摘要(本文贡献)
1.使用动态数据流/数据驱动技术提升高性能计算(HPC)系统的性能
2.提出的方法通过一个有效的、可插拔的(轻便的)、鲁棒的编程框架，开启HPC的数据驱动并发
3.提出的框架基于多线程数据驱动(DDM)、混合控制流/数据流模型，该模型可以基于线性处理器的数据可用性来编排线程
4.提出的框架使用两个不同系统上有不同特性的多个基准(基准检查程序)评估
5.评估结果：提出的框架具有良好的伸缩性，能够有效容忍调度开销和内存延迟。
6.与MPI、DDM-VM、 OmpSs@Cluster对比，本框架获得相当的或者更好地性能。

Data-Driven Concurrency for High Performance Computing

GEORGE MATHEOU and PARASKEVAS EVRIPIDOU, University of Cyprus

In this work, we utilize dynamic dataflow/data-driven techniques to improve the performance of high performance computing (HPC) systems. The proposed techniques are implemented and evaluated through an efficient, portable, and robust programming framework that enables data-driven concurrency on HPC systems. The proposed framework is based on data-driven multithreading (DDM), a hybrid control-flow/dataflow model that schedules threads based on data availability on sequential processors. The proposed framework was evaluated using several benchmarks, with different characteristics, on two different systems: a 4-node AMD system with a total of 128 cores and a 64-node Intel HPC system with a total of 768 cores. The performance evaluation shows that the proposed framework scales well and tolerates scheduling overheads and memory latencies effectively. We also compare our framework to MPI, DDM-VM, and OmpSs@Cluster. The comparison results show that the proposed framework obtains comparable or better performance.

CCS Concepts: • **Computer systems organization** → **Data flow architectures**; **Multicore architectures**; **Distributed architectures**; • **Computing methodologies** → **Parallel programming languages**; **Distributed programming languages**;

- 计算系统组织

- 数据流框架
- 多核框架
- 分布式框架
- 计算方法论

-并行编程语言
分布式编程语言

Additional Key Words and Phrases: Data-driven multithreading, distributed execution, runtime system, high performance computing

数据驱动多线程 分布式执行 运行时系统

ACM Reference format:
George Matheou and Paraskevas Evripidou. 2017. Data-Driven Concurrency for High Performance Computing. *ACM Trans. Archit. Code Optim.* 14, 4, Article 53 (December 2017), 26 pages.
https://doi.org/10.1145/3162014

1 INTRODUCTION

The end of the exponential growth of the sequential processors has facilitated the development of multicore and manycore architectures. Such architectures have dominated the high performance computing (HPC) field, from shared memory systems to large-scale distributed memory clusters. Thus, any growth in performance of HPC systems must come from *parallelism/concurrency* [25]. To achieve that, efficient parallel programming/execution models must be developed [5]. We propose to develop such systems using the data-driven multithreading (DDM) model of execution. DDM [37] is a hybrid control-flow/dataflow model that schedules threads (called *DThreads*) based on data availability on conventional processors. A DThread is scheduled for execution after all of its required data have been produced; thus, no synchronization or communication latencies are experienced after a DThread begins its execution. DDM was efficiently implemented in both software and hardware, achieving very good results [7, 8, 43–45, 48].

1.本文提出的框架是执行的数据驱动多线程模型 (DDM)
2.DDM是在传统处理器上基于数据可用性来完成编排线程(DThreads)的混合控制流/数据流

在Dthread需要的数据全都提供后，需要执行的Dthread才会被编排；因此Dthread开始执行后不会有同步问题或者通信延迟问题

This work was partially funded by the Cyprus State Scholarship Foundation (IKYK) through a scholarship for G. Matheou and by the University of Cyprus through the XProcessor project.
Authors' addresses: G. Matheou and P. Evripidou; emails: {geomat, skevos}@cs.ucy.ac.cy.

基于数据流/数据驱动的执行的系统，如DDM系统的优势：
1.细粒度任务/线程异步数据驱动执行，或者细粒度变成模型能够有效的利用底层硬件
2.由于数据流模型只强制执行真正的数据依赖关系，可以公开程序的最大并行度
3.可以有效处理并发并容忍内存和同步时延

Systems based on dynamic dataflow/data-driven execution [28], such as DDM, have several advantages over the sequential model of execution: (i) allow asynchronous data-driven execution of fine-grain tasks/threads, and fine-grain programming models have a great potential to efficiently use the underlying hardware [5, 33, 39, 61]; (ii) can expose the maximum degree of parallelism in a program since the dataflow model only enforces true data dependencies [31]; and (iii) can handle concurrency and tolerate memory and synchronization latencies efficiently [10]. Thus, systems based on dynamic dataflow can be used to efficiently exploit the computing power of current and future HPC systems [22, 33, 39, 40, 61].

本工作扩展DDM的功能来启动HPC系统上有效且轻便的分布式数据驱动并发

In this work we extend the functionalities of DDM to enable efficient and portable distributed data-driven concurrency on HPC systems. The proposed functionalities are implemented in the FREDDO system [45], an efficient C++ implementation of DDM that until recently was supporting data-driven execution on single-node multicore systems. In distributed DDM applications, remote memory accesses are introduced, resulting from producer and consumer DThreads running on different nodes. The distributed FREDDO implementation provides implicit *data forwarding* [36] to the node where the consumer DThread is scheduled to run. In particular, a consumer DThread can be scheduled for execution only when all of its input data are available in the main memory. This helps to reduce memory latencies [36]. FREDDO is publicly available for download in [42].

Distributed FREDDO provides implicit data forwarding through a distributed shared memory (DSM) implementation [54] with a shared global address space (GAS). Coherence operations implemented in typical DSM systems [53] are not required between the nodes because the produced data is forwarded to consumers that will be executed on remote nodes. DSM eases the development of distributed FREDDO/DDM applications that use shared objects/data-structures (scalar values, arrays, etc.). The programmer only needs to register the shared objects of single-node FREDDO applications in GAS and specify the output data of each DThread. For algorithms without shared objects, such as recursive algorithms, FREDDO allows data forwarding through *data objects* that are exchanged between the nodes of the system.

DSM系统上节点间不需要进行一致性操作，因为生辰的数据会传递给消费者并在消费者节点上执行。

The proposed framework allows efficient distributed data-driven execution, where scheduling operations, computations, and network functionalities are operated asynchronously. It distributes DThread instances on the system's nodes using a lightweight *distribution scheme* based on DDM's tagging system. DDM's tagging system was based on the U-Interpreter model [9]. FREDDO's memory model in combination with its distribution scheme allow complicated tasks, such as partitioning of data and computations, transferring of data during execution and preserving coherency, to be provided automatically. This article makes the following contributions:

(1) Provide an efficient, portable, and robust distributed implementation of the DDM model. All required data-structures and mechanisms were designed and implemented to support efficient distributed data-driven concurrency under the FREDDO framework.
(2) Provide distributed recursion support for the DDM model.
(3) The evaluation of the DDM model on two different systems: a 4-node AMD system with a total of 128 cores and an open-access 64-node Intel HPC system with a total of 768 cores. The DDM model was previously evaluated only on very small distributed multicore systems with up to 24 cores using the DDM-VM implementation [48].
(4) The comparison of the results obtained from FREDDO and other frameworks such as MPI [26], DDM-VM [48], and OmpSs@Cluster [16, 17]. The comparison results show that FREDDO achieves similar or better performance. Furthermore, to the best of our knowledge, this is the first work that compares a data-driven system with static dependency

resolution (FREDDO/DDM) against a data-driven system with dynamic dependency resolution (OmpSs) on distributed multicore architectures.

(5) Provide simple mechanisms/optimizations to reduce the network traffic of distributed DDM applications. Our experiments on the AMD system show that FREDDO can reduce the total amount of TCP packets by up to 6.55× and the total amount of data by up to 16.7% when compared to the DDM-VM system.

