Evaluating Data Redistribution in PaRSEC

Qinglei Cao, George Bosilca, Nuria Losada, Wei Wu,
Dong Zhong, and Jack Dongarra, *Fellow, IEEE*

Abstract—Data redistribution aims to reshuffle data to optimize some objective for an algorithm. The objective can be multi-dimensional, such as improving computational load balance or decreasing communication volume or cost, with the ultimate goal of increasing the efficiency and therefore reducing the time-to-solution for the algorithm. The classic redistribution problem focuses on optimally scheduling communications when reshuffling data between two regular, usually block-cyclic, data distributions. Besides distribution, data size is also a performance-critical parameter because it affects the reshuffling algorithm in terms of cache, communication efficiency, and potential parallelism. In addition, task-based runtime systems have gained popularity recently as a potential candidate to address the programming complexity on the way to exascale. In this scenario, it becomes paramount to develop a flexible redistribution algorithm for task-based runtime systems, which could support all types of regular and irregular data distributions and take data size into account. In this article, we detail a flexible redistribution algorithm and implement an efficient approach in a task-based runtime system, PaRSEC. Performance results show great capability compared to the theoretical *bound* and ScaLAPACK, and applications highlight an increased efficiency with little overhead in terms of data distribution, data size, and data format.

Index Terms—Data redistribution, data distribution, data size, data format, task-based programming model, dynamic runtime system, high-performance computing

1 Introduction

Massive parallelism is the dominant force behind the increased capabilities of high-performance computing (HPC) because of shifting trends towards increasingly hybrid machines and fat nodes with deep memory hierarchies augmented with various types of accelerators (GPUs, APUs, etc.). To satisfy the increasing demands of applications and achieve new levels of efficiency and performance, HPC architectures are delivering unprecedented increases in concurrency, non-uniform hardware designs, and changing performance capabilities. In this unfriendly scenario, application developers are required to expose enough parallelism from algorithms to highly utilize heterogeneous hardware resources, to decompose and express their computations in a way that is portable among shared- and distributed- memory machines with widely varying configurations. As a result, they face unfamiliar challenges at all levels, from the increasing number of nodes to the highly sophisticated architectural capabilities of each node. They also face a lack of portability between different architectures and a lack of compatibility across different versions of the same hardware. The 'MPI+X model' is one of the most popular programming paradigms for parallel applications to relieve the burdens of expressing parallelism, managing hardware resources, and addressing communications. This model explicitly exposes the complexity of handling the non-uniform platforms to developers, and encourages static assumptions about synchrony, deterministic scheduling, predictable runtime of computation and communication, and distribution of the computation between different logical domains. As the systems grow increasingly complex in core and node count and in the heterogeneity of computational resources and applications' sizes, these burdens become increasingly costly, and the static assumptions no longer hold; even a minor amount of system noise and small delays could introduce significant slack in large-scale synchronous applications [1], [2], [3].

Therefore, to support developers' productivity and perform at extreme scales, it is becoming clear that changes in the programming model paradigm are required to tackle these challenges and facilitate the development of parallel HPC applications. The task-based programming model has become popular and has proven to be efficient and productive in this regard. A task-based programming runtime could help users to efficiently manipulate the complicated low-level heterogeneous resources so that they get opportunities to focus more on their domain knowledge instead of computer science. In this context, users need to describe algorithms to the runtime by expressing computations and the corresponding data where computations perform by means of tasks and dependencies. Computations become entities (a.k.a., tasks, a set of instructions that access and modify an explicit and bounded amount of data), and the

- *Qinglei Cao, George Bosilca, Nuria Losada, Dong Zhong, and Jack Dongarra are with the Department of Electrical Engineering and Computer Science, University of Tennessee, Knoxiville, TN 37996 USA. E-mail: {qcao3, dzhong}@vols.utk.edu, {bosilca, nlosada, dongarra}@icl.utk.edu.*
- *Wei Wu is with Los Alamos National Laboratory, New Mexico 87545 USA. E-mail: wwu@lanl.gov.*

Manuscript received 25 Mar. 2021; revised 15 Sept. 2021; accepted 22 Nov. 2021. Date of publication 30 Nov. 2021; date of current version 10 Dec. 2021.
This work was supported in part by the Exascale Computing Project under Grant 17-SC-20-SC, a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. For computer time, this research used the Shaheen II supercomputer hosted at the Supercomputing Laboratory at KAUST.
(Corresponding author: Qinglei Cao.)
Recommended for acceptance by R. M. Badia.
Digital Object Identifier no. 10.1109/TPDS.2021.3131657

data flowing among them are the dependencies. Therefore, algorithms can be represented as a directed acyclic graph (DAG) with vertices as tasks and edges as dependencies. Hence, a vast amount of potential parallelism is exposed by a set of successive and fine-grained tasks, and the runtime system is then responsible for scheduling these tasks, while satisfying the data dependencies between them guided by the DAG, while dynamically mapping parallelism onto the underlying hardware resources. Task-based programming models associated with dynamic runtime systems have been thoroughly studied, and have been proven extremely efficient in intelligently using all the resources' computational power on heterogeneous platforms for many scientific computing fields—including application libraries built on top of the usual dense [4], [5], [6], [7], [8] and sparse [9], [10], [11], [12], [13] linear algebra solvers with regular, arithmetic/memory-intense computational tasks.

On the other hand, in many scientific applications, data needs to be frequently moved from one distribution scheme into another at runtime, in order to provide better data locality, load balance, as well as performance. For instance, in adaptive mesh refinement (AMR), in order to dynamically adapt the accuracy of a solution within certain sensitive or turbulent regions of simulation, these regions need to be refined; hence, redistribution is always applied to these regions with the explicit goal of a better load balance. Such data layout/distribution changes are called “data redistribution”. Actually, the question of data redistribution has been proposed for more than two decades, both statically and dynamically, as the issue was central to dealing with the imposed data distributions of early distributed-memory programming models such as High Performance Fortran (HPF) [14]. Array redistribution, popular in HPF and used to change the distribution of an array dynamically from a specified source distribution to a specified target distribution is also one of the most expensive communication patterns and is particularly important for applications where the parallelism alternates between dimensions of the data. As a result, numerous scientific literature on array redistribution exists [15], [16], [17], [18], [19], [20]. More general data redistribution focuses on redistribution between two data sets (e.g., from how it was generated by the producer to how the application needs the data to be laid out among its processes [21]) or relocating data distributed across one producer grid onto a different distribution scheme across a consumer grid [22].

Research on redistribution involves not only HPF but also the Message Passing Interface (MPI) [19], [23], towards both coarse-grained [24] and fine-grained [25], [26] and applies to many scientific domains—including linear algebra, like ScaLAPACK [27], [28], and particle codes [25], [26], [29]. However, there are two main limitations.

- These approaches usually focus on regular data distributions: the static Block Cyclic Data Distribution (BCDD; for one-dimensional, it is 1DBCDD; two-dimensional, 2DBCDD) descriptor on which the dense linear algebra community has been relying for more than two decades. Irregular data distribution (distributions other than BCDD) is also important from a load balancing perspective in terms of memory, computation and communication, as suggested by the hybrid data distribution (called ”band distribution”) utilized in [13] used for tiled low-rank (TLR) Cholesky.
- Distribution is the ultimate goal for these studies (even if the derived data size changes as a side-effect like in ScaLAPACK [30]); in fact, besides distribution, finding the right data size (a.k.a. tile size in tile-based algorithms like PLASMA [31] and DPLASMA [5])—the one that trades-off performance and level of concurrency—is also a critical step [32]. Smaller tiles further decrease the computational intensity of the mathematical kernels, while increasing the memory burden and the management overhead imposed on the supporting programming model and execution environment. Oppositely, while providing more computationally intensive operations, larger tiles decrease the degree of parallelism available, limiting the number of tasks that can run in parallel and therefore reducing the occupancy. In many cases, e.g. [33], the so-called data tiling size is critical and dependent on the problem size, and it has been elusive to determine *a single best data size* used for the whole linear algebra system including multiple stages.

Due to the increasing complexity of hardware architectures and communication topologies, many of the regular data distributions might be unfitting for modern problems, both in terms of the efficiency and the scalability of the resulting algorithms. Moreover, as the popularity of task-based runtimes increases, it is interesting to revisit the data redistribution problem in this context and imagine support for more flexible, irregular, data redistributions in a task-based runtime system. In this paper, we focus on PaRSEC [34] an event-driven task-based runtime system depending on dataflow and propose an efficient approach in PaRSEC for the flexible data redistribution algorithm [35]. We believe our redistribution algorithm is generic and could be applicable for other event-driven runtime systems; however, this is out of this paper's scope.

The challenges we are addressing in this paper are twofold, at the algorithmic level and at the programming paradigm level: (1) a flexible redistribution algorithm (2) its implementation in a task-based runtime along with runtime supports and optimizations to reduce overheads and maximize utilization of network bandwidth[1]. The following innovations represent the core of our efforts and this paper's contributions:

- designing a flexible redistribution algorithm for a general redistribution problem supporting coarse- and fine- grained [26] without constraints of data distribution and data size,
- deploying the redistribution algorithm in the PaRSEC task-based runtime system along with runtime-level support and optimizations,
- building a cost model of this redistribution implementation in PaRSEC on overhead analysis and *bound* declaration,

[1] source codes of redistribution are available in PaRSEC: https://bitbucket.org/icldistcomp/parsec with Git hash version ccf60d9

- analyzing the performance of the implementation, comparing against the theoretical bound and ScaLAPACK, and highlighting effectiveness and minimal overheads in real applications.

To the best of our knowledge, this is the first time a flexible redistribution algorithm targeting a general redistribution problem has been proposed in the task-based runtimes world, and the first time in redistribution to support irregular distributions and explicitly take data size into account.

The remainder of this paper is as follows. Section 2 presents related work, Section 3 introduces the design of the redistribution algorithm, and Section 4 provides a basic description of PaRSEC. The implementation in PaRSEC is proposed in Section 5. Section 6 depicts runtime support and optimizations for redistribution workloads. Section 7 introduces the cost model, including cost analysis, actual bandwidth, and *bound* declaration. Performance results and analysis, along with application demonstrations, are illustrated in Section 8. Finally, we conclude and present future work in Section 9.

2 Related Work

2.1 The Runtime System

Numerous efforts are ongoing to support fine-grained dataflow programming. In this section, we briefly refer to the task-based runtimes focusing on data flows.

QUeuing And Runtime for Kernels (QUARK) [36], [37], StarPU [6], and OmpSs [7] provide a task-insertion application programming interface (API) and dynamically build the task-graph. To interact with the runtime to provide dependencies, the developer expresses sequential loop nests containing asynchronous task insertion calls. A consequence in distributed settings is all of the participating processes have to discover the entirety of the graph to infer communication before reducing to the set of local tasks; otherwise, expertise information about the algorithm needs to be provided to the runtime system carefully by users. This pruning phase may limit potential scalability [38].

QUARK has no implicit support for heterogeneous nor distributed architectures and is used to allow kernel routines to execute asynchronously in parallel on a shared-memory architecture.

StarPU is a simple tasking API that provides numerical kernel designers with a convenient way to execute parallel tasks over heterogeneous architectures. On the other hand, it provides a method to easily develop and tune powerful scheduling algorithms, via the insertion of implicit point-to-point communication tasks [39] and limited collective communication tasks, assuming all dependencies related to a collective communication need to be discovered when that collective communication is performed [40].

OmpSs is a task-based programming model used as a forerunner for OpenMP and is based on compiler directives with heterogeneous architectures support. In addition, COMP Superscalar (COMPSs) [41] is developed to ease the development of applications for distributed infrastructures, which provides a programming interface for the development of the applications and a runtime system that exploits the inherent parallelism of applications at execution time.

Recent versions of the OpenMP specification [42] introduce the *task* and *depend* clauses, which can be employed to express dataflow graphs. OpenMP is widely used and supports homogeneous, shared-memory systems, and its *target* extension to support accelerators is quickly gaining traction. However, distributed-memory and inter-node communication in OpenMP need to be described explicitly and performed with the use of an external communication library (e.g., MPI, SHMEM).

Legion [43] describes logical regions of data, which are used to describe organization of data and to make explicit relationships useful for reasoning about locality and independence. It uses a low-level runtime, REALM [44], to schedule and execute tasks and uses GASNet as the underlying communication layer, which supports heterogeneous architectures and works in shared- and distributed- memory systems.

HPX [45] is an open-source implementation of the concepts of the ParalleX execution model, developed for conventional architectures and, currently, Windows and Linux-based systems (e.g., large non-uniform memory access [NUMA] machines and clusters).

