of permutational symmetry with special handling for other forms of symmetry properties needed for quantum chemistry computations, including point-group and spin symmetries. Additionally, we exploit the capabilities of CTF to enable GPU acceleration. The presented work will be integrated to the Libtensor release [1, 2]. In addition to enabling the solution of new large-scale problems with highly-accurate coupled-cluster single double (CCSD) excitation methods, this extension provided a $240\times$ speedup over the optimized shared memory implementation of Libtensor.

This study shows the need to support tensor contraction engines with multiple programming backends to achieve the most efficient use of architectural resources. These backends are hidden from computational scientists behind a unified user interface. Each backend relies on a distinct programming model. This way the computational scientist could express his intended computation once, while the runtime provides the backend best suited for the target architecture. This multi-backend approach is required for efficient data movement, which is one of the major scaling bottleneck in future extreme-scale computing architectures. The impact of the choice of the programming model on data movement has not been previously thoroughly investigated. We show in this study through empirical measurements the impact of the programming model on communication performance, hoping this experience and methodology benefits other application domains. The first backend relies on CTF, which uses a special cyclic data distribution for load-balancing computation and a contraction engine that adopts a communication-avoiding adaptation of the SUMMA [3] for matrix-matrix computation. It uses a complex data mapping and distribution and coordinated communication primitives. The second uses a task-based model, which utilizes dynamic load-balancing without disruptive changes to the data layout.

We contrast Libtensor multi-backend approach with NWChem, which extends the use of tasking models to distributed memory machines. We show that at scale, when performance becomes communication-bound, the coordinated transfer approach (MPI collectives) used by the CTF backend yields less data movement. At small scale, on a shared memory node architecture, the task-based approach provides the best opportunity to leverage all symmetry properties of the computation to maximize performance. We also present a strategy for improving energy-to-solution by up to 29% based on dynamic voltage and frequency scaling (DVFS). We show the importance of adopting

Table 1: Test cases for CCSD calculations.

	P1	P2	P3	P4	P5
Molecular system	uracil	methylated uracil–water dimer	methylated uracil–water cluster	nucleobase tetramer (AATT) with FNO approximation	nucleobase tetramer (AATT)
Basis set	cc-pVTZ	(mU–H ₂ O) 6-311+G(d,p)	(mU) ₂ –H ₂ O 6-31+G(d,p)	6-311+G(d,p)	6-311+G(d,p)
Basis functions	296	302	489	968	968
Point group symmetry	C_s	C_s	C_1	C_1	C_1
Occupied/virtual orbitals (by irrep)	16/176 (A')	23/184 (A')	58/410	98/552	98/830
	5/91 (A'')	8/76 (A'')			

such a technique increases as the problem becomes communication-bound. Ultimately, this work demonstrates the need to use multiple programming models for efficient execution. Simultaneously, we show that these distinct models can be hidden from computational scientists using a unified interface.

The rest of this paper is organized as follows: Section 2 discusses related work. Section 3 presents the experimental setup, the workload, and the software stack of Libtensor framework. In Section 4 we discuss the computational characteristics of tensor contractions in quantum-chemistry computations and the influence of programming models on optimization strategies. In Section 5 we present the performance evaluation of the software stack and the backend that delivers the best performance for each architecture. In Section 6 we show the influence of the programming model on data movement and thus the performance and energy. We conclude in Section 7.

2. Related work

Tensor computations are a natural extension of matrix computations to many dimensions; they are used to solve many-body problems in physics, in particular in quantum chemistry and nuclear physics. Most of the tools previously developed are specific to their applications and lack transferability. Among those targeting large-scale distributed systems are TCE [4], the ACES/SIAL framework [5], CFOUR [6], which target both distributed and shared-memory architectures, are also quantum chemistry application-specific. Recently, general-purpose tensor tools started to receive more attention, for example CTF [7] and TiledArray [8].

Generally speaking, there are two primary challenges that all of these tensor tools face when converting a physical problem into an efficient computer program. First, they need to provide an interface that allows one to describe the physical problem as a set of tensor equations, and second, there needs to be a runtime environment that is capable of computing those equations. TCE and SIAL handle the complexity by enabling users to write an electronic structure operator and compute its tensor representation. Programming is done through code generation in Fortran (TCE) or a domain-specific programming language (SIAL). CTF and TiledArray provide a general programming interface making them open to applications outside of electronic structure theory. A notable example of using CTF in quantum chemistry computations is provided by the Aquarius package [9, 10].

From the parallel communication standpoint, the packages mentioned here use either a one-sided model, for instance the use of global arrays in NWChem [11] and GTFock [12], or a two-sided model such as MPI in CTF [7]. While providing attractive solutions for large-scale runs, they

typically suffer from large overheads on small machines. They typically rely on the aggregation of the physical memory of many lightweight compute nodes in order to accommodate a large problem. Working in a distributed memory environment influences many of their design choices. For instance, exploitation of some form of symmetry and sparsity can lead to complex communication patterns. All of these packages face similar challenges, including how to load-balance the computation, how to manage locality, and how to efficiently execute computation while providing an easy interface. They use either a tasking model for load-balancing computation with simple data layout [13, 11, 12], or a complex data layout with bulk synchronous communication [7]. A single programming model is used by these packages for all problems and at all run scales.

Extending tasking-based programming models to distributed environment has been successfully realized by multiple runtimes including Charm++ [14], HPX [15], StarPU [16]. These models provide primitives and patterns to load-balance workloads based on the idea of splitting computation into manageable tasks. The synchronization is used mainly for dealing with dependencies between tasks, thus avoiding global coordinated synchronization in bulk synchronous models. Locality or data movement coordination is typically a second-tier objective in these programming models. The choice between bulk-synchronous models and tasking models is greatly influenced by the degree of load imbalance, the computation to communication ratio, whether data layout and algorithms that enable static load-balancing exist or not, etc. In this paper, we focus on tensor contraction operations used in quantum chemistry to contrast these models.

Libtensor [1] started as a general-purpose tensor algebra library that targets problems that can be solved within a single node. It adopted the shared-memory programming model and exploits multiple forms of symmetry including permutational symmetry, spin symmetry, and point group symmetry to minimize memory usage and floating point operation count. This work extends Libtensor support to the distributed memory environment using multiple backend programming models with a unified interface. We extended Libtensor to transparently choose the best performing programming model without the need to interact with multiple computational chemistry packages.

3. Experimental setup

3.1. Molecular Systems

Table 1 summarizes computational test cases used in this study. These problems represent typical molecular systems amenable to high-level electronic structure calculations. All the molecules used here are of a closed-shell type; thus, they exhibit spin symmetry (equality between α and β molecular orbitals). Tests P1 and P2 also feature point-group symmetry, resulting in the sparsity of tensor quantities expressed in the molecular-orbital basis. In addition, some tensors are symmetric (or anti-symmetric) with respect to the permutation of indices. These symmetries can be exploited to reduce the amount of data and computations. Tests P4 and P5 have a large memory footprint, requiring hundreds of terabytes.

The Libtensor interface allows computational scientist to express different symmetry properties of tensors. Figure 1 shows the different forms of symmetry that Libtensor supports: permutational symmetry, spin symmetry and point-group symmetry. The importance of such optimizations increases with the scale and the dimensionality of the problem. These properties are expressed at the tile granularity (as opposed to element-wise), thus tiles may have extra padding. By defining the data properties rather than structuring the data explicitly, the mathematical expressions (contractions) are greatly simplified using an equivalent to dense format expressions. Internally, the library executes the computation while considering these properties.

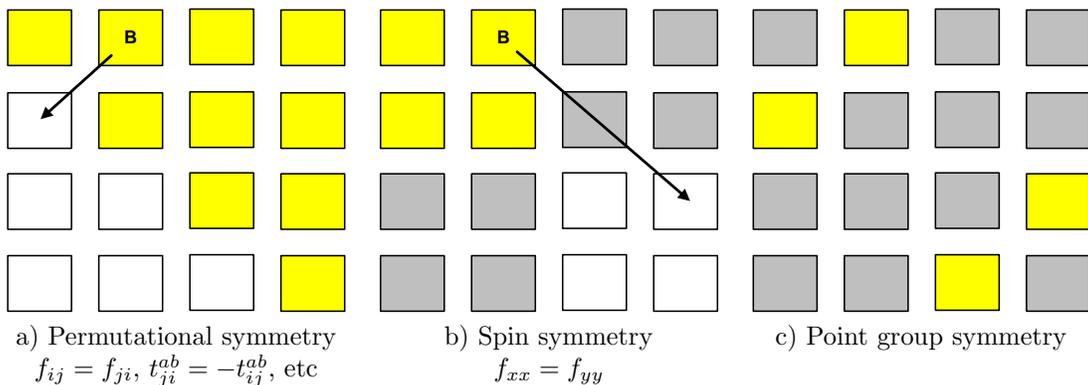


Figure 1: Libtensord supports multiple forms of symmetry properties: permutational, spin, and point group. Gray areas carries zero block, while white areas can be deduced from canonical blocks. Leveraging these symmetry properties in reducing the memory footprint typically causes load imbalance in the computation.