(6) The implementation of a connectivity layer with two different network interfaces: a custom network interface (CNI) and MPI. The CNI support allows a direct and fair comparison with frameworks that also utilize a CNI (e.g., DDM-VM) where MPI support provides portability and flexibility to the FREDDO framework. Finally, we provide comparison results for the CNI and MPI for several benchmarks.

The remainder of this article is organized as follows. Section 2 describes the DDM model and the current single-node FREDDO implementation. Section 3 presents the distributed FREDDO implementation. An example of a FREDDO programming example is presented in Section 4. The experimental results and related work are presented in Sections 5 and 6, respectively. Finally, Section 7 concludes this work.

2 DATA-DRIVEN MULTITHREADING

DDM是一个非阻塞的多线程模型，分解程序的同步执行模块，允许他们异步执行，因此能够有效容忍同步和通信时延。

DDM [37] is a nonblocking multithreading model that decouples the execution from the synchronization part of a program and allows them to execute asynchronously, thus tolerating synchronization and communication latencies efficiently. In DDM, a program consists of several threads of instructions (called *DThreads*) that have producer-consumer relationships. A DThread is scheduled for execution in a data-driven manner—that is, after all of its required data have been produced. DDM's tagging system is based on the U-Interpreter model [9], and it enables multiple instances of the same DThread to coexist in the system. Each DThread instance is identified uniquely by the tuple: Thread ID (TID) and Context. Re-entrant constructs, such as loops and function calls, can be parallelized by mapping them into DThreads. For example, each iteration of a parallel loop can be executed by an instance of a DThread. The core of the DDM model is the thread scheduling unit (TSU), which is responsible for scheduling DThreads at runtime based on data availability. For each DThread, the TSU collects metadata (also called *thread templates*) that enable the management of the dependencies among the DThreads and determine when a DThread instance can be scheduled for execution. The TSU schedules a DThread instance for execution when all of its producer instances have completed their execution.

DDM中，程序由多个指令(DThreads)的线程组成，只能具有生产者-消费者的关系。

Dthread实例由tuple区分：TID+Context

DDM模型的核心是线程编排单元，其负责基于数据可用性编排运行时Dthreads

对于每一个Dthread，TSU收集元数据metadata(线程模板 thread templates)--可以启动Dthreads之间依赖的管理并决定执行的Dthread实例什么时候被编排。
TSU在执行过程的所有生产者实例全部完成后才编排DThread实例。

2.1 Single-Node FREDDO Implementation

FREDDO [45, 46] is an efficient object-oriented implementation of the DDM model [37]. It is a C++11 framework that supports data-driven execution on conventional multicore architectures. FREDDO provides a C++ application programming interface (API) that includes a set of runtime functions and classes that help programmers develop DDM applications. The API provides basic functionalities for parallelizing loops, recursive functions, and simple function calls with data dependencies. In FREDDO, a program consists of DThreads that are implemented as C++ objects. FREDDO allows efficient DDM execution by utilizing three different components: an optimized C++ implementation of the TSU, the kernels, and the runtime support. A kernel is a POSIX thread (PThread) that is responsible for executing ready DThread instances that are received from the TSU. The runtime system enables communication between kernels and the TSU through the main memory. It is also responsible for loading the thread templates in the TSU, for creating and running the kernels, and for deallocating the resources allocated by DDM programs.

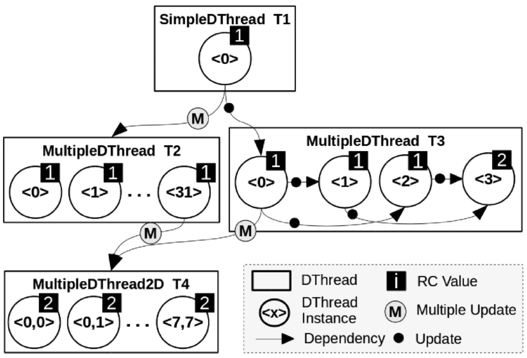


Fig. 1. Example of a dependency graph.

2.2 DDM/FREDDO Dependency Graph

FREDDO has the same dependency graph as any other DDM implementation. The only difference is that DThreads are implemented as C++ objects. The dependency graph is a directed graph in which nodes represent the DThread instances and arcs represent data dependencies among them. Each instance of a DThread is paired with a special value called *Ready Count* (RC) that represents the number of its producers. An example of a dependency graph is shown in Figure 1, which consists of four DThreads (T1 through T4). The number inside each node indicates its Context value. T1 is a SimpleDThread object—that is, it has only one instance. Multiple DThread objects are introduced to manage DThreads with multiple instances. Such objects can be used to parallelize one-level loops (with MultipleDThread), two-level nested loops (with MultipleDThread2D), and three-level nested loops (with MultipleDThread3D). T2 and T3 are MultipleDThread objects where T4 is a MultipleDThread2D object. In the latter case, the Context values consist of two parts: the outer and the inner. The RC value is initiated statically and is dynamically decremented by the TSU each time a producer completes its execution. A DThread's instance is deemed executable when its RC value reaches zero. In DDM, the operation used for decreasing the RC value is called *Update*. Update operations can be considered as tokens that are moving from producer to consumer instances through arcs of the graph. *Multiple Updates* are introduced to decrease multiple RC values of a DThread at the same time. This reduces the number of tokens in the graph. For instance, T1 sends a Multiple Update command to T2 to spawn all of its instances instead of sending 32 single Updates.

FREDDO的分布式框架介绍：
1）内存模型
2）编排和终止机制
3）网络支持
4）系统中降低网络拥塞的技术
5）分布式递归函数支持

3 FREDDO: DISTRIBUTED SUPPORT

In this section, we describe FREDDO's distributed architecture and its memory model, scheduling and termination mechanisms, network support, and techniques used for reducing network traffic in the system. We also briefly describe the distributed recursion support.

3.1 Distributed Architecture

FERDDO由全部网络内部进行连接的多核节点组成，每个节点上都有网络管理器Network Manager的实现

The distributed architecture of FREDDO is depicted in Figure 2. It is composed by multicore nodes connected by a global network interconnect (Ethernet, InfiniBand, etc.). A *Network Manager* is implemented in each node, which abstracts the details of the network interconnect and allows internode communication. FREDDO's runtime system is responsible for (i) handling communication and data management across nodes, (ii) managing the applications' dependency graphs, and (iii) scheduling/executing ready DThread instances on the cores of the entire system. The same

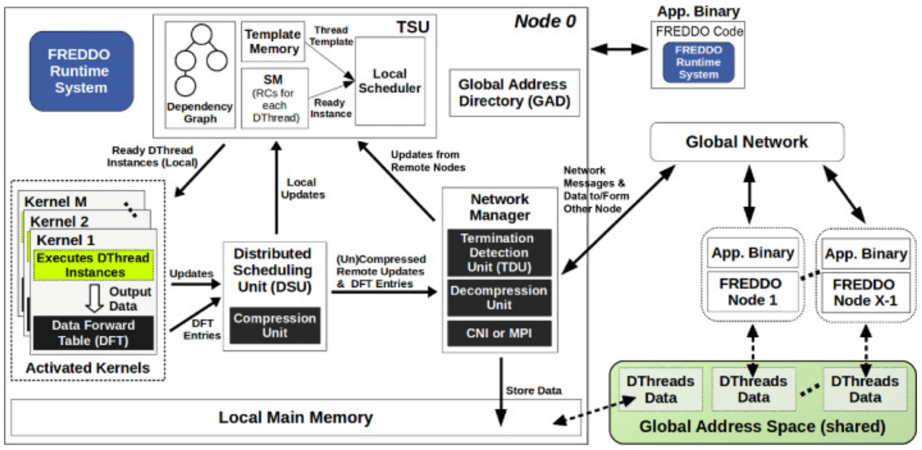


Fig. 2. FREDDO's distributed architecture.

application binary is executed on all nodes, where one of them is selected as the *RootNode*. The *RootNode* is responsible for detecting the termination of the distributed FREDDO applications and for gathering the results for validation purposes (this is optional).