The SuperMatrix [8] approach is similar in motivation and technique to SMP superscalar (SMPSs)[46], which is exclusively focused on linear algebra algorithms. No pragmas or specific programming model is defined, since the runtime directly considers for parallelization a set of linear algebra routines. SuperMatrix also implements a task dependency analysis, but in this case, data renaming and/or redistribution is not considered.

Open Community Runtime (OCR) [47] is a work-in-progress effort to create a low-level, task-based runtime for extreme-scale parallel systems, with support for fault-tolerance. It currently supports homogeneous architecture in distributed systems and uses Intel Threading Building Blocks (TBB) to manage threading.

2.2 Redistribution

For more than two decades, research on data redistribution has evolved around regular data distributions. In the 1990s, research about array and data redistribution sprung up after the appearance of HPF [15], [17], [18], [48]. For instance, Akiyoshi Wakatani *et al.* [15] proposed a new scheme, *strip mining redistribution* to reduce the communication time for arrays redistribution; Antoine P. Petitet and Jack J. Dongarra [48] presented various data redistribution methods for block-partitioned linear algebra algorithms operating on dense matrices that were distributed in a block-cyclic fashion and introduced techniques redistributing data ”on the fly”, to make the data distribution blocking factor independent from the architecture-dependent algorithmic partitioning.

In the 2000s, research spreads to more broad fields [25], [49], [50]. Early 2000’s, frameworks such as ReSHAPE [50] were developed to support dynamic resizing of parallel MPI applications executing on distributed-memory platforms, including support for releasing and acquiring processors and efficiently redistributing application state to a new set of processors. Similarly, [25] designed a new data distribution operation $MPI\_Alltoall\_specific$. It is based on collective MPI operations, point-to-point communication operations, or parallel sorting, that allows an element-wise distribution of data elements to specific target processes

TABLE 1
Parameters and Notations

Symbol	Description
$\mathcal{R}$	Function or routine for redistribution
SRC	Source data descriptor
TG	Target data descriptor
A_{sub}	Submatrix to be redistributed
$size_{\{row,col\}}$	Row/column size of A_{sub}
$disp\_row_{\{s,t\}}$	Row displacement in source(s)/target(t)
$disp\_col_{\{s,t\}}$	Column displacement in source(s)/target(t)
$\{M,N\}_{\{s,t\}}$	Row(M)/column(N) size of source(s)/target(t)
$\{MB,NB\}_{\{s,t\}}$	Row(MB)/column(NB) tile size of source(s)/target(t)
$D_{\{s,t\}}$	Data distribution of source(s)/target(t)
$\{P,Q\}_{\{s,t\}}$	Row(P)/column (Q) distribution of source(s)/target(t)
$\{SMB,SNB\}_{\{s,t\}}$	Row(SMB)/column (SNB) block distribution of source (s)/target(t)
$\{m,n\}_{\{s,t\}}$	Tile row(m)/column(n) index of source(s)/target(t)
local	Source and target data on the same process
remote	Source and target data on different processes
random	Process ID of each tile is achieved by random number generator
SEGMENTS	*NW*, *N*, *NE*, *W*, *I*, *E*, *SW*, *S*, and *SE*

and is used to implement irregular data distribution operations, for example, in particle codes.

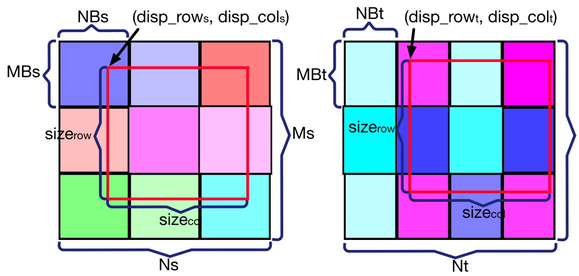
More recently, we witnessed a resurgence of interest in data redistribution due to increasingly complex applications, which need to relocate data distributed across one grid onto a different distribution scheme across another grid to improve data locality and/or reduce the cost of data movement [21], [22], [24], [26], [51]. [51] as a representative, studied the complexity of the problem—”finding a re-mapping of data items onto processors such that the data redistribution cost is minimal and the operation remains as efficient”—computed optimal solutions, evaluated through simulations, and showed the NP-hardness to find the optimal data partition and processor permutation (defined by new subsets) that minimized the cost of redistribution followed by a simple computational kernel.

However, all research on the array or data redistribution: (1) focused on a simplified problem, i.e., the regular distributions BCDD; (2) tried to address load imbalance caused by the data distribution, but ignored impact from data size. They also highlighted that with the increasing complexity of hardware architectures and communication topologies, targeting only regular distributions BCDD is not enough. As task-based runtime systems emerge, interest in data redistribution and resource management increases; for instance, DMRlib [52] enable job reconfiguration as an OmpSs extension providing support for malleable application by allowing users to provide their own redistribution functions with explicit communications. However, none of them deals with the data redistribution problem itself in a task-based runtime system. Therefore, a flexible redistribution algorithm, taking into account not only regular and irregular data distribution but also the impact of data size in a task-based runtime system, becomes necessary.

3 Redistribution

3.1 Problem Definition

A general redistribution problem $\mathcal{R}$ is a function or routine to change distribution schemes (Table 1 describes parameters and notations): $\mathcal{R}$: SRC $\rightarrow$ TG with the following properties:



(a) Source SRC (b) Target TG

Fig. 1. General redistribution problem; matrix is stored in tile format, each color represents a different process, and rectangle circled in red is the submatrix to be redistributed.

- Source SRC with the distribution D_s, possibly *random* (details in Section 6.1) or on a process grid $P_s \times Q_s$, $P_s, Q_s > 0$;
- Target TG with the distribution D_t, possibly *random* or on a process grid $P_t \times Q_t$, $P_t, Q_t > 0$;
- Submatrix A_{sub} to be redistributed with size of $size\_row \times size\_col$ and with displacements ($disp\_row_s$, $disp\_col_s$) in SRC and ($disp\_row_t$, $disp\_col_t$) in TG, and A_{sub} should not exceed the bounds of SRC and TG.

Fig. 1 depicts a general redistribution problem in matrix format, redistributing a submatrix from SRC to TG with different distributions and tile sizes. While the problem is generic, in our particular context A_{sub} is to be redistributed between two dense matrices stored in tile format. There are several features that need to be clarified:

- D_s and D_t could be *random*; more general than concepts of redistribution in related work, which is a regular data distribution along each dimension of the array.
- Tiles in SRC and TG are rectangles, not specified as square, and MB_s, NB_s, MB_t, NB_t are independent with D_s and D_t.
- Displacements of ($disp\_row_s$, $disp\_col_s$) in SRC and ($disp\_row_t$, $disp\_col_t$) in TG could be any points not exceeding the bounds of SRC or TG respectively.

3.2 Algorithm Description

For a general redistribution problem, to redistribute a submatrix between two matrices with different distributions, tile sizes, and displacements, an efficient algorithm should be as flexible as possible to deal with all possible cases. Hence, as shown in Fig. 2, zooming in one tile in TG to catch its source data in SRC, we split the TG tile into nine parts, or SEGMENTS, according to their location in SRC, NorthWest (*NW*), North (*N*), NorthEast (*NE*), West (*W*), Inner (*I*), East (*E*), SouthWest (*SW*), South (*S*) and SouthEast (*SW*). Fig. 3 shows the possible categories based on the existence of SEGMENTS, determined by combinations of $size\_row$, $size\_col$, MB_s, NB_s, MB_t, NB_t and location of TG tiles' starting points in SRC. The idea is that:

- each tile in TG is independent;
- focusing only on a tile in TG with fixed MB_t and NB_t, and varying MB_s and NB_s at their dimension independently from ε to ∞, where ε is an infinitesimal number and ∞ an infinite number.

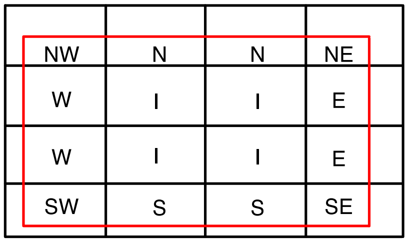


Fig. 2. The red rectangle represents one target TG tile while black rectangles are the corresponding source SRC tiles.

Hence, all possible cases of general redistribution problems are extensions of these nine categories, including several *N*, *S*, *W*, *E* or *I*, e.g., Fig. 2 is an extension of Fig. 3 (8).

Algorithm 1. Serial Algorithm of Redistribution

```
for $m_t = disp\_row_t / MB_t$ to $(size\_row + disp\_row_t\text{-}1) / MB_t$ do
  for $n_t = disp\_col_t / NB_t$ to $(size\_col + disp\_col_t\text{-}1) / NB_t$ do
    Calculate $m_s\_start$, $m_s\_end$, $n_s\_start$, and $n_s\_end$ that ($m_t$, $n_t$) associated with
    for $m_s = m_s\_start$ to $m_s\_end$ do
      for $n_s = n_s\_start$ to $n_s\_end$ do
        if Remote then
          Send SEGMENTS
        end if
      end for
    end for
    if Remote then
      Receive SEGMENTS
    else
      Copy SEGMENTS
    end if
  end for
end for
```

The serial algorithm, revealed in Algorithm 1, follows the idea that for tiles in TG, send/receive (when *remote*) or copy (when *local*) SEGMENTS. Memory copy is the additional overhead, but it is necessary.

- SRC and TG are two different data descriptors with exclusive data storage; therefore, memory copy is necessary to connect different memory pieces, i.e., SEGMENTS, between SRC and TG when they are *local*.
- When SEGMENTS in SRC and TG are *remote*, the shape (or memory layout) for each of the SEGMENTS that needs to be transmitted varies on both the sender and receiver side; this shape is problem-dependent, determined during runtime by a combination of $size\_row$, $size\_col$, MB_s, NB_s, MB_t, NB_t, $disp\_row_s$, $disp\_col_s$, $disp\_row_t$, and $disp\_col_t$, and is hardly predicted in advance. If a new MPI data type (typically MPI_Type_vector) is committed for every shape of data, it will be very expensive [53], [54], because each new derived data type may be utilized only once and possibly tens of thousands have been created (the maximum number of potential derived data types is $min(MB_s, MB_t) \times min(NB_s, NB_t)$). Hence, memory copy is utilized to send from a contiguous buffer and receive to a contiguous buffer in this case.

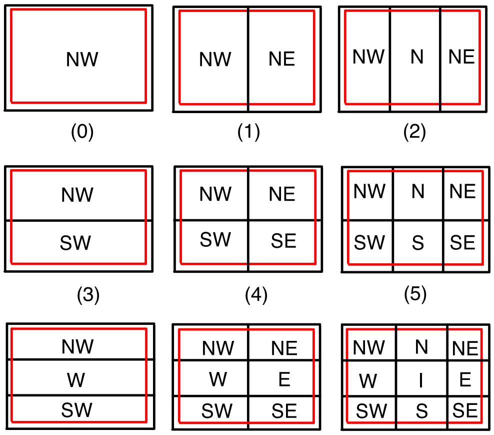


Fig. 3. All possible nine categories; red rectangles represent one TG tile while black rectangles are the corresponding SRC tile(s).

The benefits of this design are that it

- is capable for coarse- and fine- grained redistribution problems for tile- or block- based matrix partition;
- isolates distribution and tile size; actually, it could solve a redistribution problem with absolute flexibility on distribution, tile size and displacement;
- suits tile format libraries with either ScaLAPACK memory layout or DPLASMA memory layout (details in Section 8.7.2).

In this way, all possible redistribution problems could be reduced to a combination of these nine SEGMENTS, and operations on these SEGMENTS could be considered as tasks which thus could be efficiently handled by a multi-threaded task-based runtime system.