3.2. Evaluated Computer Systems

HPC systems are growing in diversity to achieve energy efficiency. The studied systems, shown in Table 2, represent a diverse set of architectures. Edison and Cori (Cray XC) use complex out-of-order multi-issue multicore architectures (Ivy Bridge and Haswell respectively), whereas Mira (IBM Blue Gene/Q (BGQ)) uses lightweight simple in-order cores. Conversely, Titan (Cray XK7) relies on NVIDIA GPU accelerators to improve the computational capability. All these architectures have moderate memory capacity per node, thus large problems require the use of many of these nodes to ensure the problem fits in the physical memory. By contrast, the Carver node, an older four-socket Nehalem Xeon, couples very large memory capacity (1 TB) to accommodate large problems coupled with out-of-core processing efficiently implemented using local disk storage attached to the compute node. Systems relying on distributed memory use specialized interconnects to efficiently transfer data between compute nodes. The connection radix increases from 3D torus, to 5D torus, to dragonfly with a corresponding reduction in diameter.

Coupled-cluster computations typically involve many tensor contractions that are mapped to matrix-matrix multiplications. Physical properties may be leveraged to optimize the computation as discussed in Section 4. As the computations were dominated by DGEMM calls at small scale, on all systems we used vendor optimized and threaded implementations of BLAS routines to maximize performance.

3.3. Computational Chemistry and Libtensord

Coupled-cluster theory offers a hierarchy of practical, highly accurate and systematically improvable methods for modeling electronic structure. Programming expressions in coupled-cluster theory are most conveniently written in a tensor form, resulting in equations involving contractions and other operations on multidimensional quantities.

Libtensord aids in expressing tensor computations by providing a simple programming interface that enables one to code equations in a form that closely resembles actual mathematical expressions, partly shown in Table 3 with details in [2]. This programming interface can be viewed as a domain-specific embedded language (DSEL) [17, 18] using C++ interface. The expressions are backed by a native Libtensord computational engine that uses a task-based programming model on shared

Table 2: Systems used in this study.

	Edison	Titan	Mira	Cori	Carver	
Processor	Intel Xeon E5-2695 v2	AMD Opteron 6274	NVIDIA Kepler K20X	IBM BGQ	Intel Xeon E5-2698 v3	Intel Xeon X7550
Clock (GHz)	2.4	2.2	0.733	1.6	2.3	2.0
NUMA×Cores	2×12	2×8	1×14	1×16	2×16	4×8
DP GFlop/s	461	141	1314	204	1178	256
Memory (GB)	64	32	6	16	64	1000
System and System Software						
Nodes	5,576	18,688	49,152	1630	1	
Interconnect	Dragonfly	3D torus + local PCIe		5D torus	Dragonfly	N/A
Compiler	Intel 16.0.0	Intel 13.0.1	nvcc 6.5	XLC	Intel 16.0.0	Intel <i>icc</i> 13.0.1
BLAS	MKL 11.3.0	MKL 11.0.0	cuBLAS 6.5	ESSL	MKL 11.3.0	MKL 11.0.1

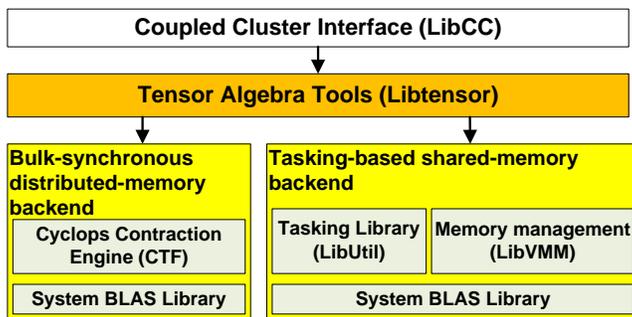


Figure 2: Libtensor stack for coupled-cluster computations with newly introduced distributed memory support. Coupled-cluster computations are expressed using a unified interface. The choice of the backend depends on the target architecture.

memory parallel computers. The software stack based on Libtensor is used in the implementation of coupled-cluster, equation-of-motion coupled-cluster, and algebraic diagrammatic construction methods within Q-Chem [19], a general-purpose electronic structure package. While providing a significant advantage in the speed of code development, Libtensor demonstrates performance that is competitive with the best codes based on explicit matrix multiplication and data handling [8].

In this work, the programming interface and expression evaluation component of Libtensor are extended to support multiple backends. This new capability makes it possible to deliver several tensor engines in a single binary executable so that the most efficient backend can be selected, as shown in Figure 2. In addition to the native shared memory engine, Libtensor features a newly introduced backend that targets distributed memory computers using CTF, which uses a bulk synchronous programming model (MPI). The choice of a backend is based on the target architecture and is chosen transparently by the runtime. However, we allow the user to override the default choice of the backend, for instance, by using the distributed memory implementation in a shared memory environment.

Table 3: Libtensor expressions programming interface.

Expression	C++ code
$c_{ijkl} = \sum_m a_{ijm} b_{mlk}$	<code>c(i j k l) = contract(m, a(i j m), b(m l k));</code>
$c_{ijk} = a_{ijk} b_{kji}$	<code>c(i j k) = mult(a(i j k), b(k j i));</code>
$c_{ijkl} = a_{ik} b_{jl}$	<code>c(i j k l) = a(i k) * b(j i);</code>
$c = \sum_{ijk} a_{ijk} b_{ijk}$	<code>c = dot_product(a(i j k), b(i j k));</code>
$c_{ij} = a_{ij} + a_{ji}$	<code>c(i j) = symm(i, j, a(i j));</code>
$c_{ij} = a_{ij} - a_{ji}$	<code>c(i j) = asymm(i, j, a(i j));</code>

3.4. Shared Memory Task-based Backend

The original backend used in Libtensor uses a tasking execution model for shared-memory between threads [1]. Shared memory provides a unified name space thus simplifying expressing complex data layout. The main complexity associated with leveraging the symmetry properties is that the computational cost per equal tile becomes variable, depending on whether a tile is stored in its canonical fashion or a symmetry operator needs to be applied. The tasking model facilitates dynamic computational load-balancing, but typically makes it harder to optimize for locality. The tasking model incurs scheduling overheads at queueing or dequeuing tasks. Distribution of task granularities typically shows a wide variation with dominance of small tasks [20]. Achieving higher concurrency also involves using smaller blocks. As such, a simple heuristic for load-balancing is important in attaining low-overhead scheduling. We find it is often better not to coordinate read access of a given tile by multiple threads as such coordination leads to excessive synchronization and impedes computational load balancing.

3.5. Distributed-Memory, Bulk-Synchronous Backend

In this paper, we extend Libtensor to interface with the Cyclops Tensor Framework (CTF) framework [7] in order to provide capability in a disk-less, distributed memory environment. CTF support is nominally limited to permutational symmetry. To support other symmetry properties, such as spin-symmetry and point-group symmetry, we adopted the approach of applying these symmetry properties, by creating sub-tensors, before handing them over to CTF for managing the distributed data layout. CTF uses communication-avoiding techniques to reduce the volume of communicated data coupled with a specialized memory mapping and data distribution on a virtual grid of processes. Moreover, CTF leverages available memory to store redundant copies of the data, thus eliminating a substantial fraction of the data movement from the critical path of computation. Finally, CTF creates a special mapping between the global data structure and the local data structure that preserves the permutational symmetry properties and creates a balanced distribution of the computation.

A user-level Libtensor contraction operation does not typically map to a single call to CTF because libtensor supports three forms of symmetry for efficient memory usage and computation, while CTF only directly supports permutational symmetry. As such tensor generation and contraction through Libtensor could differ from a direct implementation on CTF. Typically, a Libtensor contraction involves multiple CTF contractions. To match our performance on CTF, a domain scientist would need to manually implement the logic to leverage point-group symmetry and spin symmetry for high dimensional tensors. Therefore, leveraging CTF through the productive Libtensor interface automates a tedious error-prone process.

4. Coupled-Cluster Computations

Optimized coupled-cluster computations typically leverage physical properties to speed the computational kernels. For instance, exploiting symmetry allows for a reduction in the requisite memory and computation. On the other hand, it poses challenges in load-balancing the computation, managing locality, data distribution, regularity of the communication, and regularity of the computation. A programming model could influence how to tackle these challenges. There is also an interaction of techniques to handle these issues especially while exploiting symmetry properties to reduce memory and computation requirements. In this section, we discuss these challenges and the interaction with different programming models.