3.2 Memory Model

FREDDO implements a software DSM system [54] with a shared GAS support. Part, or all, of the main memory space on each node is mapped to the DSM's GAS. This approach creates an identical address space on each node, which gives the view of a single distributed address space. The conventional main memory addresses of shared objects (scalar values, vectors, arrays, etc.) are registered in the GAS by storing them in the global address directory (GAD) of each node. For each such address, the runtime assigns to it a unique identifier (called *GAS_ID*), which is identical in each node. This allows the runtime system to transfer data between nodes (from one local main memory to another) using GAS_IDs.

FREDDO uses the DSM implementation to employ implicit *data forwarding* [36]. The produced/output data of a DThread instance is *forwarded* to its consumers, running on remote nodes, before the latter start their execution. This is guaranteed by sending the Update operations after data transfers are completed. To support this functionality, each kernel is associated with a data forward table (DFT). A DFT keeps track of output data segments of the currently executed DThread instance. The DFT allocates a separate entry for each output data segment. When a DThread instance finishes its execution, the DFT entries related to the DThread instance are removed from the associated DFT.

A DFT entry consists of the following attributes: GAS_ID, *addrOffset* (the offset in bytes from the conventional address that maps to the GAS_ID), *segmentSize* (the size in bytes of the data segment), and *sentToTable* (marks the nodes that have received the data segment). *sentToTable* is used to send a data segment to a node only once. The runtime uses DFT entries to transfer produced data to remote nodes, implicitly. Algorithm 1 depicts the basic algorithm of sending Updates and output data of a DThread instance to remote nodes. Expensive coherence operations implemented in typical DSM systems [53] are not required. The remote read operations are eliminated; thus, the total communication cost can be reduced. Coherence operations are applied only within each node's memory hierarchy, by hardware, since each node is a conventional multicore processor. It

任务同时写相同的数据，并且没有特定依赖，会导致未定义的行为

is important to note that applications where tasks write to the same data simultaneously, without specifying dependencies, result in an undefined behavior.

ALGORITHM 1: Sending Updates and output data of a DThread instance *DI* that is executed on kernel *k*

```
foreach $Upd \in Updates\ of\ DI\ code$ do
    i ← the id of the node that will execute Upd
    if $i = local\ node\ id$ then
        Send Upd to local TSU
    else
        foreach $dftEntry \in DFT\ of\ kernel\ k$ do
            if $dftEntry.sentToTable[i] = false$ then
                sendData(i, dftEntry.GAS_ID, dftEntry.addrOffset, dftEntry.segmentSize)
                dftEntry.sentToTable[i] ← true
        Send Upd to the node with id = i
```

DSM eases the development of distributed FREDDO/DDM applications that use shared objects/data-structures. The programmer only needs to register shared objects of single-node FREDDO applications in GAS and specify the output data of each DThread, using special runtime functions (see Section 4.2). For algorithms without shared objects, FREDDO allows data forwarding through data objects that are exchanged between the nodes. As a proof of concept, we have used this approach for FREDDO's distributed recursion support, where data objects (called *DistRData* objects) are used to transfer the arguments and return values of recursive function calls. In this case, the DSM implementation and DFTs are not used. FREDDO sends Update operations to remote nodes after transfers of the data objects are completed.

3.3 Distribution Scheme and Scheduling Mechanisms for DThread Instances

FREDDO provides a lightweight distribution scheme, based on DDM's tagging system [37], to distribute DThread instances on the system's nodes. In particular, FREDDO implements a *static scheme* in which the mapping of DThread instances to nodes is determined at compile time, based on their Context values (tags). The node in which a DThread instance will be executed is defined by the following formula: $node\ id = f_{cn}(Cntx\ \%\ totNumCores)$, where $Cntx$ is the Context value of the DThread instance and $totNumCores$ is the total number of cores of the entire system. f_{cn} returns the node ID of a core (e.g., in a 4-node system with four cores per node, $f_{cn}(0) = 0$ and $f_{cn}(15) = 3$). The static scheme only specifies where a DThread instance will be scheduled for execution. However, DThread instances are scheduled for execution at runtime, based on data availability (DThread instances are dynamically created). This approach simplifies the scheduling and data management operations. In addition, it reduces runtime overheads.

FREDDO utilizes two different scheduling mechanisms for executing DThread instances: intranode and internode. The *internode* mechanism is provided by the distributed scheduling unit (DSU), which decides, based on FREDDO's distribution scheme, if Update operations and the output data of a producer-DThread instance will be forwarded to a remote node (or nodes) or in the local node. In the former case, the runtime will send Update operations and the output data as network messages, via the Network Manager, to the corresponding remote node(s). In the latter case, Update operations are sent to the local TSU.

The *intranode* mechanism is handled by the TSU, which is responsible for storing the dependency graph of the applications, the thread templates (in the template memory), and the RC values of the DThread instances (in the synchronization memory (SM)). The TSU fetches local Updates from the kernels, through the DSU, or remote Updates from the Network Manager. For each Update (TID + Context), the TSU locates the thread template of the DThread from the template memory (TM) and decrements the RC value in the SM. If the RC value of any DThread's instance reaches zero, then it is deemed executable and is sent to the TSU's local scheduler. The local scheduler distributes the ready DThread instances to the kernels to achieve load balancing (it selects the kernel with the least amount of work). Further details about the TSU's functionalities and its architecture can be found in Matheou and Evripidou [45, 46].

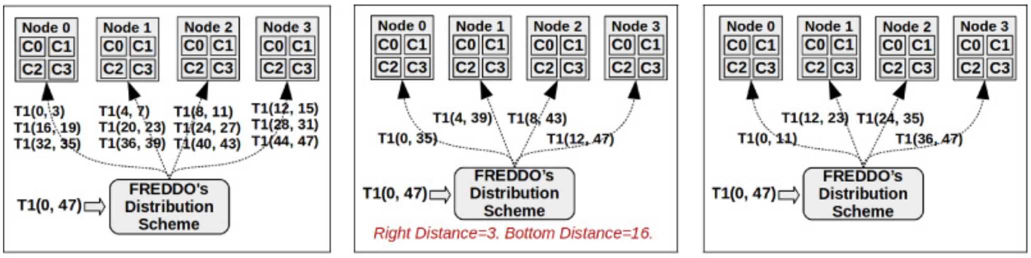
3.4 Distributed Execution Termination

Detecting termination of data-driven programs in distributed execution environments is not a straightforward procedure, as the availability of data governs the order of execution. In this work, we have implemented an implicit distributed termination algorithm based on Dijkstra and Scholten's parental responsibility algorithm [23], which requires minimal message exchange. The algorithm assumes termination when: the state of all nodes is passive (idle) and no messages are on their way in the system. In our implementation, the state *passive* refers to the state when the TSU has no pending Update operations or pending ready DThread instances waiting for execution. When the parent node (*RootNode*) detects termination, it broadcasts a termination message to the other nodes and waits for their acknowledgements to have a graceful system termination. The distributed termination algorithm is implemented by the termination detection unit (TDU), which keeps track of the incoming and outgoing network messages in each node. We choose an implicit distributed termination detection algorithm to reduce the programming effort. The same algorithm was adopted by DDM-VM. The main difference between the two implementations is that FREDDO uses *atomic variables* to count the number of outgoing and incoming messages in each node. In DDM-VM, the same functionality is implemented using lock/unlock operations, which incur more overheads.