4 The PaRSEC Runtime System

PaRSEC [34] is a generic task-based runtime system for asynchronous, architecture-aware scheduling of fine-grained tasks on distributed many-core heterogeneous architectures. It is capable of dynamically unfolding a concise description of a graph of tasks on a set of resources and satisfying all data dependencies by efficiently shepherding data between memory spaces (between nodes but also between different memories on different devices) and scheduling tasks across heterogeneous resources. Overall, the PaRSEC runtime system is designed to overcome the four challenges towards algorithm scalability and efficiency:

1) *starvation*, a problem encountered in concurrent computing, when there is insufficient concurrent work available to maintain high utilization of all resources;
2) *latency*, the time-distance delay intrinsic to access remote resources and services and causing delays due to oversubscribed shared resources;
3) *overhead*, any additional work required for the management of parallel actions and resources on the critical path of execution, which is not necessary for a sequential variant;
4) *heterogeneity*, systems that use more than one kind of processor or core, supporting for specialized

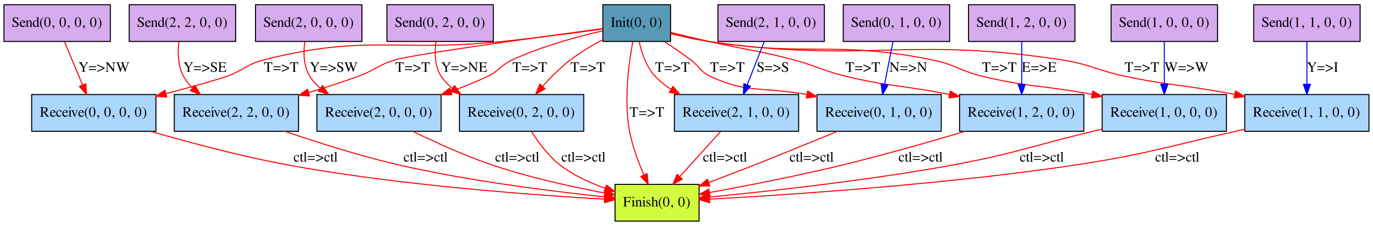


Fig. 4. Corresponding DAG for an example detailing the dependencies between tasks. Arrow → represents dependency between tasks; red means *local* and blue *remote*. ⇒ shows data or control dependency (ctl) between tasks. Init, Send, Receive and Finish are the four task classes; Init and Finish with parameters (m_t, n_t), and Send and Receive with (m_s, n_s, m_t, n_t). Besides the nine SEGMENTS, **Y** and **T** represent tiles of SRC and TG respectively.

hardware to maximize performance (accelerators) and minimize overheads (smart communication hardware/NIC).

Like most task-based runtime systems, algorithms are described to PaRSEC through computations and the corresponding data by means of tasks and dependencies: computations become entities (a.k.a., tasks, a set of instructions that access and modify an explicit and bounded amount of data), and the data flowing among them are the dependencies; therefore, algorithms can be represented as a DAG with vertices as tasks and edges as dependencies. Several domain-specific languages (DSLs) [55] in PaRSEC are used to express the DAG, which helps domain scientists to focus only on their domain knowledge instead of low-level computer science aspects, such as the complex hardware architectures, hierarchical memory layout, different types of communication prototypes, etc.

The DSL, Parameterized Task Graph (PTG) [56], used to describe our redistribution algorithm, utilizes a concise, parameterized task-graph description called Job Data Flow (JDF) to represent the dependencies between tasks, which could be considered as a collection of task classes containing information that enables the creation and execution of the task instances. Different types of communications, one-to-one, one-to-many, and many-to-many, are supported in PTG to enhance the productivity of the application developers. These types are implicitly inferred by the expression of the tasks in PTG. Algorithms written in PTG are capable of delivering a significant percentage of the hardware peak performance on many hybrid distributed machines in many scientific applications. Among the successful usages of PaRSEC and PTG, we can enumerate a linear algebra library for dense matrices, DPLASMA, that yields superior performance compared with the most widely used library, ScaLAPACK [30], or compared with state-of-the-art in computational chemistry [57], [58] and in climate and weather prediction [13], [33], [59].

Other DSLs, such as Dynamic Task Discovery (DTD) [38], are less domain science–oriented and provide alternative programming models to satisfy more generic needs by delivering an API that allows for sequential task insertion into the runtime instead of expressing in a parameterized manner. This programming model is simple and straightforward and delivers good performance on small and medium-sized platforms, but it suffers from the same high overhead due to the sequential discovery of tasks that hinders the scalability of other distributed task-insertion runtimes, such as StarPU or QUARK.

5 Implementation in PaRSEC Runtime System

The PTG DSL is adopted along with data descriptor in PaRSEC and DPLASMA, forming the following interface:

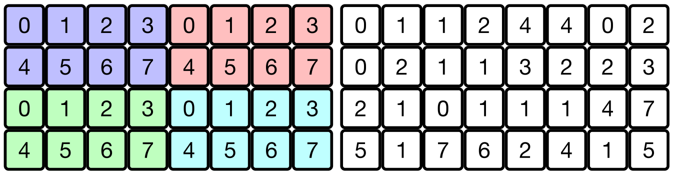
```
int parsec_redistribute(
  parsec_context_t *parsec,
  parsec_tiled_matrix_dc_t *source,
  parsec_tiled_matrix_dc_t *target,
  int size_row, int size_col,
  int disp_row_s, int disp_col_s,
  int disp_row_t, int disp_col_t)
```

Because PaRSEC is an event-driven system, to implement Algorithm 1 in PaRSEC and expose all potential parallelism, four different types of tasks (a.k.a task classes) are specified:

- Init: initialize event and prepare TG data for tasks in task classes Receive and Finish;
- Send: on SRC process, send SEGMENTS to the other process if *remote* or pass the address on shared memory if *local*;
- Receive: on TG process, receive SEGMENTS from other process if *remote* or copy SEGMENTS on shared memory if *local*;
- Finish acts as synchronization/finalization for each tile in TG with multiple SEGMENTS to finish all related tasks in Receive and serves for control dependency optimization.

Fig. 4 presents an example detailing the dependencies between tasks for a simple redistribution problem to demonstrate the case in Fig. 3 (8): redistributing SRC having 3 × 3 tiles with distribution 2DBCDD of $\mathtt{P}_s \times \mathtt{Q}_s = 2 \times 2$ to TG having 1 tile on process ID 0, size of SRC, TG and A_{sub} are the same, and displacements $disp\_row_s$, $disp\_col_s$, $disp\_row_t$ and $disp\_col_t$ are 0. There is only one tile in target TG (0, 0), so one task in Init and Finish respectively; however, nine tiles are involved in SRC, indexing from (0, 0) to (2, 2), hence there are nine correlated tasks in Send and Receive respectively.

For the purpose of data locality, Send is local to SRC's tiles, while Init, Receive, and Finish reside on TG's tiles. In addition, to increase computational resources occupancy and network bandwidth, two kinds of control dependencies are utilized to re-configure the DAG to (1) maximize the parallelism exposed to the runtime on the receiver side if there are multiple SEGMENTS within a TG tile, and (2) reduce runtime overheads for task management. Moreover, to efficiently utilize network bandwidth on the sender side, as much data movements as possible should be exposed to



0	1	2	3	0	1	2	3
4	5	6	7	4	5	6	7
0	1	2	3	0	1	2	3
4	5	6	7	4	5	6	7

(a) 2DBCDD

0	1	1	2	4	4	0	2
0	2	1	1	3	2	2	3
2	1	0	1	1	1	4	7
5	1	7	6	2	4	1	5

(b) Random distribution

Fig. 5. 2DBCDD and *random* distribution with matrix size 4×8 on 8 processes; each number indicates the process ID for that tile. (a) 2DBCDD with process grid of 2×4; (b) process ID of each tile generated by a random number generator with seed 2783.

the communication library as early as possible, to provide both overlap between data copies to/from multiple peers, and allow the runtime to aggregate ”multiple send exposed simultaneously”. More details are described in the following Section 6.

6 Runtime Support and Optimizations

In this section, we detail the runtime-level support and optimizations. Runtime support extends the redistribution implementation to a novel field in the runtime world, and the three runtime-level optimizations deliver more efficient performance. We believe the runtime support and optimizations could go beyond the scope of this paper.

6.1 Runtime Support: Random Distribution

In addition to the traditional regular distributions BCDD on which the dense linear algebra community has been relying for more than two decades, PaRSEC supports any type of irregular data distribution. This includes the hybrid data distribution [13], which logically superposes two intertwined 2DBCDDs of different process grids together, to mitigate the load imbalance of a 2DBCDD towards the sparse linear algebra algorithms. In addition, PaRSEC supports *random* distribution (process ID for each tile is randomly generated), which is an extreme case of irregular data distributions. Fig. 5 shows an example of 2DBCDD and *random* distribution: (a) with process grid 2×4, and (b) the process ID for each tile randomly generated by seed 2783. This support of *random* distribution demonstrates the ability of our redistribution algorithm to tackle redistribution problems with irregular data distributions and extends the implementation to an untouched field.

In PaRSEC, the function $rank\_of$ allows the user to define the data distribution in a flexible way. The user can supply any mapping of data to processes, or any random function of the multi-dimensional tile index (e.g., m, n for a 2D space) to indicate at which rank the indexed data block is to be located. Similarly, the $data\_of$ function returns the corresponding data descriptor (i.e., a handle to the data memory block) for the related task when invoked at that rank. From a distributed-memory perspective, tiles on each process should be capable of retrieving their own information (e.g., location of $data\_of$ stored on that process). Therefore, in order to represent a random distribution, we implement these functions using a global structure $table\_tile$, with respect to the number of tiles, which stores rank and data information needed for each tile. Take SRC for instance; the size of $table\_tile$ will be $ceil(\frac{M_s}{MB_s}) \times ceil(\frac{N_s}{NB_s})$ to store information for every tile; for each element in $table\_tile$, every process will check its locality based on $rank\_of$, and only set information / allocate memory when it is *local* to avoid redundant memory allocation. Therefore, no data coherency condition can occur, and a globally unique key could be used for the retrieval.

6.2 Runtime Optimizations

6.2.1 Control Dependency

In a data flow task-based runtime like PaRSEC, the number of classes of input/output flow/data is determined in the PTG DSL, and data of each input flow is unique for each task (i.e., one input flow in one task could not receive multiple data simultaneously). This is generally true for most of parallel programming models based on data-flow. Task parallelism is discovered using data dependencies between tasks, leading to two issues.

- PaRSEC, as a general runtime system, enables tasks as soon as all their dependencies are available, and can therefore enable maximum parallelization, whose efficiency has been proven by DPLASMA and in many scientific fields [13], [57], [58], [60], [61]. Redistribution is a particular algorithm where there is no predecessor for all tasks in Send, which means all these tasks will be exposed to the runtime system simultaneously. The number of these tasks is no less than $ceil(\frac{size\_row}{min(MB_s,MB_t)}) \times ceil(\frac{size\_col}{min(NB_s,NB_t)})$. Hence, millions or even billions of tasks will be exposed to the runtime system simultaneously, leading to higher overheads to maintain the list of ready tasks, higher risk of inappropriate decisions made by runtime with fewer possibilities of reusing temporary buffers, and unnecessarily increasing the cost of scheduling.
- A TG tile could contain multiple SEGMENTS in redistribution as shown in Fig. 2. Ideally, these SEGMENTS within a tile should be received and written independently of the proper memory location to maximize parallelism on the receiver side. However, the minimal data unit in PaRSEC is a tile, resulting in tasks written on the same tile being serialized, limiting the potential parallelism.

Applications are complex entities, their requirements can not be enumerated into the design principles of a programming paradigm or runtime system. Therefore, a well-designed runtime system should limit the number of constraints it imposes on developers and instead provide fundamental, hopefully efficient, capabilities supported by a flexible control. As mentioned before, PaRSEC enables tasks as soon as all their dependencies are available and therefore enables maximum parallelization; meanwhile, the concept of control dependency in PaRSEC allows users to dynamically control the workflow. This control dependency is an artificial dependency without data encapsulated and can be used to guide the task execution order and priorities, as well as to limit the number of tasks revealed to the runtime at a given time. Using this control dependency, we optimize the redistribution workflow in the following two ways on the sender and receiver side respectively.

- The first is a batch parameter (or columns of tiles) to limit the number of concurrent tasks exposed to the runtime system on the sender side. It batches the number of SRC columns of data being under transfer at the same time, because there is no predecessor for tasks in Send and they could be exposed to runtime simultaneously. By adding control dependencies between tasks in Finish and tasks in the next batch of Send, we prevent tasks beyond the column scope of the current batch from being discovered. To maximize hardware occupancy, batch could be the lowest common multiple (LCM) of $Q_s \times SNB_s$ and $Q_t \times SNB_t$ for the normal 2DBCDD distributions, while for redistribution between *random* distributions, a suitable batch could be chosen, e.g., $min(ceil(size_{col/}NB_t), num\_of\_nodes)$.

- The other is to maximize parallelism on the receiver side for cases of multiple SEGMENTS within a TG tile. That is to add a local control dependency between tasks in task classes Receive and Finish that isolate tasks to receive data—there are control dependencies between every task in Finish and task/tasks in Receive with SRC data related to TG data in that task. For instance, in Fig. 3 (8), the task in Finish with the TG tile circled in red has nine dependencies with tasks in Receive receiving the nine SEGMENTS, as shown in Fig. 4. Therefore, with this control dependency, all tasks to receive SEGMENTS within a TG Tile are independent, maximizing the parallelism exposed to the runtime system.