4.1. Tensor Index Mapping

Tensors are multidimensional arrays that, at their simplest form, could be indexed using a multidimensional linear mapping. Many implementations use a more complex indexing of the elements to improve locality. For instance, explicitly tiled blocks could be used to improve access locality. Libtensor uses this approach in its shared memory backend. Global address space abstractions adopted by *global arrays* [21] allow extending that simple indexing to distributed memory, *e.g.* in NWChem. In a distributed environment, the mapping function typically involves the process rank. The processors could be thought to have a linear topology, 2D torus topology, or higher dimension torus. A complex mapping function for indexing could be used to leverage permutational symmetry for saving memory while allowing simple data distribution of the tensor into a regular multidimensional processors topology. Two notable packages using this mapping are HPL on a 2D virtual torus, and CTF on a multi-dimensional virtual torus topologies, *e.g.* IBM BGQ. The mapping of the virtual processor topology to the physical topology could be challenging on some architectures, *e.g.* dragonfly in Cray XC30.

4.2. Tensor Data Distribution

The data distribution of a tensor is critical to performance both in distributed memory machines and on shared memory machines with non-uniform memory access (NUMA) attributes. With a linear index mapping, the data distribution is typically more complex whereas a simple distribution creates severe imbalance. Optimized distributed memory applications typically use a cyclic (or round robin) distribution to balance the data distribution. The same approach is used in distributed memory models using the partitioned global address space (PGAS) abstraction, such as global arrays. Even in shared memory machines with non-uniform memory access (NUMA), such a cyclic distribution is essential for improving performance [20]. A more complex indexing, such as that used in CTF, allows a regular distribution of data using a mapping function between the tensor dimension and the processes based on a virtual layout.

4.3. Tensor Index Mapping and Symmetry

A linear mapping (including the use of blocking) is typically the easiest means of expressing different forms of symmetry including permutational symmetry, point group, and spin symmetry. This simple mapping is adopted in the shared memory backend of Libtensor and the distributed memory NWChem.

CTF successfully showed that permutational symmetry could be expressed with a more complex mapping that allows a balanced data distribution. Expressing other forms of symmetry is a more tedious task because of the difficulty in maintaining a balanced distribution of data. Thus, the

distributed memory backend uses Libtensord’s native capabilities to express point group and spin symmetry, but delegates the handling of permutational symmetry to CTF. Preserving symmetry properties while applying operators is influenced by the data representation, tiling factor, etc. Therefore, the CTF-based implementation does not handle symmetry properties as efficiently as the case with a simple linear mapping and thus impedes our efforts to use it as a distributed memory back end.

4.4. Load Imbalance in Coupled Cluster Computations

In the Libtensord framework, the load imbalance in tensor contractions for coupled-cluster computations stems from the fact that part of the data is stored in their canonical form, while others could be deduced from applying a symmetry operator on the canonically stored blocks. Additionally, some data blocks may contain solely zeros thus nullifying the need to perform any computations. As such, the computation cost involving the same amount of data could significantly differ based on how the data is stored. Trying to balance computation typically increases the chance of accessing non-local data, thus creating another source of load imbalance: the non-uniform function of the cost of moving the same amount of data. One additional source of load-imbalance is that tiling data rarely yields equal partitions because of the irregularity of the array dimensions. The dimensions of a tensor are typically dictated by the physical properties of the studied problem. All these factors lead to wide range of task granularity, which was analyzed in a related study [20].

Distribution of computation in a balanced fashion requires accurate estimates of the cost of doing computation and an accurate estimate for the cost of moving data to the compute engine. Distribution of work could also be simplified if the granularity of the work assignment is constant or has a small variation. Dealing with load-imbalance inherent in coupled-cluster computations has two main approaches. The first relies on a tasking model to balance the distribution of tasks. The alternative is to use a complex data mapping with static scheduling to achieve load-balancing. Dynamic task scheduling is favorable with varying task granularity, whether the variability arises from uneven floating-point operations per data, uneven cost of moving data, or variably sized data tiles. The performance advantage of dynamic task distribution requires low overhead task distribution and relies on weighing the performance advantage of work balancing against the performance degradation of ignoring locality and the possibility of increasing data movement as detailed in Section 6.

Static scheduling is typically optimal for equal-granularity tasks. The CTF-based backend uses such static scheduling relying on its data mapping strategy. This approach also allows better coordinated transfers and locality aware data access. The data distribution is tied to the work scheduling. Balancing data distribution typically uses some sort of cyclic assignment to avoid overloading a computational node’s resources.

4.5. Locality versus Load-Balancing

Dynamic load-balancing typically requires frequent assignments of work based on the availability of workers. This approach is typically used with a centralized queuing system of available tasks. An alternative approach adopts work stealing between workers, where an initial work distribution is used, then workers who finished their assignment try to move work from overloaded workers. In both cases, the locality is difficult to maintain. Locality is typically leveraged with static work scheduling, where data placement could be taken into consideration while assigning work. In the distributed environment, exploitation of locality motivates the caching of recently used blocks either explicitly by the application or implicitly through the runtime. Both approaches are used in *global*

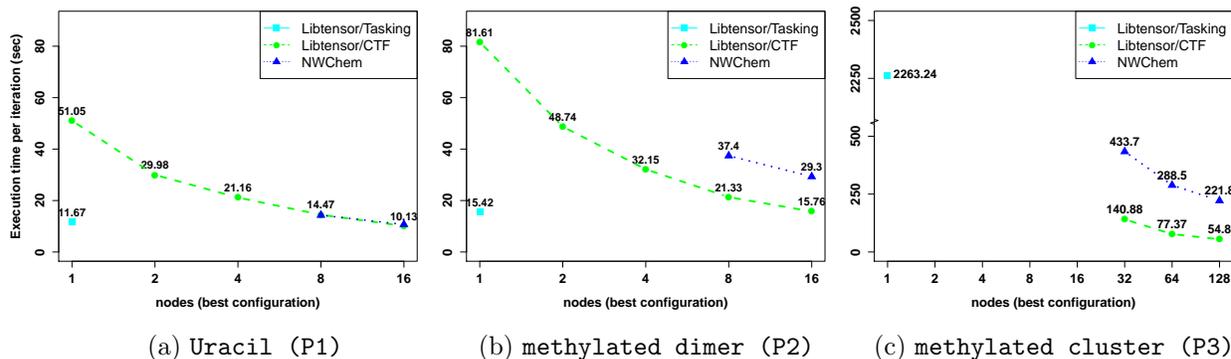


Figure 3: Performance scaling for small problems (P1 & P2) that fit in the physical memory of a single node with LibTensor/Tasking, LibTensor/CTF, and NWChem. Problems P1 & P2 were run on the Cray XC30 (Edison), but Problem P3 requires a large memory compute node (Carver) to run on shared memory.

array-based implementations. Managing locality is typically explicit in CTF framework as the data distribution and work scheduling are static. Whenever the data distribution leads to a complex access pattern, CTF changes the layout using a global exchange operation (data reshuffle) to simplify the access pattern during the next computational phase. On the other hand, complex symmetry properties such as point-group and spin symmetry are not tedious to implement. Finally, as CTF leverages coordinated data transfers, which imply extra synchronization, it also requires additional storage to hold communicated data — common in distributed memory environments.

5. Performance Analysis

In this section, we leverage our distributed memory backend based on CTF for coupled-cluster computations and discuss the performance-limiting factors in different architectures. The first architecture (Mira) is optimized for power-efficiency and presents a unique balance between in-node bandwidth to memory and the inter-node communication bandwidth. The next two architectures (Edison, Cori) rely on powerful compute nodes that are interconnected with the dragonfly topology that allows low latency between compute nodes. Titan is accelerated by a powerful GPU architecture, but is, unfortunately, using an older generation Cray interconnect (3D Gemini). The fat memory node allows solving large problems leveraging the shared memory model abstractions.

The compute kernels typically rely on vendor-optimized DGEMM libraries. This allows assessing the machine balance in a fair fashion, and enables focusing on the analysis of the programming model efficiency in handling locality, load imbalance, data movement, etc.