3.5 Network Manager

The Network Manager is responsible for handling the internode communication. It is implemented as a software module that relies on the underlying network hardware interface. It also has the following responsibilities: (i) establishes connections between the system's nodes, (ii) exchanges network messages between the nodes, (iii) processes the incoming network messages appropriately (e.g., it sends the incoming Updates to the local TSU), and (iv) supports data forwarding across the GAS. The Network Manager handles the low-level connectivity by utilizing two different network interfaces: a CNI, which is an optimized implementation with TCP sockets, and the widely used MPI library [26]. The CNI implements a fully connected mesh of internode connections (i.e., each node maintains a connection to all other nodes). Currently, the CNI supports only Ethernet-based interconnects. The major benefits of providing MPI support are portability and flexibility. However, the CNI implementation allows a direct and fair comparison with similar frameworks that also utilize a CNI.

The Network Manager tolerates the network communication latencies by overlapping its sending/receiving functionalities with the DThread instances' execution and the TSU's functionalities. The sending functionalities (i.e., operations for sending commands and produced data) are handled by the kernels. This removes the cost of such operations from the TSU's critical path. Notice that a sending operation returns when the message was stored in the network layer buffers of the OS. For the receiving functionalities, an *auxiliary thread* is used, which continuously retrieves the incoming network messages from the other nodes. For this purpose, the *pselect* routine is used for the CNI implementation. For the MPI implementation, we are using the *MPI_Recv* routine with *source=MPI_ANY_SOURCE*.



(a) No compression (b) Compression for DThreads with $RC > 1$ (c) Reducing Update messages for DThreads with $RC = 1$

Fig. 3. Example of reducing the network traffic generated by a Multiple Update. T1(X,Y) denotes a Multiple Update for DThread T1.

3.6 Reducing the Network Traffic

Reducing the network traffic in HPC systems is critical, as it can help avoid network saturation and reduce the power consumption. The U.S. Department of Energy (DOE) [5, 56] clearly states that the biggest energy cost in future massively parallel HPC systems will be in data movement, especially moving data on and off chip. To this end, we are recommending four simple and efficient techniques for reducing the network traffic in distributed DDM applications running on HPC systems. These techniques are mostly applied to the Update operations, which are the most frequent commands executed in a DDM application:

(1) Use General Network Packets: A network message can carry any type of command (Update, Multiple Update, Data Descriptor, etc.). The most common practice to send such a message is to use a header (as a separate message) that describes the message's content, as in Michael et al. [48]. In this work, we are introducing a *general packet* with four fields (Type=1-byte, Value_1=4-bytes, Value_2=*sizeof* (Context Value),[1] and Value_3=*sizeof* (Context Value)[1]) that can carry all basic types of commands. As a result, the number of sending network messages can be halved.

(2) Compressing Multiple Updates for DThreads With RC > 1: Multiple Updates decrement the RC values of several instances of a DThread. Since the mapping of instances to the nodes is based on their Context values, a Multiple Update should be unrolled and each of its Updates should be sent to the appropriate node. Figure 3(a) shows an example where a Multiple Update, with Contexts from <0> to <47>, is distributed to a 4-node system (each node has four cores). We are compressing consecutive Multiple Updates that are sent to the same node based on a simple pattern recognition algorithm. The algorithm takes into account the difference between the minContext and maxContext of each Multiple Update command (called *Right Distance*) and the difference between the minContexts of two consecutive Multiple Updates (called *Bottom Distance*). In this example, the algorithm compresses the Multiple Updates that are sent to each node using $RightDistance = 3$ and $BottomDistance = 16$ (see Figure 3(b)). As a result, the number of messages will be reduced by 75% for this specific Multiple Update command. The proposed algorithm is implemented by the DSU's *compression unit*. When a node receives a compressed Multiple Update, the Network Manager decompresses it by its *decompression unit*.

(3) Reducing the Number of Messages in the Case of Multiple Updates for DThreads With RC = 1: In FREDDO, the DThreads with RC = 1 are treated differently compared to other DDM implementations. The TSU does not allocate RC values for their instances in order to reduce the memory allocation [45]. Instances of such DThreads are scheduled immediately when Updates

[1]FREDDO supports four different sizes for the Context values: 32 bit, 64 bit, 128 bit, and 192 bit.

are received for them. This approach allows scheduling a DThread instance for execution to any node. We can benefit from this by dividing the range of Context values of a Multiple Update, into equal parts where the number of parts is equal to the number of nodes. As an example, consider the Multiple Update of Figure 3(a) and that the DThread T1 has RC = 1. In this case, four different Multiple Updates will be distributed to the nodes as described in Figure 3(c). This methodology does not require compression, and it is possible to reduce the number of messages by 75% for this specific example.

(4) Packing Correlated Updates Together: Our final technique reduces the messages that carry Update commands for the same destination node. Specifically, when a producer instance sends several Update commands (Single Updates, (Un)compressed Multiple Updates) to a remote node, for the same DThread, FREDDO's runtime performs two steps. It sends the DThread's TID along with the number of the Updates that will be sent through a general packet. After that, the Context values of the Updates are sent as a single data packet to the remote node.

3.7 Distributed Recursion Support

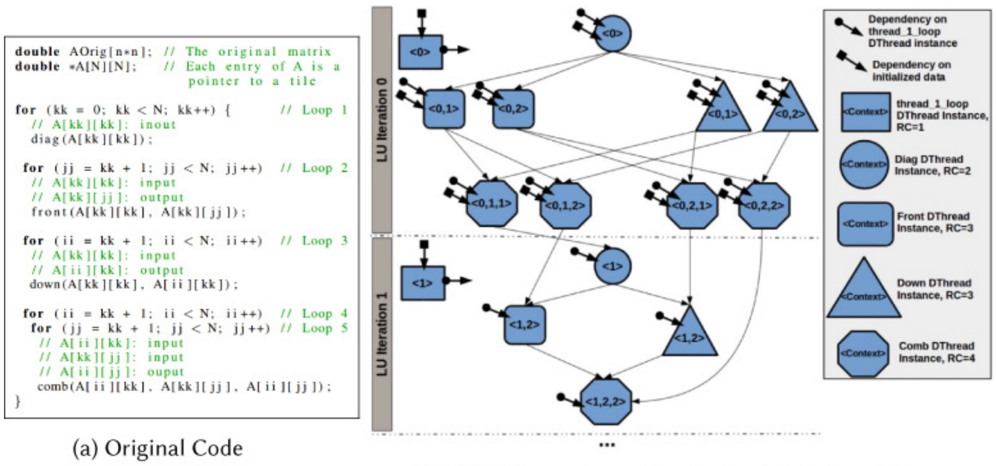
For supporting distributed execution of recursive algorithms in a data-driven manner, we have extended the functionalities of two of FREDDO's DThread classes: *RecursiveDThread* and *ContinuationDThread* [45]. RecursiveDThread is a special class that allows parallelizing different types of recursive functions (linear, tail, etc.). It allocates/deallocates the arguments and the return values of recursive instances dynamically at runtime. The ContinuationDThread can be used in combination with the RecursiveDThread to implement algorithms with multiple recursion (or any similar algorithms). For instance, it can be used to sum the return values of two children recursive calls and return the result to their parent, in a parallel implementation of the recursive Fibonacci algorithm. Each recursive call is associated with a DistRData object. DistRData holds the arguments of a recursive call, pointers to the return values of its children (if any) and a pointer to the DistRData of its parent. Thus, the DistRData objects correlate children recursive instances with their parents. When a parent instance calls one or more children instances, the DSU decides which of them will be executed on remote nodes, based on their Context values. In this case, the Network Manager will send the DistRData objects and the children instances' Context values to the remote nodes to be scheduled for execution. When a child instance returns a value to its parent, the runtime system acknowledges if the parent instance is mapped on the local node or on a remote node. In the latter case, the return value is sent via a network message to the remote node, and finally it is stored in the parent's DistRData object.