6.2.2 Dynamic Communication Volume

As mentioned above, the size of each SEGMENTS in Fig. 2 varies, so messages of variable size would be sent/received. Hence, if data is communicated in full dense tiles like in DPLASMA, there will be a significant communication overhead, especially for a redistribution algorithm which is by nature communication-bound. PaRSEC supports sending variable-sized data to remote processes even when this size is dynamically decided when the task produces the corresponding data, by specializing the information about the data to the communication engine in the activation message. This feature is common in communication libraries such as MPI, but it is relatively new in task-based runtimes so far, as most of the runtimes mentioned in related work are still trying to cope with mostly regular, dense cases. We used this feature in Lorapo [13] and extended it to the redistribution implementation. Taking advantage of this PaRSEC capability makes it possible to minimize the data transfers to the actual size, providing a possible path toward communication optimality. Moreover, such a feature may alleviate the bandwidth saturation and communication overhead, while releasing temporary memory pressure on the receiver side.

6.2.3 Multiple Send Exposed Simultaneously

In the PaRSEC runtime, communication is implicit and produced from the resolution of dependencies between data flows. When a task completes, it issues control messages to the target ranks that will execute the successor tasks (as described by the user-supplied process affinity of the task). Then, the target rank will issue the communication orders asynchronously and mark the target task ready when all data movement has been satisfied. In order to maximize the overlap between computation and communication and to enable the asynchronous progress, the PaRSEC runtime delegates the issue of control messages and data movement to a separate, runtime-internal thread. Control orders are passed to that thread through a thread-safe command dequeue. In our early experiments, we discovered that this thread-funneled access to the communication engine could limit the rate of control messages and small data messages issued [62], and that will restrict the effective bandwidth achieved by the redistribution algorithm which consists of pure communication, except the index computation of very low arithmetic intensity, and possible small SEGMENTS.

To address this issue, we have enabled the computational threads to directly issue communication orders for control messages and short data messages (assuming a thread-safe communication library). Thus, instead of funneling communications, when a thread completes a task, it will directly issue the communication for the control active messages needed to notify the dependent ranks of the task completion. When the data payload is small enough (user-configurable threshold), the payload may also be aggregated inside the control message as piggyback, completely removing the need for further communications. The reception of messages remains single-threaded at this point with a single progress thread managing incoming active messages and issuing the data transfers orders as needed. This helps improve communication performance in two ways: first, this optimization bypasses the serialization at the command queue, which improves latency and reduces the number of atomic operations performed to ensure thread-safe access to that queue. Second, a single thread is incapable of saturating the injection rate of modern high-performance networks; thus, enabling multiple threads to access the network improves the resource utilization of network bandwidth.

7 Cost Model

7.1 Cost Analysis

Data redistribution in HPF assumes two kinds of costs: index computation cost T_{index} and inter-processor communication cost $T_{communication}$ [19], which is also true for the redistribution algorithm proposed in this paper.

- T_{index} is incurred on both the sender and receiver side in calculating source and target processor and the location of the element within that processor, including displacements and shape for each SEGMENTS.
- $T_{communication}$ is incurred when data is exchanged between processors, which could be represented using an analytical model of typical distributed-memory machines, the General purpose Distributed Memory (GDM) model [63]. The GDM model represents the communication time of a message passing operation using two parameters: the start-up time $T_{startup}$ and the unit data transmission time T_{trans}. The $T_{startup}$ is the dominant overhead for small messages, while the T_{trans} becomes significant as the message size increases.

Besides these two costs, in our redistribution implementation in PaRSEC task-based runtime system, we introduce two extra costs: memory copy cost T_{memcpy} and runtime cost $T_{runtime}$.

- T_{memcpy} comes from two aspects (as detailed in Section 3.2): copying data from `SRC` to `TG` if *local*; and copying data from `SRC` to a temporary buffer and from a temporary buffer to `TG` if *remote*.
- $T_{runtime}$ is caused by those such as spawning, maintaining and scheduling tasks, dependency resolution, and memory management, which is $\frac{N \times C_T}{P \times n}$ as shown in [38] where N as the total number of tasks, C_T as the cost/duration of each task, P as total number of process and n as the number of actual cores in each process. [64] provides interesting evaluations when $T_{runtime}$ could be overlapped by computation/communication in a runtime system; as to date redistribution with pure communication bound, the larger $T_{communication}$, i.e., message size, the more possible $T_{runtime}$ could be overlapped.

Combining all costs together, the cost of redistribution is

$$T = T_{index} + T_{communication} + T_{memcpy} + T_{runtime} \qquad (1)$$

7.2 Bound Declaration

Suppose n is the total number of processes, i is the process ID, and T^i is the execution time for process P^i. Then, the total time for a parallel program is defined by the execution time of the slowest process [2]: $T = max_{i=0}^{n-1}(T^i)$. Hence, renaming $T^i_{extra} = T^i_{runtime} + T^i_{index}$ allows us to re-write Eq. (1) as

$$T = max_{i=0}^{n-1}(T^i_{communication} + T^i_{memcpy} + T^i_{extra}) \qquad (2)$$

Assume *remote* message size that process P^i sends and receives are M^i_{send} and $M^i_{receive}$ respectively. Because all communications in the redistribution algorithm are peer-to-peer, the actual bandwidth achieved is the maximal amount of messages that is sent or received among processes in a certain time period

$$actual\_bandwidth = max_{i=0}^{n-1}(max(M^i_{send},\ M^i_{receive}))/T, \qquad (3)$$

which is what we use in the following experiments.

For the bound of $actual\_bandwidth$, suppose $M^i_{remote} = max(M^i_{send},\ M^i_{receive})$, M^i_{local} is *local* message size for P^i, B_{net} and B_{memcpy} are the theoretical network bandwidth between two processes and the memory bandwidth respectively, then

$$T = max_{i=0}^{n-1}\left(\frac{M^i_{remote}}{B_{net}} + 2 \times \frac{M^i_{remote}}{B_{memcpy}} + \frac{M^i_{local}}{B_{memcpy}} + T^i_{extra}\right) \qquad (4)$$

where on process P^i, $\frac{M^i_{remote}}{B_{net}}$ represents the communication cost, $2 \times \frac{M^i_{remote}}{B_{memcpy}}$ shows the two *remote* memory copy costs (details in Section 7.1), and $\frac{M^i_{local}}{B_{memcpy}}$ depicts the *local* memory copy cost.

Suppose $M_{remote} = max_{i=0}^{n-1}(M^i_{remote}),\ M_{local} = max_{i=0}^{n-1}(M^i_{local})$, $T_{extra} = max_{i=0}^{n-1}(T^i_{extra})$ and $r = M_{local}/M_{remote}$, then according to Eq. (4)

$$bound = \frac{(1 - T_{extra}/T) \times B_{net} \times B_{memcpy}}{(2 + r) \times B_{net} + B_{memcpy}} \qquad (5)$$

where *bound* is the maximum network bandwidth that this redistribution algorithm could be achieved. It shows the smaller extra overheads (T_{extra}/T) and memory copy overheads (B_{net}/B_{memcpy}), the higher network bandwidth could be gained towards the theoretical network bandwidth B_{net}, which makes sense: with the smaller ratio of additional overheads (caused by index computation, memory copy and runtime), higher network bandwidth is achieved.

8 Performance Results and Analysis

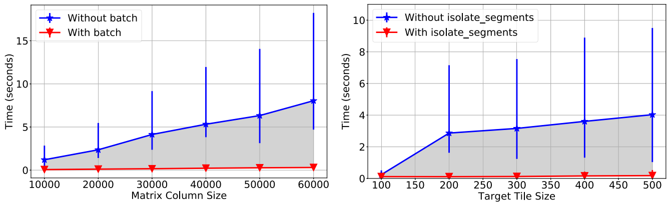
8.1 Experiments Settings

Experiments are conducted on two HPC clusters with different ratios of computational capacity to network bandwidth: `NaCL` and `Shaheen II`.

- `NaCL` includes 66 compute nodes connected by InfiniBand QDR, and each node has two 2.8 GHz Intel Xeon X5660.
- `Shaheen II` is a Cray XC40 system with 6,174 compute nodes; each node is equipped with two 16-core Intel Haswell CPUs running at 2.30 GHz and 128 GB DDR4 RAM; the interconnect is Cray Aries with Dragonfly topology.

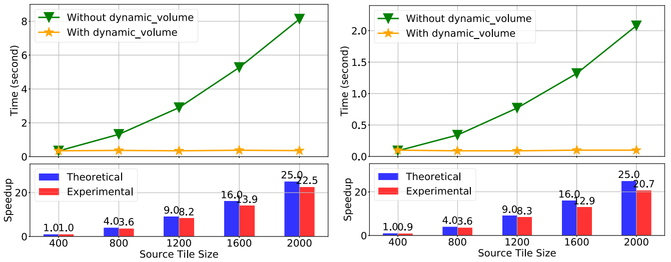
The actual achieved bandwidth in experiments is calculated by Formula 3 where $max_{i=0}^{n-1}(max(M^i_{send},\ M^i_{receive}))$ is problem-dependent and T is actual execution time. This actual bandwidth is compared with *bound* in Formula 5, and we assume there is no memory constrain for data redistribution. Here are some common settings used for the experiments.

- Communication relies on Open MPI 4.0.0 on `NaCL` and Cray MPICH 7.7.0 on `Shaheen II` (both initialized in thread multiple mode).
- For better performance, a hybrid model (MPI + Pthreads) is deployed in `PaRSEC`, i.e., one process per physical node, guaranteeing for each process the same maximum bandwidth.
- B_{net} in Formula 5 is measured by the `NetPIPE` [65], a widely used benchmark to get upper-bound of network bandwidth for different message sizes between two processes, and one process is deployed on each node for fair comparison. It is worth noting that all communications in redistribution are peer-to-peer, and that is where the cost model in Section 7.2 is based. Hence, we conduct experiments in Section 8.2 and 8.3 on two processes (a.k.a., nodes in `PaRSEC`) to better show the effect of optimization itself and comparison to *bound* because B_{net} is produced between two nodes.
- B_{memcpy} in Formula 5 utilizes the theoretical memory bandwidth, which *potentially leads to a lax bound*.
- The *bound* is calculated by setting $T_{extra/}T = 0$, because runtime overheads are hard to quantify, especially considering the potential overlap of computations, communications and runtime overheads [64]; *so bound may be increased again by this setting.*
- We use weak and strong scaling. The number of tiles of each node is maintained for *weak scaling*, while the total number of tiles is maintained for *strong scaling*.
- If not specified, we present the average over 20 runs.



(a) `batch` (b) `isolate_segments`

Fig. 6. Effect of `control_dependency`; lower is better.



(a) Results on `NaCL` (b) Results on `Shaheen II`

Fig. 7. Effect of `dynamic_volume`; lower is better.

- To simplify settings but without losing generality, in experiments, the entire matrix is redistributed, i.e., $M_s = M_t = size_{row}$, $N_s = N_t = size_{col}$, and displacements are 0; if not specified, tiles are square, that is $MB_s = NB_s$ and $MB_t = NB_t$.
- Data is stored as two dimensional matrix using the data descriptor in `PaRSEC`, and all calculations and communications are performed in double-precision floating-point arithmetic, so the message size is $nb\_elms \times 8$ Bytes (nb_elms is the number of elements in each `SEGMENTS`).

8.2 Incremental Effect of Runtime Optimizations

This section details the incremental effects of runtime-level-related optimizations hereinbefore. As a summary, the target of each optimization is listed here.

- Control dependency (`control_dependency`, Section 6.2.1): (1) `batch` limits the number of concurrent tasks exposed to the runtime system on the sender side so that it benefits cases with a huge amount of tiles to be redistributed; (2) the second type is `isolate_segments` which isolates `SEGMENTS` within a `TG` tile to receive data to maximize parallelism on the receiver side, so that it has effects when multiple `SEGMENTS` are within a `TG` tile.
- Dynamic communication volume (`dynamic_volume`, Section 6.2.2): suits almost all cases except those only reshuffling distributions like the most in the related work, i.e., the whole matrix is redistributed, tile sizes of `SRC` and `TG` are the same, and displacements are 0.
- Multiple Send Exposed Simultaneously (`multiple_send`, Section 6.2.3): targets cases with small messages that could not hide the runtime overheads $T_{runtime}$.