5.1. Performance at Small Scale

For small problems that fit in the memory of a single compute node, we can directly compare the shared memory programming models (LibTensor/tasking) with the distributed memory codes (LibTensor/CTF, NWChem)² by running one of more processes per socket. In Figure 3(a), we

²For optimal performance, multiple tiling sizes need to be explored at each run scale. Larger tiles are typically better for increasing the efficiency of the computational kernels and reduce the impact of scheduling, while smaller

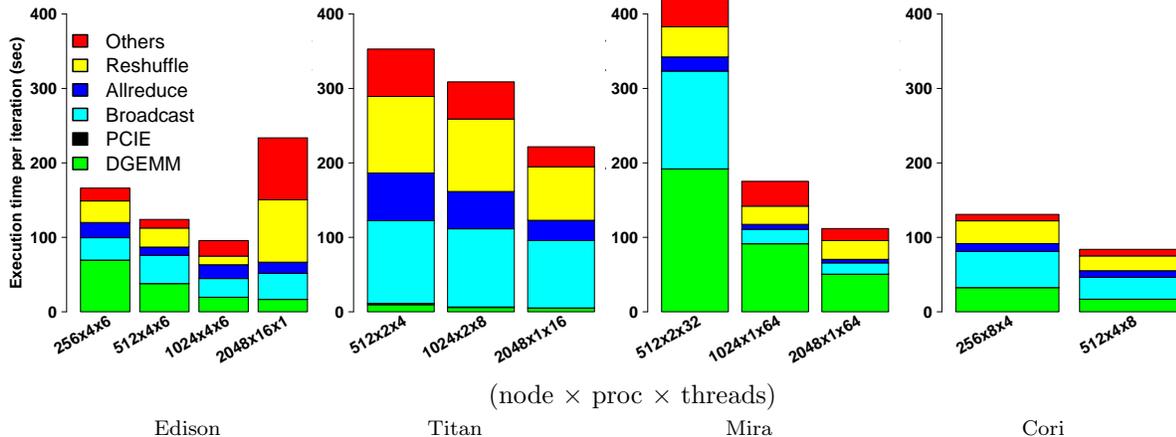


Figure 4: Scaling of the nucleobase tetramer-FN0 (P4) problem on Edison, Cori, Mira, and Titan (w/GPUs). Although performance is dominated by the communication time on all architectures, there are benefits to increased concurrency. Although Cori and Titan have comparable peak performance per node, Cori’s network ensures superior performance. Note, in each figure, each column represents twice the hardware (node) concurrency of the previous column but allows for a different balance of MPI vs. OpenMP on node. We start with the minimum number of nodes that makes the problem fits in memory. Observe the combination of compute and network is critical to obtaining performance and scalability.

show that when the problem fit in memory the task-based backend provides a higher performance ($4.37\times$ for P1 on a single node) compared with the distributed memory backend (CTF) because the distributed memory version (based on CTF) is less effective in preserving the symmetry properties as operator are applied to data. The challenges for supporting these forms of symmetry with CTF is discussed in Section 4. The shared memory task-based backend reduces the memory and the computational requirements, as well as the memory footprint by avoiding the redundant states used to optimize the communication in CTF. Similar trends are observed for the P2 problem, 3(b). The shared memory version outperforms the distributed memory version by $5.29\times$. With CTF backend, Libtensor outperforms NWChem for this problem and the P3 problem by up to $4\times$.

Earlier studies show a similar performance scaling advantage of CTF-based implementations compared with NWChem [11]. Moderately large problem such as P3 can be executed on a node with a large physical memory (such as Carver). The distributed memory implementations benefit from compute capability scaling with the available memory. NWChem extends the task model to distributed memory by relying on the global-arrays abstraction. The performance of NWChem version and the CTF version match each other in the 8-16 node range. However, in our experiments, we observe that NWChem requires more nodes for the coupled-cluster CCSD computation (i.e. CTF is more memory efficient). The importance of being memory efficient is likely to increase in future system due to the trend of decreasing memory capacity per compute core.

NWChem uses the global-array abstraction to extend the use of the task model to the distributed environment. It allows expressing complex symmetry properties in distributed environments and uses dynamic load-balancing for work distribution. We do not intend in this study to carry a full

tiles increases the available task count. Details are omitted for brevity.

comparison between these programming environments because the supported computational kernels by these programming environments are different.

The Libtensor software stack provides the ability to use multiple programming model backends, while providing a single interface for the computational scientists. The efficiency of these models depends on the architectural configurations. The problems that could fit in memory change with each new generation of architectures depending on the size of the physical memory.

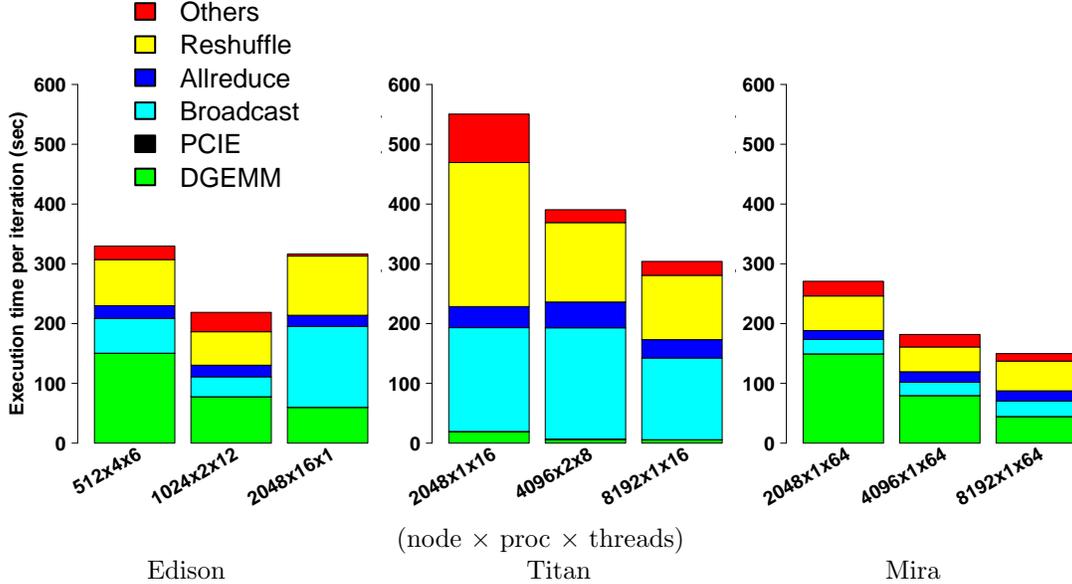


Figure 5: Scaling of the nucleobase tetramer (P5) problem (much larger) on different supercomputing architectures, Edison, Mira, and Titan. Libtensor scales to hundreds of thousands of cores. Note, Cori’s data partition does not have enough nodes to study the scaling for this size problem.

5.2. Performance at Large Scale

In order to avoid out-of-core processing, large problems such as P4 and P5 require hundreds of megabytes to multiple terabytes of memory. In a distributed environment, data replication (and caching) techniques are also used for communication-avoiding algorithms thus the memory requirements for efficient execution typically exceeds the bare-minimum requirements. Figure 4 presents the scaling behavior of the P4 problem³ on different HPC platforms and the breakdown of the execution time. In each system, we show the run configuration associated with the best performance. The results show that performance starts with a large compute component — 42% on the Cray XC30 (Edison). The contribution of communication component grows as expected with the scale and eventually dominates the execution time. We see that P4 problem does not scale

³The minimum number of nodes in each experiment is decided by the aggregate physical memory of these nodes that would make the problem fits in memory.

beyond 1024 nodes (24576 cores) and similar trends are observed on IBM BGQ (Mira) where the contribution of the communication time to the total execution is slightly less because the ratio of the interconnect communication capability to the compute power of the lightweight core is higher compared with other architectures. The improvement in core computational power by the Intel Haswell reduces the DGEMM time on Cori, while the communication has not improved despite the same Aries interconnect on both Edison and Cori.

For the GPU-accelerated architecture, Titan, we observe that computation is a tiny fraction (less than 3%) of the total execution time. The computational capability afforded by the compute node (principally the accelerator) is not matched by the interconnect: Cray’s previous generation torus interconnect (Gemini). Note, time spent in communication on Edison (Cray Aries interconnect) is significantly lower at comparable concurrency. Using the Libtensol stack, we are able to run large coupled-cluster computations that were not possible on fat-memory shared memory machines, such as P5 (no frozen natural orbitals (FNO) optimization). The achieved improvement in time to solution is up to 240× compared with the highly optimized shared memory implementation of Libtensol for the P4 problem on the Carver system⁴. The main performance-dominant factor of Libtensol remains the communication component, when scaling to hundreds of thousands of cores.

As shown in Figure 5, the CTF backend allows us to run efficiently a larger problem, e.g., **nucleobase tetramer (P5)**, which could have hardly run on a large memory node. We observed a better scaling behavior, but we found similar trends in terms of being dominated by the communication component at scale, especially for GPU-accelerated architectures.

5.3. Performance Tuning At Scale

So far we presented the best configuration for performance at scale. This typically involves tuning across multiple parameters. In this section we show that the impact of tuning the balance between processes and threads on each node. As the CTF backend of Libtensol only uses the master thread for communication, the level of injection concurrency is a function of the number processes per node.

Whereas CTF nominally requires a power-of-two number of processes, on Edison, with 12 cores per socket, the use of threads allows for fully exploiting its non-power-of-two core count (24) by the MKL library when executing DGEMM. Otherwise, small improvements are observed with threads compared with the use of processes. In Figure 6, we show that on Mira using one process per node and 64 threads provides the best performance.

Having one process per node simplifies the use of hardware-accelerated collectives. Interestingly, the communication and the computation part affect each other because both the compute and the communication data are brought to the chip because the network controller is integrated on-chip. On Titan, moderate concurrency (2 processes per node sharing the 1 GPU) improves the communication for P4 problem. The compute part is not affected because it is mostly done by GPUs. Increasing the concurrency further reduces the message size and possibly creates more contention in the interconnect. Consequently, we observe an increase in the communication time with more than two processes per node for the P4 problem. This issue exacerbates at scale with the P5 problem where we observe that performance degrades with any increase in process concurrency beyond a single process.