4 PROGRAMMING EXAMPLE

In this section, we present a programming example using a benchmark with a complex dependency graph: the tile LU decomposition algorithm. The algorithm is based on an earlier version developed in StarSs [52], and it factors a dense matrix into the product of a lower triangular L and an upper triangular U matrix. The dense $n \times n$ matrix A is divided into an $N \times N$ array of $B \times B$ tiles ($n = NB$). The code of the original algorithm is shown in Figure 4(a), which is composed of five nested loops that perform four basic operations on a tiled matrix. For demonstration purposes, we choose the following indicative names for the operations: *diag*, *front*, *down*, and *comb*.

4.1 Dependency Graph

One possible implementation of the tile LU algorithm in DDM/FREDDO is to map the outermost loop and the four basic operations into five DThreads, called *thread_1_loop*, *diag_thread*, *front_thread*, *down_thread*, and *comb_thread*. The data dependencies between the five DThreads are listed next:



```
double AOrig[n*n]; // The original matrix
double *A[N][N];   // Each entry of A is a
                      pointer to a tile

for (kk = 0; kk < N; kk++) {         // Loop 1
  // A[kk][kk]: inout
  diag(A[kk][kk]);

 for (jj = kk + 1; jj < N; jj++)     // Loop 2
  // A[kk][kk]: input
  // A[kk][jj]: output
  front(A[kk][kk], A[kk][jj]);

 for (ii = kk + 1; ii < N; ii++)     // Loop 3
  // A[kk][kk]: input
  // A[ii][kk]: output
  down(A[kk][kk], A[ii][kk]);

 for (ii = kk + 1; ii < N; ii++)     // Loop 4
  for (jj = kk + 1; jj < N; jj++)    // Loop 5
   // A[ii][kk]: input
   // A[kk][jj]: input
   // A[ii][jj]: ouput
   comb(A[ii][kk], A[kk][jj], A[ii][jj]);
}
```

LU Iteration 0
LU Iteration 1
<0>
<0,1>
<0,2>
<0,1,1>
<0,1,2>
<0,2,1>
<0,2,2>
<1>
<1,2>
<1,2,2>
...
Dependency on thread_1_loop DThread instance
Dependency on initialized data
<Context> thread_1_loop DThread instance, RC=1
<Context> Diag DThread Instance, RC=2
<Context> Front DThread Instance, RC=3
<Context> Down DThread Instance, RC=3
<Context> Comb DThread Instance, RC=4

(a) Original Code

(b) DDM Dependency Graph of a 3 × 3 tile matrix

Fig. 4. Tile LU decomposition.

- The DThreads that execute the operations depend on the *thread_1_loop* since the index of the outermost loop is used in the four operations.
- The *front_thread* and *down_thread* DThreads depend on the *diag_thread*.
- The *comb_thread* depends on the *front_thread* and *down_thread* DThreads.
- The next LU iteration depends on the results of the previous iteration. The results produced by the *comb_thread* invocations, in the current iteration, are consumed by the invocations of the *diag_thread*, *front_thread*, *down_thread*, and *comb_thread* of the next LU iteration.

The DDM dependency graph shown in Figure 4(b) depicts the dependencies among the instances of the DThreads for the first two iterations of the tile LU algorithm. For simplicity, a 3 × 3 tile matrix (N = 3) was selected. Each DThread's instance is labeled with the value of its Context.

4.2 The FREDDO Code

Figure 5 depicts the FREDDO code for the tile LU decomposition algorithm. We used the DSM/GAS features of FREDDO since the algorithm uses a shared object: the tile matrix *A*. In FREDDO, each DThread object is associated with a *DFunction* [45]. The DFunction is a callable target (C++ function, Lambda expression, or functor) that holds the code of a DThread. Each DFunction has one input argument: the Context value. Different Context structures (ContextArg, Context2DArg, and Context3DArg) are provided based on the type of DThread class. In this example, the code of each DThread was placed in standard C/C++ functions.

Each call of an Update command in the DThreads' code corresponds to one dependency arrow in Figure 4(b). A DThread object (DObject) provides several Single and Multiple Update methods that target itself or its consumers [45]. For example, the first Update command of the loop_1_code DFunction indicates a Single Update that decrements the RC value of the instance kk of diag_thread (diagDT) by one. The second Update command of the same DFunction is a Multiple Update that decrements the RC value of multiple instances of the front_thread (frontDT) by one. The Update commands at the end of the comb_code DFunction implement a switch actor, of which depending on the Context of the DThread's instance, a different consumer-instance is updated.

In the *main function* of the program the matrices are allocated and initialized. After that, the tile matrix *A* is registered in the GAS using the *addInGAS* runtime function. At this point, FREDDO's

```
#include <freddo/dthreads.h>
using namespace ddm; // Use the freddo namespace

// DThread Objects
MultipleDThread   *loop_1DT, *diagDT;
MultipleDThread2D *frontDT,  *downDT;
MultipleDThread3D *combDT;
AddrID gasA; // The GAS_ID of matrix A
TYPE ***A;   // The tile matrix
TYPE *Aorig; // The original matrix
int tS = B*B*sizeof(TYPE); // size of tile in bytes

// The code of the thread_1_loop DThread
void loop_1_code(ContextArg kk) {
 diagDT->update(kk);

 if (kk < N-1) {
  frontDT->update({kk, kk+1}, {kk, N-1});
  downDT->update({kk, kk+1}, {kk, N-1});
  combDT->update({kk, kk+1, kk+1}, {kk, N-1, N-1});
 }
}

// The code of diag_thread DThread
void diag_code(ContextArg kk) {
 addModifiedSegmentInGAS(gasA, A[kk][kk], tS);
 diag(A[kk][kk]); // diag operation
 sendDataToRoot(gasA, A[kk][kk], tS);

 if (kk < N-1) {
  frontDT->update({kk, kk+1}, {kk, N-1});
  downDT->update({kk, kk+1}, {kk, N-1});
 }
}

// The code of the front_thread DThread
void front_code(Context2DArg context) {
 int kk = context.Outer, jj = context.Inner;
 addModifiedSegmentInGAS(gasA, A[kk][jj], tS);
 front(A[kk][kk], A[kk][jj]); // front operation
 sendDataToRoot(gasA, A[kk][jj], tS);

 combDT->update({kk, kk+1, jj}, {kk, N-1, jj});
}

// The code of the down_thread DThread
void down_code(Context2DArg context) {
 int kk = context.Outer, jj = context.Inner;
 addModifiedSegmentInGAS(gasA, A[jj][kk], tS);
 down(A[kk][kk], A[jj][kk]); // down operation
 sendDataToRoot(gasA, A[jj][kk], tS);

 combDT->update({kk, jj, kk+1}, {kk, jj, N-1});
}
```

```
// The code of the comb_thread DThread
void comb_code(Context3DArg context) {
 int kk = context.Outer, ii = context.Middle,
     jj = context.Inner;
 addModifiedSegmentInGAS(gasA, A[ii][jj], tS);

 // comb operation
 comb(A[ii][kk], A[kk][jj], A[ii][jj]);

 // Updates for the next LU iteration
 if (ii == kk+1 && jj == kk+1) {
      diagDT->update(kk+1);
 } else if (ii == kk+1) {
      frontDT->update({ii, jj});
 } else if (jj == kk+1) {
      downDT->update({jj, ii});
 } else {
      combDT->update({kk+1, ii, jj});
 }
}

// The main program
void main(int argc, char* argv[]) {
 // Initialize data (matrices, etc.)
 initializeData();

 // Register A in GAS
 gasA = addInGAS(A[0][0]);

 // Initializes the FREDDO execution environment
 init(&argc, &argv, NUM_OF_KERNELS);

 // Allocation of the DThread Objects
 loop_1DT = new MultipleDThread(loop_1_code, 1);
 diagDT   = new MultipleDThread(diag_code, 2);
 frontDT  = new MultipleDThread2D(front_code, 3);
 downDT   = new MultipleDThread2D(down_code, 3);
 combDT   = new MultipleDThread3D(comb_code, 4);

 // Updates resulting from data initialization
 if (ddm::isRoot()) {
  loop_1DT->update(0, N-1);
  diagDT->update(0);
  frontDT->update({0, 1}, {0, N-1});
  downDT->update({0, 1}, {0, N-1});
  combDT->update({0, 1, 1}, {0, N-1, N-1});
 }

 // Starts the DDM scheduling in each node
 run();

 // Releases the resources of distributed FREDDO
 finalize();
}
```