8.2.1 Effect of Control Dependency

Two kinds of `control_dependency` are proposed, and their incremental evaluations are shown in Fig. 6 on `NaCL` when redistributing between two different 2DBCDDs. Fig. 6a shows the effect of `batch`. As mentioned before, `batch` is especially beneficial for redistributing a huge amount of tiles; hence, in this figure, matrix column size N_s increases while other settings, except distributions, remain constant, i.e., $M_s = M_t = 2000$ and tile sizes of `SRC` and `TG` are 100×100, so that the number of tiles within each `batch` is the same. Fig. 6b shows the effect of `isolate_segments`. In this case, matrix size and `SRC` tile size stay the same, i.e., matrix size of `SRC` and `TG` is 10000×10000 and `SRC` tile size $MB_s = NB_s$ =100, while the `TG` tile size $MB_t = NB_t$ varies, so that an increasing number of `SEGMENTS` exist within a `TG` tile.

Figs. 6a and 6b show the great improvement achieved when using `control_dependency`, which is more significant in terms of performance and fluctuation as N_s (i.e., the number of tiles) and MB_t (i.e., the number of `SEGMENTS` in a `TG` tile) increase when using `batch` and `isolate_segments`, respectively.

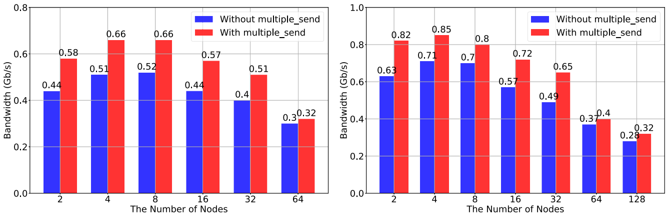
8.2.2 Effect of Dynamic Communication Volume

`PaRSEC` can dynamically handle variable-sized data by sending only the necessary size. During redistribution for each `SEGMENTS`, only the actual data is transmitted instead of sending all the $MB_s \times NB_s$ elements associated in an `SRC` tile. This significantly decreases the communication volume and thus the cost of T_{trans}, as well as the memory usage on the receiver, because the receiver buffers can now be tightly allocated with the real target tile size. Fig. 7 shows the performance comparison with and without `dynamic_volume` on `NaCL` and `Shaheen II`. To better show the effect, a large message size is chosen to minimize the runtime overhead. Thus MB_t (NB_t) is fixed to 400, and MB_s (NB_s) is increased from 400 to 2000 by increments of 400. The redistribution pattern of this experiment is to convert the tile size of a matrix from MB_s to MB_t, while keeping the data distribution constant (2DBCDD).

Usually, the speedup of the `dynamic_volume` for a general redistribution problem with arbitrary displacement is $\frac{NUM \times MB_s \times NB_s}{SIZE}$, where NUM is the total number of remote `SEGMENTS`, and $SIZE$ is the total size of remote `SEGMENTS`. In this particular setting along with displacements being zero, the speedup achieved is about $\frac{MB_s}{MB_t} \times \frac{NB_s}{NB_t}$. As seen in Fig. 7, `dynamic_volume` significantly improves the performance of redistribution and is able to achieve 80% of the theoretical speedup on both systems (ratio of Experimental to Theoretical speedups).

8.2.3 Effect of Multiple Send Exposed Simultaneously

In `PaRSEC`, multiple sends can be exposed simultaneously by enabling the computational threads to issue direct communication for control and short data messages thereby reducing latency and improving saturation of the network. Fig. 8 evaluates the effect of `multiple_send` when varying the number of nodes. This experiment uses strong scaling, so that the total number of tasks remains constant for the different tests. The redistribution pattern in this case converts the original matrix between two distributions,



(a) Results on NaCL

(b) Results on Shaheen II

Fig. 8. Effect of `multiple_send`; higher is better.

while keeping the tile size constant. The `multiple_send` optimization is designed to reduce the runtime overhead $T_{runtime}$ when sending data, especially when dealing with small data sizes. Hence, to reproduce the case when $T_{runtime}$ is the dominant overhead due to small messages, we use a small tile size for SRC and TG, i.e., 10×10 (message size 800 Bytes). As observed in Fig. 8, there is always a positive effect of `multiple_send`, although this effect decreases as the number of nodes increases. This happens because there is less parallelism and more communication overhead when increasing the number of nodes for strong scaling.

8.3 Evaluation of Nine Categories

For a general redistribution problem, the number of possible combinations are innumerable, making it impossible to test them all; however, any scenario can be classified into one of the nine categories shown in Fig. 3. In this section, we evaluate these nine categories—which could literally cover all redistribution problems. Fig. 9 shows the performance of these nine categories on NaCL by reporting performance along with fluctuations and ratio to *bound* at different message sizes. In this experiment, the source tiles are square, i.e., $MB_s = NB_s$, and we vary the target tile sizes by

- $MB_t = MB_s * (cid/3 + 1)$,
- $NB_t = NB_s * (cid\%3 + 1)$,

where the category ID cid, $0 - 8$, represents the nine categories in Fig. 3 (0)-(8), respectively. MB_s varies from 10 (800 Bytes) to 400 (1,280,000 Bytes). D_s and D_t are two different kinds of 2DBCDDs. In Fig. 9, performance improves as message size increases, and so does the achieved ratio to *bound* in most cases, which makes sense because for small message size,

- $T_{startup}$ is the dominant overhead;
- overheads of runtime system matters ($T_{extra}/T = 0$ when calculating *bound*), since the smaller the message size, the smaller the possibility for the runtime to hide its own overheads [64];
- B_{memcpy} is set to the theoretical memory bandwidth, which may not be reached by small message size without cache reuse.

As the message size increases, it could achieve more than 80% efficiency of the theoretical *bound* for large message sizes.

8.4 Scatter and Gather Patterns

To provide a comprehensible evaluation of different redistribution patterns, we evaluate two extreme cases, scatter and gather, where all data starts or ends on a single process.

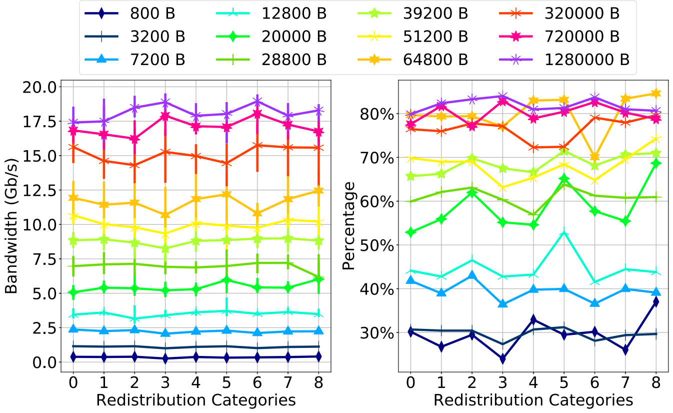


Fig. 9. Performance of the nine redistribution categories on NaCL. Left shows the bandwidth, and the corresponding ratio to *bound* is depicted on the right.

Scatter and gather are two collective communication patterns useful in many parallel algorithms, such as parallel sorting and searching, and are provided in MPI via MPI_Scatter and MPI_Gather. Scatter involves a designated root process equally distributing different chunks of data among all processes in a communicator. Gather is the inverse operation, taking elements from all processes and gathering them into a single process. It is worth noting that these two collective patterns consist of point-to-point communication in this redistribution implementation in PaRSEC.

The scatter and gather patterns described in this section are not exactly the same as their definitions in MPI, but similar. Instead of having a designated root, in this experiment, each process act as the root to distribute data onto other processes. Fig. 10 describes scatter and gather patterns deployed on four processes (a.k.a., nodes in PaRSEC). There is only one tile on each process on the SRC of the scatter pattern and on the TG of the gather pattern. With scatter, the D_s is a 1DBCDD: $1 \times num\_nodes$, and D_t follows normal 2DBCDD, and vice versa for gather. The other settings of the experiment are:

- MB_s is a multiple of MB_t in scatter pattern and MB_t a multiple of MB_s in gather pattern to ensure message size through the network is constant to get the actual *bound*;
- MB_t (NB_t) for scatter and MB_s (NB_s) for gather vary from 10 to 400 (message size from 800 to 1,280,000 Bytes).

Fig. 11 shows the performance of the redistribution implementation in PaRSEC for the scatter and gather pattern using weak and strong scaling. Theoretically, the actual bandwidth

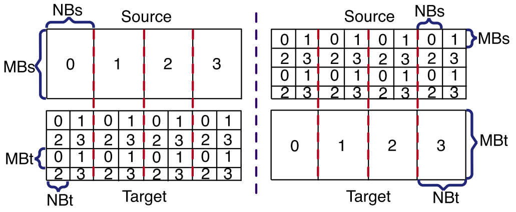
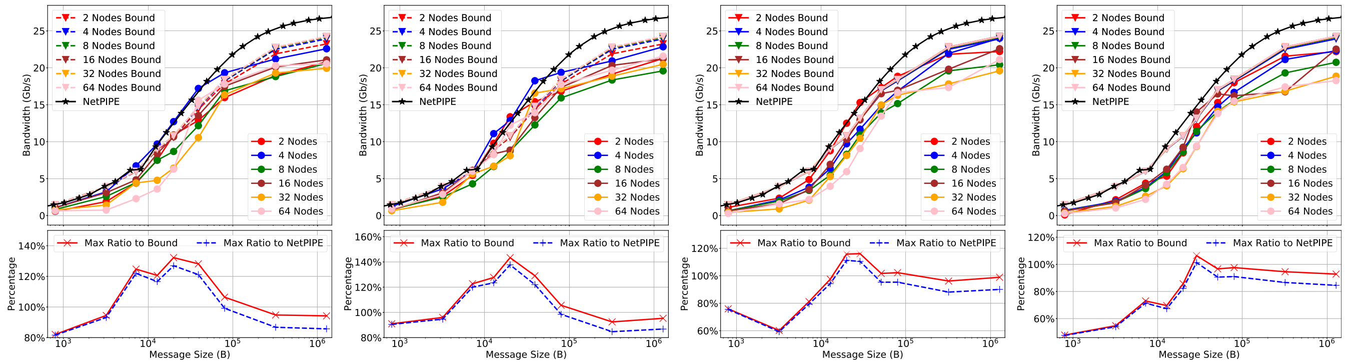


Fig. 10. Left, scatter pattern; right, gather pattern. The process ID is shown in figures; for scatter (or gather), one tile per process in SRC (or TG).



(a) Scatter, weak scaling (b) Scatter, strong scaling (c) Gather, weak scaling (d) Gather, strong scaling

Fig. 11. Scatter and gather pattern on NaCL. For each figure, upper is actual bandwidth; while maximum ratio to NetPIPE and *bound* about different message sizes is shown below.

is the same on different numbers of nodes for messages of specific sizes, ignoring all additional overheads resulting from network fluctuations, communication injections, runtime manipulations, etc. In practice, the performance presents slight variations on different numbers of nodes, as shown in Fig. 11, but it loosely follows NetPIPE/ *bound* trend for both scatter and gather.

When the message size is small, the maximal ratio (actual bandwidth over *bound*/NetPIPE) is low, around 80% for scatter and 60% for gather, because of T_{extra}/T and B_{memcpy} as analyzed in Section 8.3. It is interesting to note that the ratio could go over 100% at medium message sizes. NetPIPE may not reach the maximum point-to-point network bandwidth with one process per node with such message sizes; however, multiple_send in PaRSEC allows multiple concurrent communications to saturate the network, practically increasing messages transferred simultaneously. Another possible reason may be the optimized eager protocol in PaRSEC.

As the message size increases further, the portion of runtime overheads decreases. Therefore, for big message sizes such as 1M Bytes, our redistribution implementation could achieve about 90% of NetPIPE and *bound*. Performance of the gather is a little worse than that of the scatter. This is explained by the fact that there is only one communication thread on the receiver side to manage and receive data in PaRSEC, and control dependencies between Receive and Finish tasks exist in the gather but not in the scatter.

8.5 Performance on Randomly Distributed Matrices

Random distribution is an extreme case for irregular distributions as shown in Section 6.1, and such a feature differentiates PaRSEC from other task-based runtimes. Maintaining a high efficiency on such data distributions is a desirable capability, providing ground for better performance for low-rank and sparse algorithms. Fig. 12 depicts performances of redistribution between two random distributions (seed 2873 and 3872) and compares them to redistribution between two regular 2DBCDD distributions, by showing actual bandwidth (left y-axis, solid line) and speedup of ”2DBCDD” to ”*random* distribution” (right y-axis, dashed line). The performance of *random* distribution is worse than 2DBCDD mainly because of (1) the more optimized 2DBCDD than *random* distribution where each process in 2DBCDD only cares about its own data without retrieving the global structure; (2) the more optimized batch parameter for 2DBCDD as mentioned in Section 6.2.1. It is noteworthy to explain the bell shape of the speedup of ”2DBCDD” to ”*random* distribution”. When the message size is small, $T_{startup}$ and $T_{runtime}/T$ dominate (see Section 7.1); as the message size increase, the overheads caused by *random* distribution compared to 2DBCDD may be critical; however, when the message size continues to grow, T_{trans} becomes significant, and performances of 2DBCDD and *random* distribution are similar.