⁴We use shared memory as the base for comparison because in our experience shared memory solution, when available, delivers the best performance.

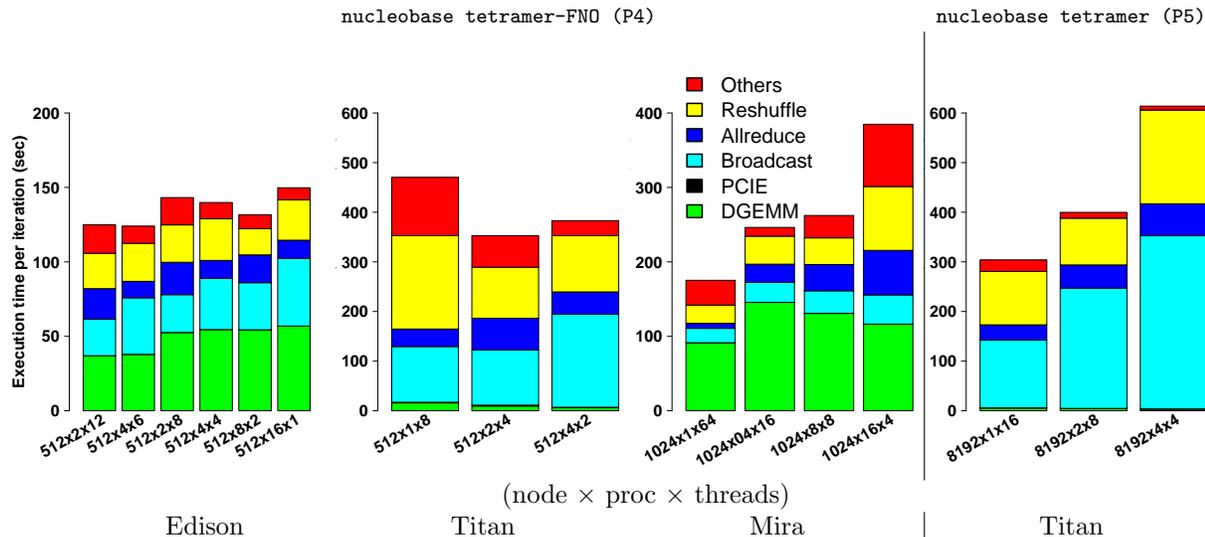


Figure 6: Runtime as a function of the balance between processes and threads for fixed hardware concurrency for the P4 and P5 problems. Observe on-node DGEMM time is relatively constant, but inter-node communication time varies substantially as one shifts on-node parallelism from threads to processes.

These data suggest that no single policy is adequate for all platforms. Even on the same platform, one must tune the available parallelism for maximum performance. Automatically determining these configuration parameters is an interesting problem that we leave as a future work. A user of this software is advised to evaluate multiple test configurations before conducting a long running computation.

These results show that CCSD computations are communication-bound at scale, especially when using accelerator-based technologies such as GPUs. The data movement, its volume and pattern, influences the performance. In turn, data movement is influenced by the algorithm and the programming model (e.g. task-based or orchestrated collectives).

5.4. Energy to solution

Given that the time spent in communication dominates the execution time, we explored techniques to reduce the energy-to-solution. To achieve this objective, we explored the use of dynamic voltage and frequency scaling (DVFS)⁵ [22, 23] feature in Intel Haswell (Cori). Contrary to the conventional wisdom that optimal time to solution yields an optimal energy to solution (race-to-halt), DVFS opens the chance to reduce the energy by increasing the execution time. Cori has a factor of $2.26\times$ in total node power consumption between the minimum and the maximum, which is substantially higher than Edison difference (only 50%). On Cray XC machines, energy could easily be measured through the performance counters exposed by the operating system. We instrumented our code to take such energy measurements of the iterative part of the solution. All measurements

⁵In circuit designs, reducing frequency allows reducing the operating voltage. Given that power is proportional to $V \times f^2$, the energy saving is asymptotically cubical.

in shown in Figure 7 are per iteration.

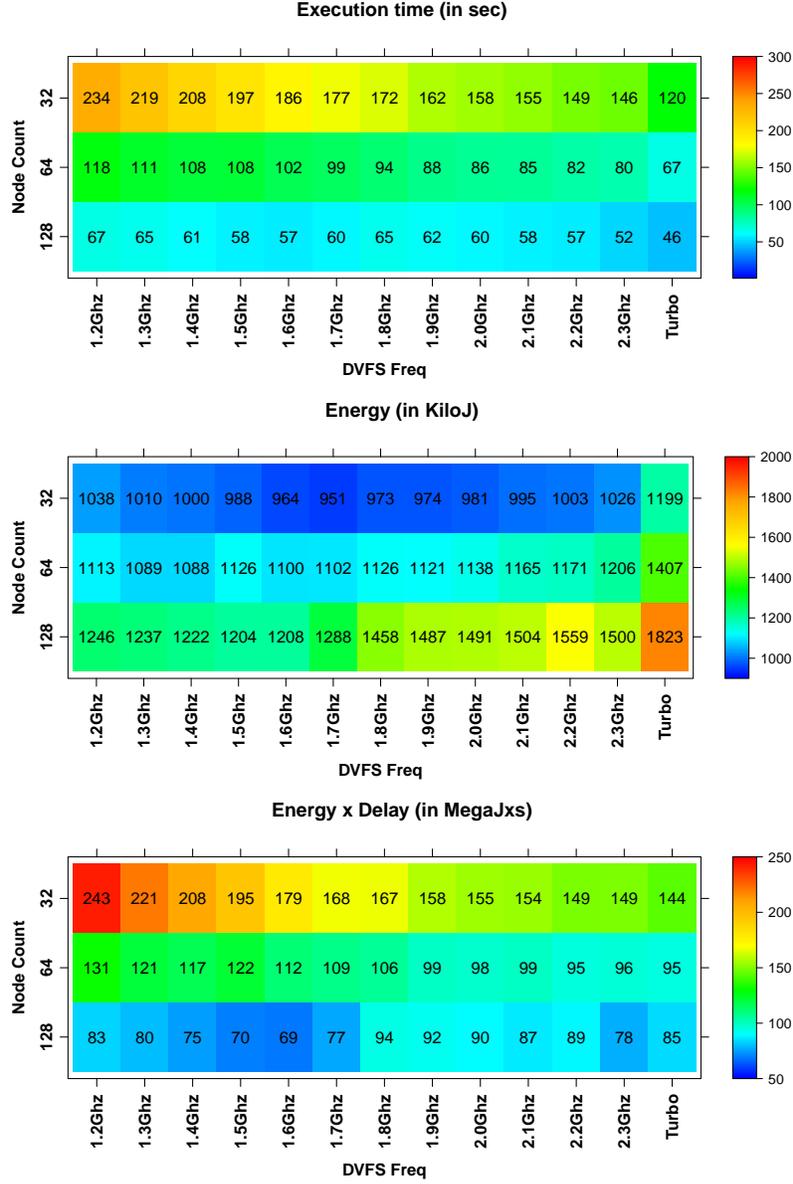


Figure 7: Energy consumption per iteration with different execution times for the `methylated cluster (P3)` problem on Cori. As we strong scale the problem, reducing the frequency becomes essential in reducing energy to solution. The energy-delay product provides a reasonable guide for optimal configuration. Note, in all cases, lower (bluer) is better.

Generally, DVFS reduces the performance of a node without significantly impacting the communication speed. As such, wasting energy while waiting for long communication events does not

help in conserving energy. As shown in Figure 7, up to 29% improvement in energy-to-solution is achievable with the DVFS technique for the P3 problem on 64 nodes. On 128 nodes, the improvement could reach 51%. On the other hand, we observe corresponding slowdowns of 78% and 46% for node concurrency level of 32 and 64, respectively.

We propose the use of the energy \times delay metric [24]⁶ used in electronic circuit design to guide the heuristics for choosing optimal frequency. Observing energy or delay (time) alone makes it harder to choose an optimal DVFS operating point because each is monotonically changing. Energy is increasing with the DVFS frequency, while time is decreasing. The energy \times delay behavior is typically convex, providing some balance between the two conflicting trends. Using it makes the choice of a good operating point, at least from the system perspective, more straightforward. As shown in Figure 7, the energy \times delay suggests that decreasing the frequency as we scale maintains a reasonable energy use by the compute nodes. For instance, we would choose for P3 the frequencies of Turbo, 2.2 Ghz, 1.6 Ghz for node concurrency levels of 32, 64, 128, respectively.

The only impediment in enabling such energy optimizations in our software framework is that compute facilities typically charge users per cpu-hour (instead of KWh). As such the most performant solution is the most favorable from the user perspective.