Fig. 5. FREDDO code of the tile LU algorithm (the highlighted code is required for the distributed execution).

runtime registers the address of the tile matrix *A* in the GAD of each node. The runtime also creates a GAS_ID for the matrix *A*, which is stored in the *gasA* variable. The *init* runtime function initializes FREDDO's execution environment, and it starts NUM_OF_KERNELS kernels in each node. Each DObject's constructor takes two arguments: the DFunction and the RC value (e.g., the diagDT object has DFunction=diag_code and RC = 2).

After the creation of the DObjects, the initial Updates are sent to the TSUs for execution. These Updates correspond to the arrows of Figure 4(b) that describe dependencies on initialized data. The initial Updates have to be executed one time only. In this example, the *RootNode* was selected to execute these Updates that are distributed across the nodes through its DSU module. Notice that the initial Updates or any other Updates can be executed by any node of the system. The *run* function starts DDM scheduling and waits until FREDDO's runtime detects the distributed execution termination (see Section 3.4). When the *run* function returns, all resources allocated by FREDDO are deallocated using the *finalize* function.

Table 1. Systems Used for Benchmark Evaluation

System Specs	AMD	CyTera
Processor Type	AMD Opteron 6276	Intel Xeon X5650
Number of Nodes	4	64
Per Node Specs	(Total RAM for both systems: 48GB)	
- Clock Frequency	1.4GHz	2.67GHz
- Cores	16 (2 sockets, 8 cores/socket)	12 (1 socket)
- Hardware Threads/Core	2	1
- Total Hardware Threads	32	12
- L1-I$, L1-D$, L2$/Core	32KB, 16KB, 1MB	32KB, 32KB, 256KB
- L3$	24MB	12MB
Interconnect	Gigabit Ethernet	Infiniband QDR (40Gb/s)
Linux Kernel	3.13.0	2.6.32
Compilers	gcc/g++ 4.8.4	gcc/g++ 4.9.3

FREDDO's memory model in combination with its distribution scheme and the implicit distributed termination approach allows the distributed FREDDO programs to be fundamentally the same as the single-node ones. For the distributed data-driven execution, users have to (i) provide a peer file that contains the IP addresses or the host names of the system's nodes, (ii) register the shared objects in the GAS using the *addInGAS* function, and (iii) specify the output data of each DThread using the *addModifiedSegmentInGAS* runtime function. For gathering the results in *RootNode*, users have to use the *sendDataToRoot* runtime function. Both the *addModifiedSegmentInGAS* and *sendDataToRoot* functions require the GAS_ID of a shared object, the conventional main memory address of that object, and its size in bytes. For instance, in the diag_code DFunction, the tile *A[kk][kk]* is declared as a modified segment since is computed by the *diag* routine. The size of this tile is equal to *tS*, and its GAS_ID is equal to *gasA* since it is a part of the tile matrix *A*. In Figure 5, the required code for the distributed execution is highlighted.

FREDDO's DSM system creates an identical address space on each node, which gives the view of a single distributed address space. Currently, this requires the shared objects to have the same memory size in each node (e.g., the tile matrix *A*). This simplifies the implementation of the proposed programming model, but it limits the total amount of memory used by a DDM program. A program can use only as much memory as is available in the *RootNode* since the output results are gathered in that node. Mechanisms that will overcome this limitation are in our to-do list.

5 EXPERIMENTAL EVALUATION

In this section, we present the evaluation of distributed FREDDO using eight benchmarks with different characteristics. We start with an overview of the hardware environment used in our experiments, followed by the description of the benchmarks. After that, we present performance results and comparisons with other systems. Finally, we present network traffic analysis results.

5.1 Experimental Setup

For the experimental evaluation, we used two different systems: AMD and CyTera. AMD is a 4-node system. CyTera [29] is an open-access HPC system that provides up to 64 nodes per user. The specifications of the systems are shown in Table 1. Our benchmark suite contains three applications that require low communication between the nodes (*BMMULT*, *Blackscholes*, and *Swaptions*), three benchmarks with complex dependency graphs that require heavy internode communication (*LU*, *QR*, and *Cholesky*), and two recursive algorithms (*Fibonacci* and *PowerSet*) that require medium internode communication:

(1) *BMMULT* performs a dense blocked matrix multiplication of two square matrices.
(2) *Blackscholes* calculates the prices for a portfolio of European options analytically with the Black-Scholes partial differential equation (PDE) [11].

(3) *Swaptions* uses the Heath-Jarrow-Morton (HJM) framework to price a portfolio of swaptions [11]. A Monte Carlo simulation is used to compute the prices (the simulation number variable is set to 20,000).
(4) *LU* calculates the LU decomposition of a tile matrix. It was based on an earlier version written in StarSs [52].
(5) *QR* implements the right-looking tile QR factorization. The algorithm uses LAPACK [6] (V3.6.1) and PLASMA [1] (V2.8.0) routines: *geqrt*, *tsqrt*, *ormqr*, and *tsmqr*.
(6) *Cholesky* calculates the lower triangular matrix L of a symmetric positive definite matrix A such that $A = LL^T$. Operations on the tiles are performed using the following LAPACK [6] (V3.6.1) routines: *syrk*, *gemm*, *potf2*, and *trsm*.
(7) *Fibonacci* calculates the Fibonacci numbers using a double recursion algorithm.
(8) *PowerSet* calculates the number of all subsets of a set with N elements, using a multiple recursion algorithm. The original algorithm was retrieved from BSC [15].

For the benchmarks working on tile/block matrices, we used both single-precision (SP) and double-precision (DP) floating-point dense matrices. All source codes and libraries/packages were compiled using the *−O3* optimization flag. For the performance results that are reported as *speedup*, speedup is defined as Savg/Pavg, where *Savg* is the average execution time of the sequential version of the benchmark (without any FREDDO overheads) and *Pavg* is the average execution time of the FREDDO implementation. For the average execution times, we executed each benchmark (both sequential and parallel) five times. In the parallel execution time of each execution, we included the time needed for gathering the results to the *RootNode*.

We have executed benchmarks using FREDDO with CNI support (called *FREDDO+CNI*) and with MPI support (called *FREDDO+MPI*). The default implementation for the AMD system is FREDDO+CNI. In CyTera, the MPI libraries provided to us are configured for the InfiniBand interconnect. As such, we are using the FREDDO+MPI implementation as the default since it provides faster communication compared to FREDDO+CNI. For the FREDDO+MPI implementation, we are using the OpenMPI library (V1.8.4 for CyTera and V2.0.1 for AMD). Notice that for both implementations, the size of the Context values is set to 64 bit.

5.2 Performance Evaluation

We performed a scalability study to evaluate the performance of the proposed framework by varying the number of nodes on the two systems. Each benchmark is executed with three different problem sizes. For the tiled algorithms (*BMMULT*, *LU*, *Cholesky*, and *QR*), we choose the optimal tile size for both the sequential and parallel implementation of each algorithm. For each different execution (problem size and number of nodes), we run experiments with three different tile sizes: 32×32, 64×64, and 128×128. Out of the total number of cores in each node, one of them is used for executing the TSU code, whereas the rest are used for executing the kernels. Unlike the kernels and the TSU, which are pinned to specific cores, the Network Manager's *receiving thread* is not pinned to any specific core. This gives the opportunity to the operating system to move the receiving thread to an idle core, or to migrate it regularly between the cores. For the single-node execution of the benchmarks, the Network Manager's receiving thread is disabled.