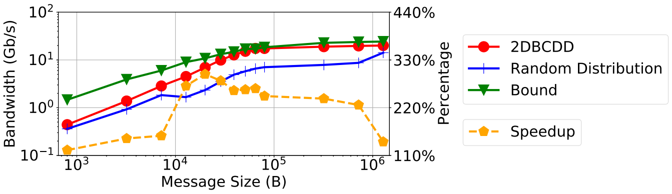
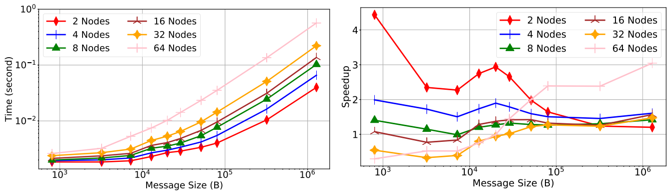


Fig. 12. Random distribution on 16 nodes NaCL. Actual bandwidth: left y-axis, solid line; speedup of ”2DBCDD” to ”*random* distribution”: right y-axis, dashed line.

8.6 Comparison to ScaLAPACK

ScaLAPACK is a high-performance library for linear algebra routines. ScaLAPACK’s data format is inherited from LAPACK [66], but it targets parallel distributed memory machines instead. It should be noted that HPF is not maintained anymore and is out-of-date, and that is also the case of all research on redistribution mentioned in the related work for HPF. In the scope of our implementation in PaRSEC, targeting dense matrices, the only implementation of redistribution that is still actively developed and widely used is ScaLAPACK. It also should be noted that ScaLAPACK only supports redistribution between regular, block-cyclic data distributions, so we restrict the scope of this evaluation to such data distributions. As mentioned in the related work, all previous research on the array or data redistribution: (1) focused on a simplified problem, i.e., the regular distributions BCDD; (2) tried to address load imbalance caused by the data distribution but ignored the impact of the data size, and we are first to (1) extend the scope of redistribution to the task-based runtime systems, (2) support irregular distributions, and (3) explicitly take data size into account. Therefore, in this



(a) Execution time of ScaLAPACK (b) Speedup to ScaLAPACK

Fig. 13. Comparison to ScaLAPACK of gather pattern on NaCL.

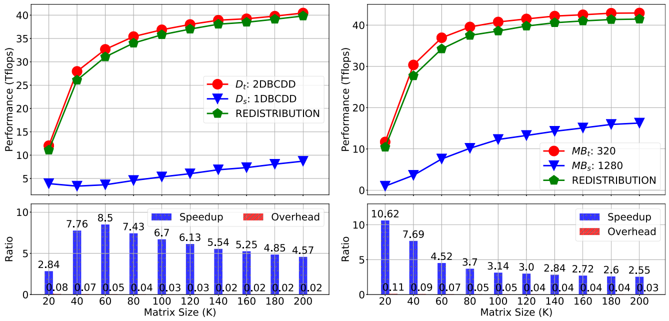
section, we only compare our implementation in PaRSEC to redistribution routines in ScaLAPACK on regular data distributions BCDD.

Fig. 13 shows a weak-scaling experiment for the gather pattern on NaCL, therefore, using the same matrix size per node for all number of nodes, to maintain a similar runtime overhead on each node. Fig. 13a presents ScaLAPACK execution time for the redistribution process, while Fig. 13b the speedup of our implementation compared to ScaLAPACK. These results show ScaLAPACK behaves better for small message sizes, especially with a larger number of nodes. This happens because $T_{runtime}$ exists in task-based runtime systems like PaRSEC, while it is not present on ScaLAPACK. The actual execution time is very small (Fig. 13b), so $T_{runtime}$ becomes increasingly dominant in the weak scaling experiment. In fact, small task granularity is not the most suitable setup for task-based runtime systems [64]. However, as the message size grows, the redistribution implementation in PaRSEC introduces a positive speedup that is almost constant for the different number of nodes (there is an unknown issue for ScaLAPACK on 64 nodes).

8.7 Evaluation in Applications

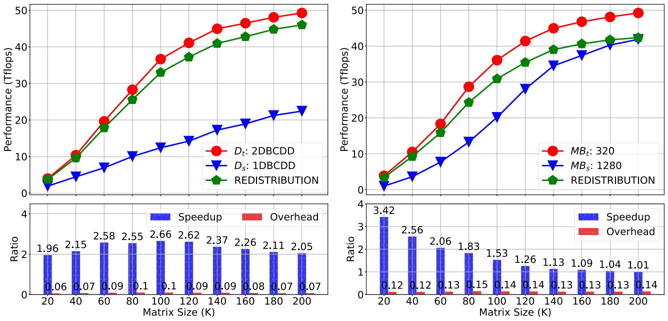
8.7.1 Redistribution Effect on Cholesky and QR Factorization

Cholesky and QR factorizations are two widely used algorithms to solve linear systems of equations ($Ax = B$). We use tiled dense Cholesky and QR factorization from DPLASMA and TLR Cholesky from Lorapo [13], both using the PaRSEC runtime system, to evaluate the benefits and overheads of redistribution with the assumption that enough memory is available to perform it. We evaluate cases where the data generator provides a distribution that is not optimal and would result in an inefficient execution, while the redistribution of the data can result in a more efficient execution. The following figures present the effects of redistribution on two different setups, converting data distribution and tile size. These optimizations may be combined in practice [13], [33].



(a) Effect of data distribution. (b) Effect of tile size conversion.

Fig. 14. QR factorization on Shaheen II (64 nodes).

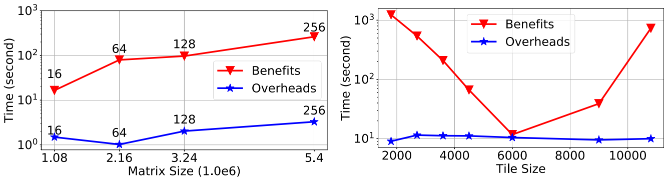


(a) Effect of data distribution. (b) Effect of tile size conversion.

Fig. 15. Cholesky factorization on Shaheen II (64 nodes).

Figs. 14a and 15a showcase the effect of data distribution maintaining the tile size ($MB_s = MB_t = 320$). Running the algorithm directly using the source matrix, a 1DBCDD with D_s: $(P, Q) = (1, 64)$, does not expose enough parallelism, and thus exhibits poor performance. On the other hand, a much more suitable distribution for this case, known theoretically but also highlighted in the figures, corresponds to 2DBCDD with D_t: $(P, Q) = (8, 8)$. Redistributing from D_s to D_t, tagged as "REDISTRIBUTION" on the graphs, redistributes from D_s to D_t, executes the QR/Cholesky factorization, and then redistributes the result matrix back to D_s, such that the entire redistribution is transparent to the caller. On the other hand, Figs. 14b and 15b illustrate redistribution when varying the tile size. In this case, the data distribution is fixed to 2DBCDD. The source matrix uses $MB_s = 1280$, which is suboptimal especially for small matrices as it reduces the parallelism and hinders performance. Similar to above, the experiment tagged as "REDISTRIBUTION" redistributes the matrix from MB_s to more appropriate $MB_t = 320$ when matrix size is small, executes the QR/Cholesky factorization, and redistributes matrix back to MB_s. For both algorithms, QR or Cholesky factorization, the "REDISTRIBUTION" can automatically convert the matrix into a more suitable data distribution or tile size, with little overheads (less than 10% in most cases), which allows the execution to unfold to be the most favorable setup on the platform and introduces relevant speedups (up to 3.42X).

TLR Cholesky factorization proposed in Lorapo contains dense tiles near the diagonal, i.e., within scope of the band size. Hence, redistribution is a very appropriate mechanism to address the load imbalance issue caused by the discrepancy of dense and low-rank tiles in TLR Cholesky factorization, redistributing these dense tiles to a more balanced workload. Fig. 16 depicts a similar experiment using TLR Cholesky factorization where both distribution [13] and tile size [33] are critical. Fig. 16a shows the impact of the data distribution while maintaining the tile size, where the D_s is 2DBCDD and D_t is less regular, a 2DBCDD distribution with a band of tiles around the diagonal in a 1DBCDD distribution (”band distribution", the benefits for such a distribution are analyzed in [33]). A kind of modified ”weak scaling" in terms of memory constrain is deployed on 16, 64, 128 and 256 nodes Shaheen II for *st-3D-sqexp* [13] with matrix size up to $5.4M \times 5.4M$. On the other hand, Fig. 16b depicts the impact of tile size changes (MB_s varies



(a) Effect of data distribution. (b) Effect of tile size conversion.

Fig. 16. TLR Cholesky on Shaheen II.

while $MB_t = 5400$) while maintaining a similar data distribution ("band distribution") for a matrix of $2.16M \times 2.16M$ elements on 16 nodes Shaheen II for *syn-2D* [13]. From these two figures, the overheads of data redistribution (the execution time of only calling data redistribution, i.e., *parsec_redistribute*) are small compared to the benefits introduced (the execution time of "$D_s - D_t$" in Fig. 16a and "$MB_s - MB_t$" in Fig. 16b). In all the testbed settings, the benefits of the redistribution are positive and introduce an improvement of several orders of magnitude in most cases. The only exception is $MB_s = 6000$ in Fig. 16b, where the benefits are reduced because MB_s is close to MB_t.

8.7.2 ScaLAPACK Format Over DPLASMA With Redistribution

The benefits on applications of this new redistribution feature in PaRSEC are not limited to the flexibility for re-mapping of data items onto processors and varying the tile size or the submatrix displacements. This extension also enables the conversion between different matrice's memory layouts or storage formats.

In recent works, a ScaLAPACK-like interface was implemented on DPLASMA to enable its usage by applications using ScaLAPACK [30]. These extensions are presented as an independent library that contains a wrapped version of the ScaLAPACK API and hides the PaRSEC API, while it constructs the structures necessary for the operation with matrices represented on ScaLAPACK memory layout. As mention earlier, ScaLAPACK's data format is inherited from LAPACK [66] in which the entire local matrix is stored in column-major order. On the other hand, in DPLASMA, which is inherited from PLASMA [31], each tile of the matrix is contiguously stored in memory in column-major order. Fig. 17 illustrates the differences between the two memory layouts. Although the wrappers enable the operation of the algorithms using the input data representation (i.e., ScaLAPACK layout), there exist situations in which DPLASMA blocking algorithms cannot be run directly. Some ScaLAPACK kernels (e.g., GEMM, TRSM) can operate on submatrices non-uniformly aligned or with different block sizes, therefore, preventing the use of DPLASMA blocking algorithms that rely on compatible blocking factors across the inputs matrices. Therefore, performing a redistribution of the data becomes key for the support of those ScaLAPACK kernels. Furthermore, redistributing the data can also improve performance, as the data distribution can be adjusted to achieve better exploitation of the available hardware resources. In this scenario, the DPLASMA wrapper first redistributes the input matrices on ScaLAPACK format to DPLASMA memory layout, runs DPLASMA kernels, and redistributes output matrices back to ScaLAPACK format.