6. Data Movement and Programming Model

In this section, using proxy benchmarks, we contrast the data movement of tasking-based models with bulk synchronous models to show the importance of switching the programming model as we scale computation to the point where application execution becomes communication-bound. Tasking models make the computation (or balancing it) a first priority, which is justified when data movement is not the main bottleneck (e.g., DGEMM-dominated execution time). These models make it harder to coordinate data transfers. At large scale, the performance underpinnings of NWChem’s tasking model compared with LibTensor/CTF collective model are grounded in the inherit communication requirements for each. However, the lack of orchestration can lead to superfluous data movement when executing task-based models in distributed memory.

6.1. Data Movement Analysis through a Proxy Benchmark

Understanding every facet of data movement for distributed tensor contractions (NWChem/TCE or Libtensor/CTF) can be overwhelming. To that end, we constructed an intuitive experiment involving a distributed matrix-matrix multiplication using three different approaches and measure the actual inter-node data movement in order to proxy the data movement properties of the different approaches to parallelization. The first is the style of matrix-matrix multiplication used in CTF, the second is the corresponding operation in global arrays with static scheduling, while the third uses dynamic workload balancing in global arrays (similar to NWChem). The choice of this simple experiment is to isolate the impact of programming model on data movement, without undue additional complexity in handling symmetry properties.

Prior analysis [9] shows that a tensor contraction could be defined as a generalized SUMMA and the communication volume is asymptotically the same between CTF and NWChem, while we show that data movement is influenced by the runtime and programming model. SUMMA is also shown to be communication optimal assuming no extra memory is used to apply communication-avoiding

⁶The delay in energy \times delay metric refers to the execution time.

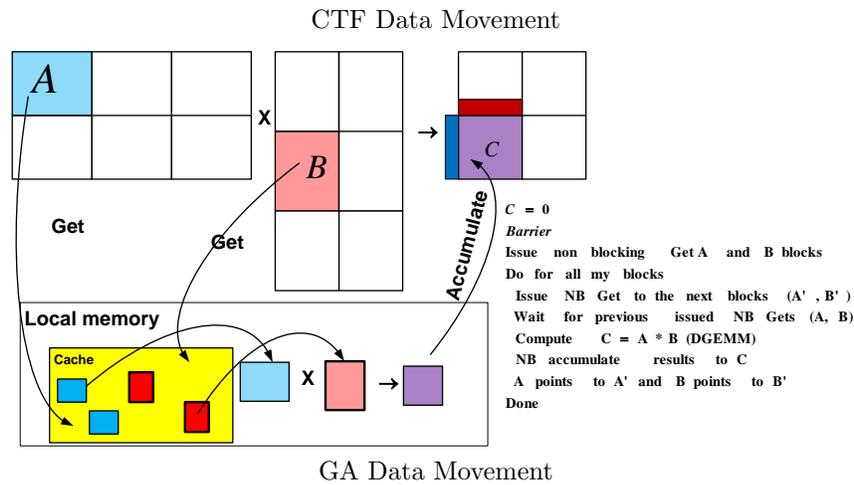
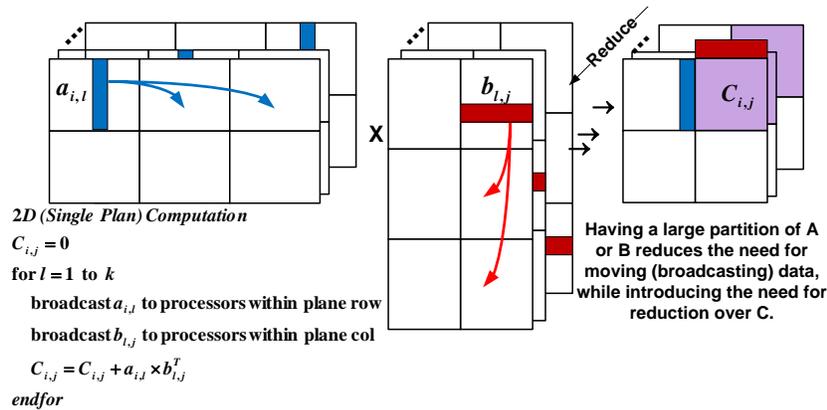


Figure 8: Matrix-Matrix multiply using adaptations of SUMMA. The bulk synchronous version, adopted in CTF, relies on MPI collectives to carry out the communication. Based on memory availability, replicated state is used to reduce communication. The point-to-point version used in global array uses one-sided get and accumulate primitives to carry out the communication. The communication volume per process should ideally be the same for identical problem configurations and memory usage.

optimizations. Both CTF [9] and NWChem [12] use adaptations of the SUMMA. They apply communication-avoiding techniques: CTF algorithmically applies partial data replication (2.5D algorithm) and global-array implementations could apply runtime block caching. Considering the data needed to carry out the computation, Solomonik analysis [9] shows that the communication lower bounds are asymptotically the same for both implementations, given an appropriate choice of a tiling size, which influences the amount of data padding and memory used.

6.2. Analytical Estimate of the Data Movement of the Proxy Benchmark

First, let us consider the fundamental bounds on inter-node data movement. The constrained-memory lower-bound of communication is estimated by $\Omega(n^2/\sqrt{p})$ [25], where n is the matrix dimension and p is the process count. For unbounded memory the lower bound is reduced to

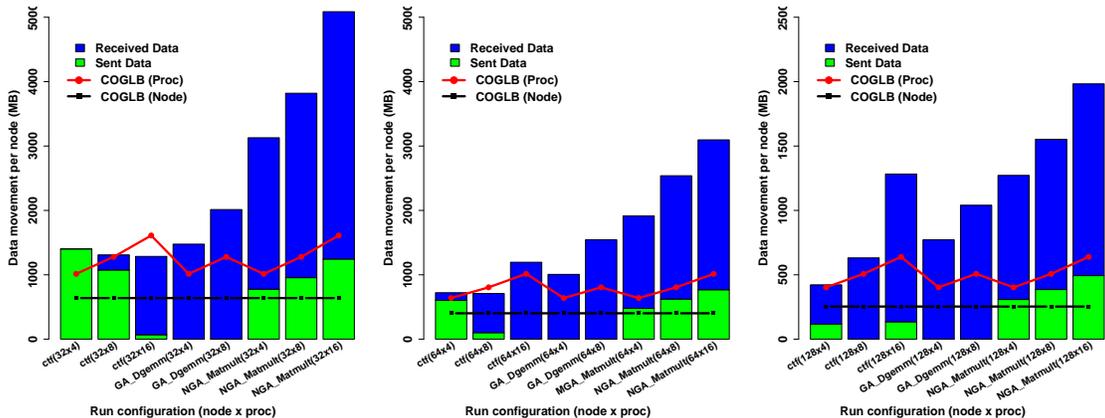


Figure 9: Data movement measure by interconnect performance counters for distributed Matrix-Matrix multiplication (16K square matrices) using different algorithmic variants (CTF uses 2.5D communication avoiding implementation, and the global array (GA) implementations uses SRUMMA variants). Data movement in CTF-based benchmark is significantly lower than the one-sided variants. Interconnect performance counters capture only off-node traffic and include all overheads by the communication protocol. We contrast the empirical measurements with the Communication-Optimal Greatest Lower-Bound (COGLB) metric. CTF follows the per-node COGLB metric when the communication is bandwidth limited.

$\Omega(n^2/p^{2/3})$. For the communication-optimal matrix-matrix multiplication case, we estimate the greatest lower-bound of the data movement between nodes as $3 \times n^2/p^{2/3}$, and call it the Communication Optimal Greatest Lower Bound metric (COGLB metric)⁷. In a memory constrained case (2D), each processor needs to hold n^2/\sqrt{p} portion of the data. In the communication optimal case, each processor should have enough memory to hold a larger partition of the input matrices (A and B) estimated by $n^2/p^{2/3}$ words [26]. Having such portions of data could involve at most $n^2/p^{2/3}$ words of data movement per input matrix. The reduction of the results also requires each process to send/receive $n^2/p^{2/3}$ words of the result matrix C. Based on these assumptions, the lower-bound should be at most $3 \times n^2/p^{2/3}$. Starting with a portion of the input matrices that overlap with workload assignment could reduce the volume of communication of the input matrices. In our experiment, the result matrix initially has some zero elements. Having nonzero elements could create a need for an additional data movement. Receiving (or sending) data is typically matched by sending (or receiving) data, but hardware accelerated *remote direct memory access* (RDMA) communication mechanisms eliminate the need to double-count the same transfer at both ends of the communication.

A compute node can be thought of as a single source of data movement. Processes within a node ideally communicate through the shared memory. For purposes of this validation experiment, we will use the Cray XC30 (Edison) as its Aries network chip exports interconnect performance counters. The measurements of the network traffic is based on reading the interconnect performance counters (Cray Aries used in XC30 series) during the execution of the computation kernel. The measured

⁷COGLB is provided as a rough metric to compare the trends with increasing parallelism and contrast different implementations. It involve some simplifying assumption regarding the initial data layout, the process count, the available physical memory, etc. A tighter bound could be estimated if we incorporate some algorithmic parameters. This analysis is beyond the scope of this work.