Figures 6 and 7 depict the results for the AMD and CyTera systems, respectively. On the former system, we executed all benchmarks, including both SP and DP versions of the algorithms working on tile/block matrices. On the latter system, we executed the two recursive algorithms and the SP versions of the tiled algorithms. *Blackscholes* and *Swaptions*, as well as the DP versions of the tiled algorithms, are excluded from our performance evaluation on the CyTera system to save computational resources (CPU hours). *Ideal speedup* refers to the maximum speedup that can be

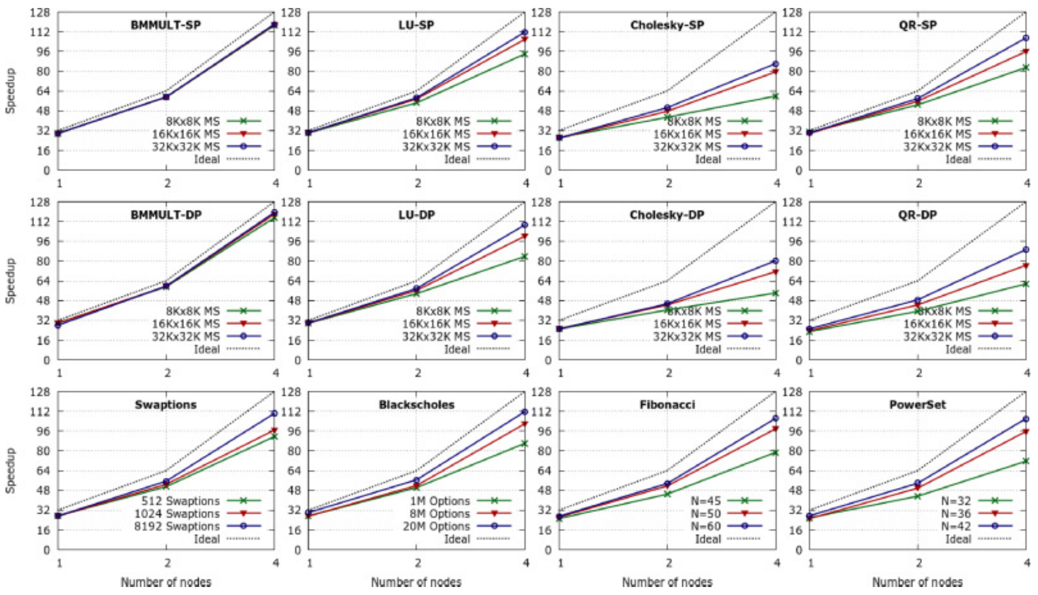


Fig. 6. Strong scalability and problem size effect on the AMD system using FREDDO+CNI (MS, matrix size; SP, single precision; DP, double precision; $K = 2^{10}$; $M = 10^6$).

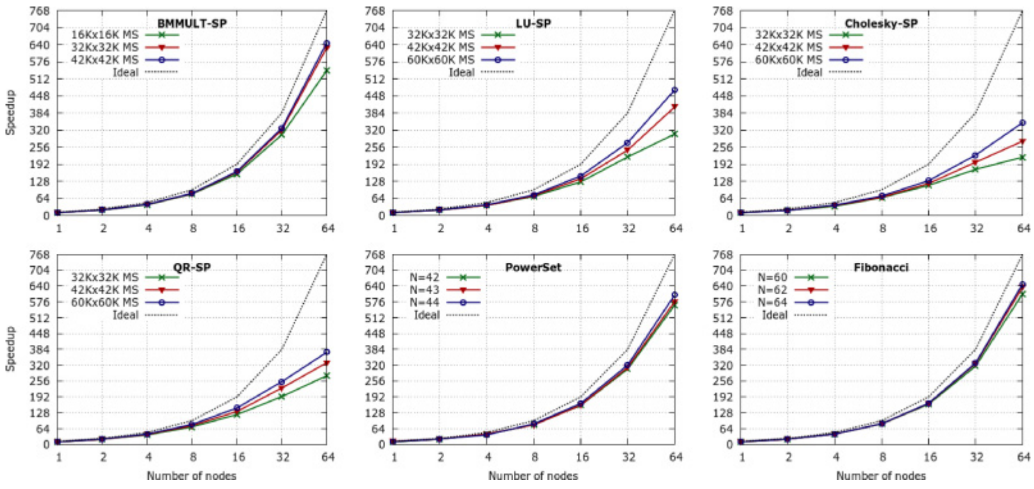


Fig. 7. Strong scalability and problem size effect on the CyTera system using FREDDO+MPI (MS, matrix size; SP, single precision; $K = 2^{10}$).

achieved in relation to the number of cores used for the parallel execution. For example, on CyTera for the 64-node configuration, the ideal speedup is equal to 768. From the performance results, we observe that generally as the input size increases, the system scales better (especially for the benchmarks with complex dependency graphs). This is expected, as larger problem sizes allow amortizing the overheads of the parallelization. Table 2 depicts the average sequential time (in seconds) of the sequential version of the benchmarks that were executed on both systems. The DP

Table 2. Average Sequential Execution Time (in Seconds) of the Sequential Version of the Benchmarks

Benchmark	AMD			CyTera		Benchmark	AMD		CyTera	
	Matrix Size	Execution Time of SP	Execution Time of DP	Matrix Size	Execution Time of SP		Problem Size	Execution Time	Problem Size	Execution Time
BMMULT	**8Kx8K**	1,893.34	2,143.96	**16Kx16K**	5,664.72	Fibonacci	**N=45**	12.59	**N=60**	8,885.24
	16Kx16K	15,110.28	17,196.18	**32Kx32K**	45,340.37		**N=50**	140.70	**N=62**	23,250.33
	32Kx32K	120,816.96	139,346.05	**42Kx42K**	102,586.97		**N=60**	17,291.87	**N=64**	60,795.54
LU	**8Kx8K**	636.48	709.37	**32Kx32K**	15,350.72	PowerSet	**N=32**	18.61	**N=42**	9,721.38
	16Kx16K	5,135.44	5,745.75	**42Kx42K**	34,411.99		**N=36**	304.29	**N=43**	19,295.20
	32Kx32K	41,572.70	47,079.94	**60Kx60K**	101,666.51		**N=42**	20,098.89	**N=44**	38,261.29
Cholesky	**8Kx8K**	170.10	181.76	**32Kx32K**	4,106.64	Swaptions	**512 Swaptions**	50.53	—	—
	16Kx16K	1,310.03	1,404.20	**42Kx42K**	9,273.48		**1024 Swaptions**	97.67	—	—
	32Kx32K	10,313.71	11,068.11	**60Kx60K**	27,034.89		**8192 Swaptions**	781.77	—	—
QR	**8Kx8K**	5,016.25	3,900.37	**32Kx32K**	131,237.58	Blackscholes	**1M Options**	141.19	—	—
	16Kx16K	40,585.11	31,867.58	**42Kx42K**	307,914.23		**8M Options**	1,123.27	—	—
	32Kx32K	324,817.37	268,552.39	**60Kx60K**	921,452.84		**20M Options**	2,826.60	—	—

Table 3. Thresholds Used for the Execution of the Recursive Algorithms

Problem Size	Fibonacci										PowerSet									
	AMD System			CyTera System							AMD System			CyTera System						
	1	2	4	1	2	4	8	16	32	64	1	2	4	1	2	4	8	16	32	64
Smaller	29	29	29	35	35	35	35	35	35	35	14	12	11	15	14	15	15	16	16	16
Medium	30	30	30	35	35	35	35	35	35	35	13	14	15	15	15	15	15	16	16	17
Largest	33	33	33	35	35	35	35	35	35	37	13	13	15	15	15	16	16	15	15	17

versions of the algorithms achieve slightly lower speedups compared to the SP ones, because in the former case the data exchanged in the network is doubled.