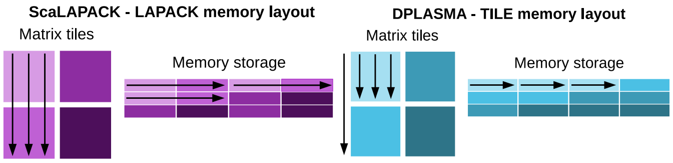
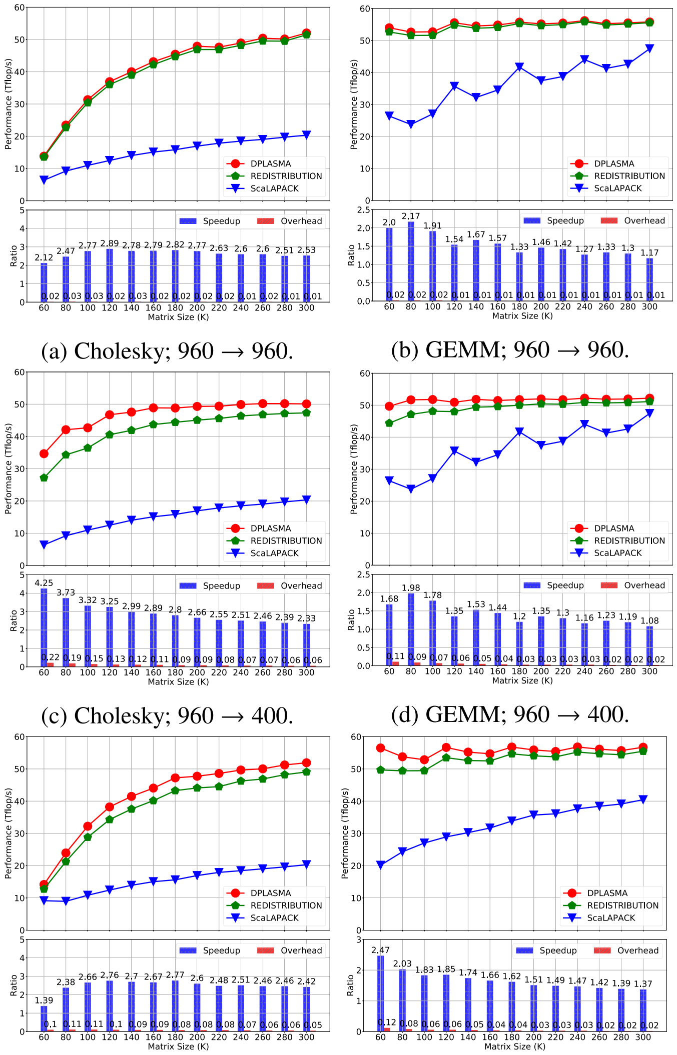

ScaLAPACK - LAPACK memory layout
Matrix tiles
Memory storage
DPLASMA - TILE memory layout
Matrix tiles
Memory storage

Fig. 17. ScaLAPACK versus DPLASMA memory layout for a matrix with four local tiles.



(a) Cholesky; 960 → 960. (b) GEMM; 960 → 960.

(c) Cholesky; 960 → 400. (d) GEMM; 960 → 400.

(e) Cholesky; 400 → 960. (f) GEMM; 400 → 960.

Fig. 18. Effect of redistribution to enable running ScaLAPACK over DPLASMA on 64 nodes Shaheen II; $a \rightarrow b$ with a tile size of ScaLAPACK format and b tile size of DPLASMA format.

Fig. 18 showcases the effect of data redistribution on DPLASMA format over ScaLAPACK format for two widely used linear algebra algorithms, dense Cholesky factorization and dense GEMM. For these experiments of Cholesky/GEMM, the data distribution is fixed to 2DBCDD. We measure different scenarios related to the tile/block size of ScaLAPACK format and DPLASMA format:

- Figs. 18a and 18b, where the tile size is the same (i.e., 960) for DPLASMA and ScaLAPACK formats;
- Figs. 18c and 18d, where the tile size of ScaLAPACK format (i.e., 960) is larger than that of DPLASMA format (i.e., 400);

- Figs. 18e and 18f, where the tile size of ScaLAPACK format (i.e., 400) is smaller than that of DPLASMA format (i.e., 960).

The graphs report the performance using the native ScaLAPACK library, the native DPLASMA algorithm, and ScaLAPACK over DPLASMA version (tagged as "REDISTRIBUTION", redistributes from ScaLAPACK format to DPLASMA format, executes the corresponding DPLASMA kernel, and redistributes back from DPLASMA format to ScaLAPACK format). For all experiments of both algorithms, the ScaLAPACK over DPLASMA version, when taking advantage of the redistribution to automatically convert the matrix format, introduce a speedup up to 4.25X with respect to the native ScaLAPACK with as little overhead as 1% over the native DPLASMA. This allows the execution to unfold to the most favorable setup on the platform and introduces important performance benefits. There is very little overhead when the block/tile size is the same in ScaLAPACK and DPLASMA format, because in this case no communication exists, and only local memory copy are needed to redistribute.

All in all, these results show that domain scientists do not have to stick anymore with predefined data distributions that impact the data generation potential, but instead, for a reasonable overhead, can allow a mismatch between data generators and users.

8.7.3 Other Applications

The data redistribution algorithm proposed in this paper could also be adopted in other applications, but it is out of this paper's scope. Here we briefly list other potential utilizations.

- *Distributed AMR.* AMR is a method in numerical analysis to adapt the accuracy of a solution within certain sensitive or turbulent regions of simulation, performed dynamically during the time the solution is being calculated, and it is applied in many scientific domains [67]. The redistribution algorithm proposed in this paper and AMR could be seamlessly connected, as domains that need to be refined in AMR could be redistributed for load balancing purposes. In addition, for its implementation in a task-based runtime system, multiple domains could be refined (redistributed) simultaneously, and PaRSEC supports this functionality as task pool topology.
- *Distributed Matrix Transpose.* Matrix Transpose is a linear algebra operator to flip a matrix over its diagonal. Research about matrix transpose on distributed memory usually depended on the assumption of 2DBCDD [68], which is not always true especially targeting complicated algorithms in sparse linear algebra, e.g., TLR Cholesky mentioned above. The data redistribution proposed in this paper is not limited by a predefined data distribution, therefore it could be an alternative to solve matrix transpose problems, because matrix transpose in its nature is a data redistribution problem. Besides, a more complicated case, i.e., submatrix transpose, could also be tackled by this data redistribution algorithm.
- *Distributed Irregular GEMM.* GEMM ($C = \alpha \times A \times B + \beta \times C$) is one of the most used BLAS routines and contributes the most performance for algorithms like Cholesky and QR factorization. In GEMM, the square matrix usually achieves higher performance than the non-square matrix because of the minimum amount of data to load [61]. Hence, in addition to the data format demonstrated in Section 8.7.2, the data redistribution algorithm in this paper could benefit two irregular GEMM, i.e., (1) GEMM with irregular tile size as in [60] and (2) GEMM of submatrix multiply, as well as the combinations of these two cases. Utilizing data redistribution, tiles of A, B, and C could automatically be converted to square tiles on the fly, so that little additional cost will be posted on the original GEMM algorithm.

9 Conclusion and Future Work

This paper presents a flexible data redistribution algorithm for the task-based runtime system, targeting a general redistribution problem and supporting any regular and irregular data distributions without constrain of data size and memory layout, which is a pioneer taking irregular data distributions and explicitly data size effect into account in task-based runtime systems. Besides the proposed cost model, we provide an implementation in a task-based runtime PaRSEC, together with a set of runtime extensions and optimizations. The practical evaluations of our implementation show it can achieve impressive performance compared with the theoretical peak and existing tools supporting some level of data redistribution, ScaLAPACK. Moreover, utilization in real applications highlights great benefits and negligible overheads in terms of data distribution, tile size, and data format with significant improvement in application time-to-solution.

For future work, we plan to explore the applicability of this redistribution algorithm to other runtime systems; while in the context of PaRSEC, we plan to further reduce communication overheads and make the redistribution a completely transparent process, an operation that could be fused with the ensuing computation to hide all overheads related to the redistribution in terms of memory and time-to-solution. We also plan to extend the scope in terms of the algorithm itself and the utilization not only on dense matrix, but also on more broadly used applications.

Acknowledgments

The authors would like to thank Aurelien Bouteiller for multi-threaded supports in PaRSEC Cray Inc. and Intel in the context of the Cray Center of Excellence and Intel Parallel Computing Center awarded to the Extreme Computing Research Center at KAUST.

References

[1] R. Garg and P. De, "Impact of noise on scaling of collectives: An empirical evaluation," *Proc. Int. Conf. High Perform. Comput.*, 2006, vol. 4297, pp. 460–471.

[2] D. Tsafrir, Y. Etsion, D. Feitelson, and S. Kirkpatrick, "System noise, OS clock ticks, and fine-grained parallel applications," in *Proc. 19th Annu. Int. Conf. Supercomput.*, New York, NY, USA: 2005, pp. 303–312.

[3] P. Beckman, K. Iskra, K. Yoshii, and S. Coghlan, “Operating system issues for petascale systems,” *SIGOPS Oper. Syst. Rev.*, vol. 40, no. 2, pp. 29–33, 2006.

[4] E. Agullo *et al.*, “Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects,” *J. Phys. Conf. Ser.*, vol. 180, 2009, Art. no. 012037.

[5] G. Bosilca *et al.*, “Flexible development of dense linear algebra algorithms on massively parallel architectures with DPLASMA,” in *Proc. IEEE Int. Parallel Distrib. Process. Symp*, 2011, pp. 1432–1441. [Online]. Available: http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6008655

[6] C. Augonnet, S. Thibault, R. Namyst, and P. Wacrenier, “StarPU: A unified platform for task scheduling on heterogeneous multicore architectures,” *Concurrency Comput. Pract. Exper.*, vol. 23, pp. 187–198, 2011.

[7] A. Duran, R. Ferrer, E. Ayguadé, R. Badia, and J. Labarta, “A proposal to extend the OpenMP tasking model with dependent tasks,” *Int. J. Parallel Program.*, vol. 37, no. 3, pp. 292–305, 2009.

[8] E. Chan, E. Quintana-Ortí, G. Quintana-Ortí, and R. van de Geijn, “Supermatrix out-of-order scheduling of matrix operations for SMP and multi-core architectures,” in *Proc. 19th Annu. ACM Symp. Parallel Algorithms Archit.*, New York, NY, USA 2007, pp. 116–125.

[9] X. Lacoste, M. Faverge, G. Bosilca, P. Ramet, and S. Thibault, “Taking advantage of hybrid systems for sparse direct solvers via task-based runtimes,” in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops*, May 2014, pp. 29–38. [Online]. Available: doi.ieeecomputersociety.org/10.1109/IPDPSW.2014.9.

[10] H. Jagode, A. Danalis, and J. Dongarra, “Accelerating NWchem coupled cluster through dataflow-based execution,” *Int. J. High Perform. Comput. Appl.*, vol. 32, no. 4, pp. 540–551, 2018. [Online]. Available: http://journals.sagepub.com/doi/10.1177/1094342016672543

[11] M. Tillenius, E. Larsson, E. Lehto, and N. Flyer, “A task parallel implementation of a scattered node stencil-based solver for the shallow water equations,” in *Proc. 6th Swedish Workshop Multi-Core Comput.*, 2013, pp. 33–36.

[12] V. Martinez, F. Dupros, M. Castro, and P. Navaux, “Performance improvement of stencil computations for multi-core architectures based on machine learning,” *Procedia Comput. Sci.*, vol. 108, no. Supplement C, pp. 305–314, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050917307408

[13] Q. Cao *et al.*, “Extreme-scale task-based cholesky factorization toward climate and weather prediction applications,” in *Proc. Platform Adv. Sci. Comput. Conf.*, 2020, pp. 1–11.

[14] D. B. Loveman, “High performance fortran,” *IEEE Parallel Distrib. Technol. Syst. Appl.*, vol. 1, no. 1, pp. 25–42, Feb. 1993.

[15] A. Wakatani and M. Wolfe, “A new approach to array redistribution: Strip mining redistribution,” in *Proc. Int. Conf. Parallel Archit. Lang. Eur.*, 1994, pp. 323–335.

[16] A. Wakatani and M. Wolfe, “Optimization of array redistribution for distributed memory multicomputers,” *Parallel Comput.*, vol. 21, no. 9, pp. 1485–1490, 1995.

[17] S. Kaushik, C.-H. Huang, J. Ramanujam, and P. Sadayappan, “Multi-phase array redistribution: Modeling and evaluation,” in *Proc. 9th Int. Parallel Process. Symp.*, 1995, pp. 441–445.

[18] R. Thakur, A. Choudhary, and J. Ramanujam, “Efficient algorithms for array redistribution,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 7, no. 6, pp. 587–594, Jun. 1996.

[19] Y. W. Lim, P. B. Bhat, and V. K. Prasanna, “Efficient algorithms for block-cyclic redistribution of arrays,” *Algorithmica*, vol. 24, no. 3–4, pp. 298–330, 1999.

[20] C.-H. Hsu, S.-W. Bai, Y.-C. Chung, and C.-S. Yang, “A generalized basic-cycle calculation method for efficient array redistribution,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 11, no. 12, pp. 1201–1216, Dec. 2000.

[21] T. Marrinan, J. A. Insley, S. Rizzi, F. Tessier, and M. E. Papka, “Automated dynamic data redistribution,” in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops*, 2017, pp. 1208–1215.

[22] C. Foyer, A. Tate, and S. McIntosh-Smith, “Aspen: An efficient algorithm for data redistribution between producer and consumer grids,” in *Proc. Eur. Conf. Parallel Process*, 2018, pp. 171–182.

[23] D. W. Walker and S. W. Otto, “Redistribution of block-cyclic data distributions using MPI,” *Concurrency: Pract. Experience*, vol. 8, no. 9, pp. 707–728, 1996.