Table 4: Matrix Multiplication Kernels.

Kernel	CTF DGEMM	GA DGEMM	GA Matmul_patch
Scheduling	static	static	dynamic
Compute by	owner	owner	any
Communication primitives	<i>broadcast, reduce</i>	<i>get, put</i>	<i>get, accumulate</i>

data movement as reported by these counters is for a single node because they are symmetric given the regularity of the problem and the even distribution. The interconnect counters capture these traffics at the initiator side, while at the remote side they appear as direct access to the memory. This property helps avoiding double-counting the traffic in the interconnect. A sent data are either a “put” operation or a request for a get operation. A received data is a response to a “get” request. At the application level, whether a broadcast operation or point-to-point operations are used, the runtime translates these requests into send or receive transfers. Note, the volume of communication could be affected by how processes are mapped to nodes, which cannot be controlled on Edison.

Given that multiple processes share the same node, some fraction of the inter-process communication happens internally within the node and does incur interconnect traffic. For our experiments, this intra-node communication (via shared memory) does not exceed 25% of the total communication volume and is not captured by the interconnect performance counters.

A basic distributed Matrix-Matrix multiplication algorithm is depicted in Figure 8. The communication part is coordinated using an MPI broadcast across a row and a column communication. CTF uses an optimized variant (2.5D) that relies on replicated state to avoid excessive communication. Depending on the available memory, CTF uses c replicas of each block and upon completion the results are reduced. The global arrays (GA) implementation uses a variant (called SRUMMA [27]) based on non-blocking *get* operations to fetch blocks of data, then perform computation locally before sending them back to the destination. It overlaps communication with computation using a form of double buffering. GA also provides interfaces for replicated (mirrored) state to reduce the volume of communication at runtime. Algorithms leverage caching of recently fetched blocks to reduce communication. The caching of blocks happens on a per process basis and not on a per node basis. As such blocks fetched by another process sharing the same node are typically fetched from the producer of the block when needed.

NGA_matmul_patch⁸ is a routine to compute patches of matrix multiplication. It is typically used with sparse matrices, or with higher dimensional tensors embedded into matrices. This routine mimics part of the algorithmic pattern used in the tensor contractions in NWChem. The routine uses a get-compute-accumulate model, which is typically used for dynamic load-balancing in NWChem. In contrast, the Global Arrays DGEMM (GA_DGEMM) uses an owner compute model because it leverage locality and regularity in the data structure. As such the results of the computation are always accumulated to the local portion of the array; “sent data” component to remote nodes does not exist. Obviously this is efficient only for evenly distributed data that has balanced computation associated with them. Table 4 summaries the key attributes for each of the matrix multiplication variants.

The CTF benchmark relies on the MPI runtime for creating coordinated transfers (through

⁸The NGA prefix is used for the n-dimensional global array routines.

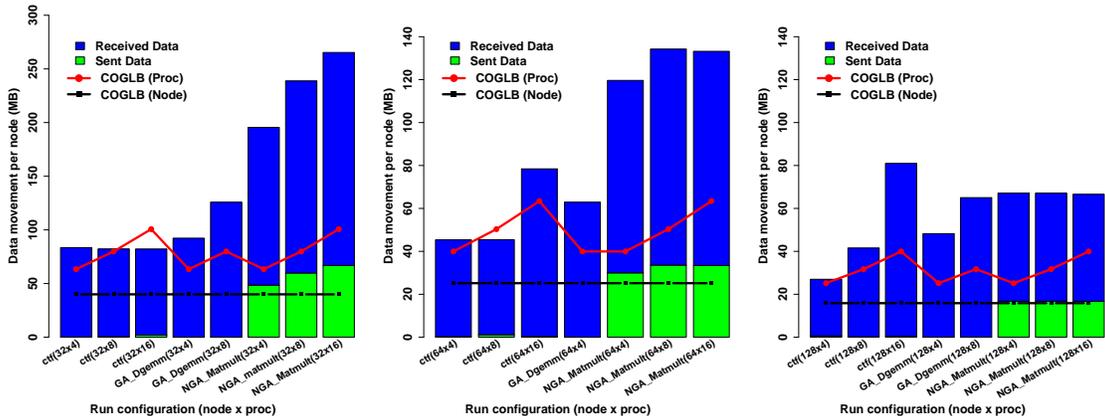


Figure 10: Data movement per node for small Matrix-Matrix (4K square matrices) multiplication for different programming models. We contrast the empirical measurements with the Communication-Optimal Greatest Lower-Bound (COGLP) metric. With strong scaling (at right figure), the data movement of the one-sided implementation could approach or become better than the CTF version due to effective caching of blocks for the one-sided version and the possible use of point-to-point protocol in the MPI broadcast. The influence of the MPI runtime on data movement is generally greater for the CTF-based benchmark.

broadcast) from the producers to all consumers along the row and column communicator. The “owner computes” rule is used, but if data is replicated, then MPI reduction is used to have optimized accumulation of the results. The CTF authors introduced a communication-avoiding technique (2.5D [28, 9]) to reduce the communication by redundantly copying the blocks of data to different nodes depending on the availability of memory.

6.3. Empirical Measurements and Analysis of the Data Movement of the Proxy Benchmark

Figure 9 shows the volume of communicated data per node for the various implementations for different processes per node. The first observation is that data movement using CTF-based benchmark is typically lower than that associated with GA_DGEMM or NGA_Matmul_patch. NGA_matmul_patch, which trades optimizing locality of access for attempting to load-balance computation. Specifically, the amount of data movement per node could be $3.96\times$ the optimized MPI version for the 32 nodes \times 16 processes configuration. Increasing the process concurrency per node, we see that coordinated transfers could keep communication volume constant, especially for large matrices ($n=16K$) at low concurrency (32 nodes). MPI protocol could switch *get*-based protocol with *put*-based protocol without impacting transfer volume. As shown in Figure 10, for small matrices ($n=4K$) at high concurrency (128 nodes \times 16 processes), we start seeing a performance advantage of dynamic load-balancing. At this point the problem stress the latency of the interconnect rather than the bandwidth.

In our experiments for CCSD computations, Section 5, the memory constraints influences the lowest number of nodes that could be used to run the problem. Therefore, Figure 9-32-node is the most dominant use scenario as we use multiple matrices with different sizes, multiple use scenarios could be encountered within each run.

We observe that the CTF benchmark follows the node-base COGLB metric (constant per per node) on 32 nodes. Following the per-node metric implies optimized data movement because the

data needed by multiple ranks within the node is brought to the node once then distributed internally. Following the per-rank metric indicates that fetching the data required by multiple ranks is not coordinated. As such, different ranks within a node may bring the same data multiple times. As we strong scale, we observe a change in behavior to following proc-based COGLB metric. We attribute this change to runtime change in protocol for broadcast and reduction operations. The runtime could optimize to reduce latency by issuing more point-to-point messages, thus increasing the volume of traffic. For global array-based implementations for the small matrix ($n = 4K$) at 128 node runs, we observed the node-based COGLB being followed due to effective caching of blocks. Otherwise, the per-process (proc-based) COGLB correlates better with the data movement for almost all other runs. This indicates that no-optimizations are used at the node level for the global array based implementation. For CTF, MPI typically uses a hybrid protocol for reductions and broadcasts. For instance, a broadcast of a block is communicated once through the interconnect to the node, then shared memory broadcast is used to distributed data locally within the node. We could easily observe that the data movements measurements follow more closely the per process estimate rather than the per node estimate for most runs. This illustrates more uncoordinated communication behavior at the node level. As we scale problems, we also notice the per-node bound approaching the per-process bounds because a larger fraction of the communication happens with external nodes.

One could easily conclude from these experiments that the programming model and its runtime have a great influence on the data movement and the efficiency of the implementation. This behavior is not typically captured through analytical models. We could see that the same algorithm (i.e., CTF) at small scale with large matrices has totally different data-movement trends compared with at large scale with small matrices. Although, communication-avoiding algorithms benefit from higher memory availability (small problems with large process counts) to reduce data movement, the runtime may switch broadcast protocol to reduce latency thus increasing data movement.

The need for coordinated transfers is not a concern in shared memory architectures because memory could be thought of as a network of constant diameter (one). Additionally, cache reuse between different cores still proves to be a hard problem, especially if the synchronization between cores is high compared with the compute time.

Overall, we find the benefit of leveraging the full symmetry properties and the use of load-balancing technique in shared memory to outweigh the benefit of coordinated transfers. Thus, the tasking model backend outperforms the distributed memory backend (CTF) when a problem fits in shared memory.