[24] A. Reisner, L. N. Olson, and J. D. Moulton, “Scaling structured multigrid to 500k+ cores through coarse-grid redistribution,” *SIAM J. Sci. Comput.*, vol. 40, no. 4, pp. C581–C604, 2018.

[25] M. Hofmann and G. Rünger, “Fine-grained data distribution operations for particle codes,” in *Proc. Eur. Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, 2009, pp. 54–63.

[26] M. Hofmann and G. Rünger, “Flexible all-to-all data redistribution methods for grid-based particle codes,” *Concurrency Comput. Pract. Experience*, vol. 30, no. 13, 2018, Art. no. e4421.

[27] J. Dongarra, L. Prylli, C. Randriamaro, and B. Tourancheau, “Array redistribution in scalapack using PVM,” *Proc. EuroPVM*, 1995, vol. 95, pp. 271–276.

[28] L. Prylli and B. Tourancheau, “Efficient block cyclic data redistribution,” in *Proc. Eur. Conf. Parallel Process*, 1996, pp. 155–164.

[29] M. Hofmann and G. Rünger, “Efficient data redistribution methods for coupled parallel particle codes,” in *Proc. 42nd Int. Conf. Parallel Process*, 2013, pp. 40–49.

[30] L. Blackford *et al.*, *ScaLAPACK Users' Guide*. Philadelphia, PA, USA: Soc. Ind. Appl. Math., 1997.

[31] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, “A class of parallel tiled linear algebra algorithms for multicore architectures,” *Parallel Comput.*, vol. 35, no. 1, pp. 38–53, 2009.

[32] J. Kurzak, H. Anzt, M. Gates, and J. Dongarra, “Implementation and tuning of batched Cholesky factorization and solve for NVIDIA GPUs,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 7, pp. 2036–2048, Jul. 2016.

[33] Q. Cao *et al.*, “Performance analysis of tile low-rank cholesky factorization using parsec instrumentation tools,” in *Proc. IEEE/ACM Int. Workshop Program. Perform. Vis. Tools*, 2019, pp. 25–32.

[34] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. Dongarra, “PaRSEC: A programming paradigm exploiting heterogeneity for enhancing scalability,” *Comput. Sci. Eng.*, vol. 15, no. 6, pp. 36–45, Nov./Dec. 2013.

[35] Q. Cao, G. Bosilca, W. Wu, D. Zhong, A. Bouteiller, and J. Dongarra, “Flexible data redistribution in a task-based runtime system,” in *Proc. IEEE Int. Conf. Cluster Comput.*, 2020, pp. 221–225.

[36] A. YarKhan, “Dynamic task execution on shared and distributed memory architectures,” Ph.D. dissertation, Dept. Comput. Sci., Univ. Tennessee, Knoxville, Tennessee, 2012. [Online]. Available: http://trace.tennessee.edu/utk_graddiss/1575

[37] A. YarKhan, J. Kurzak, and J. Dongarra, “QUARK Users' guide: QUeueing and runtime for kernels,” Innov Comput. Lab., Univ. Tennessee, Knoxville, Tennessee, Tech. Rep. ICL-UT-11–02, 2011.

[38] R. Hoque, T. Herault, G. Bosilca, and J. Dongarra, “Dynamic task discovery in PaRSEC: A data-flow task-based runtime,” in *Proc. 8th Workshop Latest Adv. Scalable Algorithms Large-Scale Syst.*, 2017, pp. 6:1–6:8.

[39] E. Agullo *et al.*, “Achieving high performance on supercomputers with a sequential task-based programming model,” *IEEE Trans. Parallel Distrib. Syst.*, to be published, doi: 10.1109/TPDS.2017.2766064.

[40] R. Lopes, S. Thibault, and A. Melo, “MASA-StarPU: Parallel sequence comparison with multiple scheduling policies and pruning,” in *Proc. SBAC-PAD IEEE 32nd Int. Symp. Comput. Archit. High Perform. Comput.*, 2020, pp. 225–232.

[41] F. Lordan *et al.*, “Servicess: An interoperable programming framework for the cloud,” *J. Grid Comput.*, vol. 12, no. 1, pp. 67–91, 2014.

[42] OpenMP, “OpenMP 4.0 complete specifications,” 2013. [Online]. Available: http://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf

[43] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, “Legion: Expressing locality and independence with logical regions,” in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2012, pp. 1–11.

[44] S. Treichler, “Realm: Performance portability through composable asynchrony,” Ph.D. dissertation, Dept. Comput. Sci., Stanford Univ., Stanford, CA, USA, 2014.

[45] T. Heller, H. Kaiser, and K. Iglberger, “Application of the parallex execution model to stencil-based problems,” *Comput. Sci. - Res. Develop.*, vol. 28, no. 2–3, pp. 253–261, 2013.

[46] J. M. Perez, R. M. Badia, and J. Labarta, “A dependency-aware task-based programming environment for multi-core architectures,” in *Proc. IEEE Int. Conf. Cluster Comput.*, 2008, pp. 142–151.

[47] J. Dokulil, M. Sandrieser, and S. Benkner, “Implementing the open community runtime for shared-memory and distributed-memory systems,” *24th Euromicro Int. Conf. Parallel, Distrib., Netw.-Based Process.*, 2016, pp. 364–368.

[48] A. P. Petitet and J. J. Dongarra, “Algorithmic redistribution methods for block-cyclic decompositions,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 10, no. 12, pp. 1201–1216, Dec. 1999.

[49] M. Guo and I. Nakata, "A framework for efficient data redistribution on distributed memory multicomputers," *J. Supercomput.*, vol. 20, no. 3, pp. 243–265, 2001.

[50] R. Sudarsan and C. J. Ribbens, "Efficient multidimensional data redistribution for resizable parallel computations," in *Proc. Int. Symp. Parallel Distrib. Process. Appl*, 2007, pp. 182–194.

[51] J. Herrmann, G. Bosilca, T. Hérault, L. Marchal, Y. Robert, and J. Dongarra, "Assessing the cost of redistribution followed by a computational kernel: Complexity and performance results," *Parallel Comput.*, vol. 52, no. C, pp. 22–41, 2016.

[52] S. Iserte, R. Mayo, E. S. Quintana-Orti , and A. J. Peña, "DMRlib: Easy-coding and efficient resource management for job malleability," *IEEE Trans. Comput.*, vol. 70, no. 9, pp. 1443–1457, 1 Sep. 2021.

[53] T. Schneider, F. Kjolstad, and T. Hoefler, "MPI datatype processing using runtime compilation," in *Proc. 20th Eur. MPI Users' Group Meeting*, 2013, pp. 19–24.

[54] T. Prabhu and W. Gropp, "Dame: A runtime-compiled engine for derived datatypes," in *Proc. 22nd Eur. MPI Users' Group Meeting*, 2015, Art. no. 4.

[55] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. Dongarra, "PaRSEC: Exploiting heterogeneity to enhance scalability," *Comput. Sci. Eng.*, vol. 15, no. 6, pp. 36–45, Nov. 2013.

[56] A. Danalis, G. Bosilca, A. Bouteiller, T. Herault, and J. Dongarra, "PTG: An abstraction for unhindered parallelism," in *Proc. 4th Int. Workshop Domain-Specific Lang. High-Level Frameworks High Perform. Comput.*, 2014, pp. 21–30.

[57] A. Danalis, H. Jagode, G. Bosilca, and J. Dongarra, "PaRSEC in practice: Optimizing a legacy chemistry application through distributed task-based execution," in *Proc. IEEE Int. Conf. Cluster .*, Sep. 2015, pp. 304–313.

[58] H. Jagode, A. Danalis, G. Bosilca, and J. Dongarra, *Accelerating NWChem Coupled Cluster Through Dataflow-Based Execution*. Berlin, Germany: Springer , 2016, pp. 366–376.

[59] Q. Cao *et al.*, "Leveraging parsec runtime support to tackle challenging 3D data-sparse matrix problems," 2020. [Online]. Available: http://hdl.handle.net/10754/665738

[60] T. Herault, Y. Robert, G. Bosilca, and J. Dongarra, "Generic matrix multiplication for multi-GPU accelerated distributed-memory platforms over parsec," in *Proc. IEEE/ACM 10th Workshop Latest Adv. Scalable Algorithms Large-Scale Syst.*, 2019, pp. 33–41.

[61] T. Herault *et al.*, "Distributed-memory multi-GPU block-sparse tensor contraction for electronic structure," Inria, Rocquencourt, France, Res. Rep. RR-9365, Oct. 2020.

[62] T. Patinyasakdikul, D. Eberius, G. Bosilca, and N. Hjelm, "Give MPI threading a fair chance: A study of multithreaded MPI designs," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2019, pp. 1–11.

[63] C.-L. Wang, P. B. Bhat, and V. K. Prasanna, "High-performance computing for vision," *Proc. IEEE*, vol. 84, no. 7, pp. 931–946, Jul. 1996.

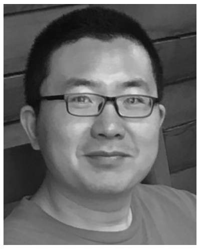
[64] E. Slaughter *et al.*, "Task bench: A parameterized benchmark for evaluating parallel runtime performance," *The Int. Conf. High Perform. Comput., Netw., Storage, Anal.*, 2020. [Online]. Available: http://arxiv.org/abs/1908.05790

[65] Q. O. Snell, A. R. Mikler, and J. L. Gustafson, "NetPIPE: A network protocol independent performance evaluator," *Proc. Int. Conf. Intell. Inform. Manage. Syst.*, 1996, vol. 6, Art. no. 49.

[66] E. Anderson *et al.*, *LAPACK Users' Guide*. Philadelphia, PA, USA: Siam, 1999, vol. 9.

[67] Wikipedia, "Adaptive mesh refinement," Accessed: Jan. 18, 2019. [Online]. Available: https://en.wikipedia.org/wiki/Adaptive_mesh_refinement

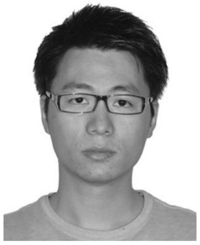
[68] J. Choi, J. J. Dongarra, and D. W. Walker, "Parallel matrix transpose algorithms on distributed memory concurrent computers," *Parallel Comput.*, vol. 21, no. 9, pp. 1387–1405, 1995.



Qinglei Cao received the BS degree in information and computational science from Hunan University and the MS degree in computer application technology from the Ocean University of China. He is currently working toward the PhD degree with the Innovative Computing Laboratory, University of Tennessee, Knoxville. He was a software engineer with the National University of Defense Technology for more than three years. His research interests include distributed or parallel computing, task-based runtime system and linear algebra, including PaRSEC, DPLASMA, HiCMA, and Open MPI.

George Bosilca is currently the research director and an adjunct assistant professor with the Innovative Computing Laboratory, University of Tennessee, Knoxville. His research interests include concepts of distributed algorithms, parallel programming paradigms, and software resilience, from both a theoretical and practical perspective.

Nuria Losada received the BS, MS, and PhD degrees in computer science from the Universidade da Coruña, Spain, in 2013, 2014, and 2018, respectively. She is currently a postdoctoral researcher with the Innovative Computing Laboratory, University of Tennessee, Knoxville. Her research interests include fault-tolerance, distributed computing, and parallel programming paradigms.



Wei Wu received the BE degree in software engineering from the Beijing Institute of Technology, the MS degree in computer engineering from Purdue University, and the PhD degree in computer science in 2017 from the University of Tennessee, Knoxville. He is currently a research scientist with Los Alamos National Laboratory. His research interests include runtime systems and programming models in high performance computing.

Dong Zhong received the BS degree in computer science from Tongji University and the MS degree in information science and electronic engineering from the Zhejiang University of China. He is currently working toward the PhD degree with the Innovative Computing Laboratory, University of Tennessee, Knoxville. His research interests include distributed computing, parallel programming paradigms including Open MPI, PMIx and PRRTE, failure detection and notification, and long vector extension analysis and usage of Arm SVE and Intel AVXs.

Jack Dongarra (Fellow, IEEE) holds an appointment with the University of Tennessee, Oak Ridge National Laboratory, and the University of Manchester. He specializes in numerical algorithms in linear algebra, parallel computing, use of advanced-computer architectures, programming methodology, and tools for parallel computers. He was the recipient of IEEE Sid Fernbach Award in 2004, first IEEE Medal of Excellence in Scalable Computing in 2008, SIAM Special Interest Group on Supercomputing's Award for Career Achievement in 2010, IEEE Charles Babbage Award in 2011, and the ACM/IEEE Ken Kennedy Award in 2013. He is a fellow of the AAAS, ACM, and SIAM and a foreign member of the Russian Academy of Science and a member of the U.S. National Academy of Engineering.