7. Conclusions

Scalability in extreme-scale computing is typically determined by the efficiency of the communication component of execution. We extended the Libtensor framework to work in a distributed memory environment, achieving up to $240\times$ speedup over the optimized shared memory implementation of Libtensor. Although we leveraged the best known approaches to optimize for data movement, we show that coupled-cluster computations could spend more than 90% of their execution time in communication, especially on architectures accelerated by GPUs. In this study, we show that the programming model has a large influence on the volume of data movement, in addition to the algorithm used. Using a proxy benchmark, we showed that extending task-based models, in order to tackle load-balancing, to the distributed memory environment could lead to higher data movement compared with bulk-synchronous models with coordinated transfers. However, tasking

models delivered the best performance at small scale, *i.e.*, within shared memory. As such, our extension of Libtensor uses multiple programming model backends, while unifying the interface for domain scientists in order to provide consistently good performance across scales ranging from a single shared memory node to supercomputers with hundreds of thousands of cores. We show the opportunity for optimizing for energy consumption for coupled cluster computation using DVFS.

Libtensor now allows computational scientists to use a single framework for coupled-cluster computations at different scales with highly efficient execution.

Acknowledgments

This research used resources in Lawrence Berkeley National Laboratory and the National Energy Research Scientific Computing Center, which are supported by the U.S. Department of Energy Office of Science’s Advanced Scientific Computing Research program under contract number DE-AC02-05CH11231. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program.

We would like to thank the anonymous reviewers for providing suggestions to get a better performance of NWChem runs and improve the presentation in this manuscript.

References

- [1] E. Epifanovsky, M. Wormit, T. Kuś, A. Landau, D. Zuev, K. Khistyayev, P. Manohar, I. Kaliman, A. Dreuw, A. Krylov, New implementation of high-level correlated methods using a general block-tensor library for high-performance electronic structure calculations, *J. Comput. Chem.* 34 (2013) 2293–2309.
- [2] E. Epifanovsky, M. Wormit, A. Dreuw, , A. Krylov, Tensor algebra library for computational chemistry, <http://iopenshell.usc.edu/downloads/tensor/>.
- [3] R. A. Van de Geijn, J. Watts, Summa: Scalable universal matrix multiplication algorithm, Tech. rep., University of Texas at Austin, Austin, TX, USA (1995).
- [4] S. Hirata, Tensor contraction engine: abstraction and automated parallel implementation of configuration-interaction, coupled-cluster, and many-body perturbation theories, *J. Phys. Chem. A* 107 (46) (2003) 9887–9897.
- [5] V. Lotrich, N. Flocke, M. Ponton, B. A. Sanders, E. Deumens, R. J. Bartlett, A. Perera, An infrastructure for scalable and portable parallel programs for computational chemistry., International Conference of Supercomputing (ICS) (2009) 523–524.
- [6] J. Stanton, J. Gauss, M. Harding, P. Szalay, A. Auer, R. Bartlett, U. Benedikt, C. Berger, D. Bernholdt, Y. Bomble, O. Christiansen, M. Heckert, O. Heun, C. Huber, T.-C. Jagau, D. Jonsson, J. Juselius, K. Klein, W. Lauderdale, D. Matthews, T. Metzroth, D. O’Neill, D. Price, E. Prochnow, K. Ruud, F. Schiffmann, S. Stopkowicz, M. Varner, J. Vazquez, F. Wang, J. Watts, in CFOUR, Coupled Cluster techniques for Computational Chemistry, a quantum-chemical program package (www.cfour.de).

- [7] E. Solomonik, D. Matthews, J. Hammond, J. Demmel, Cyclops Tensor Framework: Reducing communication and eliminating load imbalance in massively parallel contractions, *The IEEE 27th International Symposium on Parallel Distributed Processing (IPDPS)* (2013) 813–824 doi: 10.1109/IPDPS.2013.112.
- [8] J. Calvin, TiledArray, a massively-parallel, block-sparse tensor library written in C++, <https://github.com/ValeevGroup/tiledarray>, (Accessed on May 30, 2014).
- [9] E. Solomonik, D. Matthews, J. R. Hammond, J. F. Stanton, J. Demmel, A massively parallel tensor contraction framework for coupled-cluster computations, *Journal of Parallel and Distributed Computing* 74 (12) (2014) 3176 – 3190, domain-Specific Languages and High-Level Frameworks for High-Performance Computing. doi:<http://dx.doi.org/10.1016/j.jpdc.2014.06.002>.
URL <http://www.sciencedirect.com/science/article/pii/S074373151400104X>
- [10] D. Matthews, Aquarius (a parallel quantum chemistry package), <https://github.com/devinamatthews/aquarius>., (Accessed on April 2, 2016).
- [11] M. Valiev, E. J. Bylaska, N. Govind, K. Kowalski, T. P. Straatsma, H. J. J. V. Dam, D. Wang, J. Nieplocha, E. Apr, T. L. Windus, W. A. deJong, Nwchem: A comprehensive and scalable open-source solution for large scale molecular simulations, *Computer Physics Communications* 181 (9) (2010) 1477–1489.
- [12] P. Ghosh, J. R. Hammond, S. Ghosh, B. Chapman, Performance Analysis of the NWChem TCE for Different Communication Patterns, *Lecture Notes in Computer Science. High Performance Computing Systems: Performance Modeling, Benchmarking and Simulation*. 8551 (2014) 281–294.
- [13] X. Liu, A. Patel, E. Chow, A New Scalable Parallel Algorithm for Fock Matrix Construction, *The IEEE Parallel and Distributed Processing Symposium*.
- [14] L. V. Kale, S. Krishnan, CHARM++: a portable concurrent object oriented system based on C++, Vol. 28, ACM, 1993.
- [15] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, D. Fey, Hpx: A task based programming model in a global address space, in: *The 8th International Conference on Partitioned Global Address Space Programming Models, PGAS '14*, ACM, New York, NY, USA, 2014, pp. 6:1–6:11. doi:10.1145/2676870.2676883.
URL <http://doi.acm.org/10.1145/2676870.2676883>
- [16] C. Augonnet, S. Thibault, R. Namyst, P.-A. Wacrenier, StarPU: A unified platform for task scheduling on heterogeneous multicore architectures, in: *European Conference on Parallel Processing*, Springer, 2009, pp. 863–874.
- [17] P. Hudak, Building domain-specific embedded languages, *ACM Computing Surveys (CSUR)* 28 (4es) (1996) 196.
- [18] D. Abrahams, A. Gurtovoy, C++ template metaprogramming: concepts, tools, and techniques from Boost and beyond, Addison-Wesley Professional, 2004.

- [19] Y. Shao, Z. Gan, E. Epifanovsky, A. Gilbert, M. Wormit, J. Kussmann, A. Lange, A. Behn, J. Deng, e. a. Feng, X., Advances in molecular quantum chemistry contained in the Q-Chem 4 program package, *Molecular Physics* 113 (2015) 184–215.
- [20] K. Ibrahim, S. Williams, E. Epifanovsky, A. Krylov, Analysis and tuning of libtensor framework on multicore architectures, *The International Conference on High Performance Computing (HiPC)* (2014) 1–10.
- [21] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, E. Aprà, Advances, applications and performance of the global arrays shared memory programming toolkit, *Int. J. High Perform. Comput. Appl.* 20 (2) (2006) 203–231. doi:10.1177/1094342006064503. URL <http://dx.doi.org/10.1177/1094342006064503>
- [22] G. L. Valentini, W. Lassonde, S. U. Khan, N. Min-Allah, S. A. Madani, J. Li, L. Zhang, L. Wang, N. Ghani, J. Kolodziej, et al., An overview of energy efficiency techniques in cluster computing systems, *Cluster Computing* 16 (1) (2013) 3–15.
- [23] D. Hackenberg, R. Schne, T. Ilsche, D. Molka, J. Schuchart, R. Geyer, An energy efficiency feature survey of the intel haswell processor, *The IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSW)* (2015) 896–904.
- [24] M. Horowitz, T. Indermaur, R. Gonzalez, Low-power digital design, *IEEE Symposium Low Power Electronics, Digest of Technical Papers.* (1994) 8–11.
- [25] D. Irony, S. Toledo, A. Tiskin, Communication lower bounds for distributed-memory matrix multiplication, *J. Parallel Distrib. Comput.* 64 (9) (2004) 1017–1026.
- [26] J. Demmel, D. Elichu, A. Fox, S. Kamil, B. Lipshitz, O. Schwartz, O. Spillinger, Communication-optimal parallel recursive rectangular matrix multiplication, *The IEEE International Symposium on Parallel Distributed Processing (IPDPS)* (2013) 261–272.
- [27] M. Krishnan, J. Nieplocha, Srumma: a matrix multiplication algorithm suitable for clusters and scalable shared memory systems, *The International Parallel and Distributed Processing Symposium* (2004) 70–.
- [28] E. Solomonik, J. Demmel, Communication-Optimal Parallel 2.5D Matrix Multiplication and LU Factorization Algorithms, *Euro-Par 2011 Parallel Processing* 6853 (2011) 90–109. doi:10.1007/978-3-642-23397-5_10.