BMMULT, *Blackscholes*, and *Swaptions* achieve very good speedups due to the low data sharing and low data exchange between the nodes. On AMD, for the 4-node configuration and the largest problem size, they achieve up to 93% of the ideal speedup. On CyTera, for the 64-node configuration and the largest problem size, *BMMULT* achieves 84% of the ideal speedup. *LU*, *QR*, and *Cholesky* are classic dense linear algebra workloads with complex dependency graphs. FREDDO ended up with lower speedups as the number of nodes increases, due to the heavy internode communication and the complexity of the algorithms. When utilizing all nodes of CyTera, for the largest problem size, FREDDO achieves up to 61% of the ideal speedup for these complex algorithms. However, it is expected that for larger problem sizes, a better performance can be achieved.

The recursive algorithms (*Fibonacci* and *PowerSet*) also achieve very good speedups. For the 4-node configuration on AMD and the largest problem size, FREDDO achieves about 83% of the ideal speedup (106 out of 128). For the 64-node configuration on CyTera, FREDDO achieves 84% (648 out of 768) for *Fibonacci* and 79% (604 out of 768) for *PowerSet*, of the ideal speedup, also for the largest problem size. For minimizing the overheads of the parallel recursive implementations, we used *thresholds* to control the number of DThread instances that are used for executing the recursive calls. For each problem size of the algorithms, we test various thresholds and choose the one that provides the best performance. Table 3 depicts the thresholds used to achieve the best performance.

To conclude, distributed FREDDO scales well and effectively leverages the decoupling of synchronization and execution. Table 4 depicts the minimum, maximum, and average speedup results, on both systems, for each problem size and number of nodes. Next to each speedup value, the utilization percentage of the available cores is presented. The results show that FREDDO utilizes the resources of both systems efficiently, especially for the largest problem size. For the largest problem size and when all available nodes are used, FREDDO achieves an average of 82% of the ideal speedup on AMD and 67% on CyTera.

Table 4. Speedup Results Along With the Utilization Percentage of the Available Cores in Each Case

Problem Size	Speedup	AMD System			CyTera System						
		1	2	4	1	2	4	8	16	32	64
Smaller	Avg	27.5 (86%)	49.2 (77%)	83.1 (65%)	10.5 (88%)	20.5 (85%)	38.9 (81%)	75.2 (78%)	139.1 (72%)	251.3 (65%)	419.1 (55%)
	Min	22.9 (72%)	39.3 (61%)	54.1 (42%)	10.4 (86%)	19.7 (82%)	34.1 (71%)	66.6 (69%)	112.5 (59%)	172.5 (45%)	218.2 (28%)
	Max	30.6 (96%)	59.2 (92%)	117.0 (91%)	10.8 (90%)	21.2 (89%)	41.9 (87%)	82.8 (86%)	162.5 (85%)	316.1 (82%)	608.3 (79%)
Medium	Avg	27.5 (86%)	52.7 (82%)	96.5 (75%)	10.7 (89%)	20.6 (86%)	40.0 (83%)	76.8 (80%)	145.9 (76%)	271.5 (71%)	475.8 (62%)
	Min	23.7 (74%)	44.5 (69%)	71.3 (56%)	10.4 (86%)	19.8 (82%)	37.2 (78%)	67.8 (71%)	119.6 (62%)	199.3 (52%)	278.0 (36%)
	Max	30.2 (94%)	59.5 (93%)	117.9 (92%)	11.9 (99%)	21.2 (88%)	42.3 (88%)	83.1 (87%)	165.3 (86%)	326.4 (85%)	634.7 (83%)
Largest	Avg	28.0 (87%)	54.9 (86%)	104.9 (82%)	10.5 (88%)	20.7 (86%)	40.3 (84%)	79.8 (83%)	153.5 (80%)	288.1 (75%)	514.6 (67%)
	Min	25.0 (78%)	45.4 (71%)	80.2 (63%)	10.3 (86%)	18.8 (78%)	38.4 (80%)	72.5 (76%)	130.1 (68%)	225.9 (59%)	347.4 (45%)
	Max	30.3 (95%)	59.8 (93%)	119.3 (93%)	10.9 (91%)	21.5 (90%)	41.9 (87%)	83.4 (87%)	166.3 (87%)	329.0 (86%)	647.9 (84%)

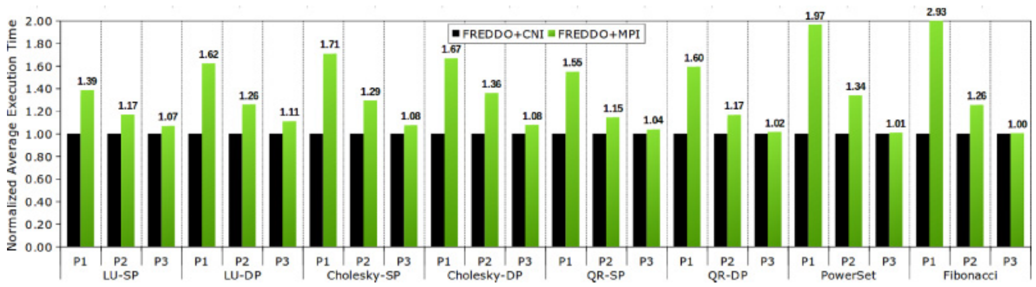


Fig. 8. FREDDO+CNI versus FREDDO+MPI on AMD for the 4-node configuration. P1, P2, and P3 indicate the smaller, medium, and largest problem sizes, respectively.

5.3 FREDDO: CNI Versus MPI

In this section, we study the performance penalties of using MPI instead of a CNI for the benchmarks that have medium and heavy internode communication. For our experiments, we used the AMD system with all available nodes. The results are shown in Figure 8 and are normalized based on the average execution time of FREDDO+CNI. The comparisons show that FREDDO+CNI is 80%, 25%, and 5% faster than FREDDO+MPI on average for the smallest, medium, and largest problem sizes, respectively. This indicates that MPI has more overheads that affect the performance of the Network Manager's receiving thread as well as the sending operations of the kernels. MPI has more overheads than a CNI since it is a much larger library that contains more functionalities than a CNI. However, MPI's overheads are hidden as the benchmark's input size increases. Thus, FREDDO+MPI can be used for real-life applications that have enormous input sizes (at least in the order of our largest problem size). This solution can provide better portability to FREDDO applications, especially when targeting large-scale HPC systems with different architectures.

5.4 Performance Comparisons With Other Systems

FREDDO is compared to MPI, DDM-VM [48], and OmpSs@Cluster [16, 17]. MPI is the reference programming model in HPC systems and clusters. DDM-VM is a dataflow system similar to FREDDO that supports static dependency resolution—i.e., the programmer/compiler is responsible for constructing the dependency graph. OmpSs@Cluster is a dataflow-based system that supports dynamic dependency resolution—i.e., the dependency graph is constructed dynamically at runtime. For the comparisons between FREDDO and MPI, we used *Cholesky*, *LU*, *Fibonacci*, and *PowerSet*. For the MPI implementation of Cholesky, we used the *pspotrf* and *pdpotrf* routines of ScaLAPACK [12] (V2.0.2). The MPI implementation of *LU* was retrieved from mpich.org [50] and utilizes MPI's one-sided communication functionalities. The recursive algorithms implemented in MPI are based on the Fibonacci algorithm presented in Cera et al. [19], which uses MPI's dynamic process management support through *MPI_Comm_spawn*. Additionally, the recursive

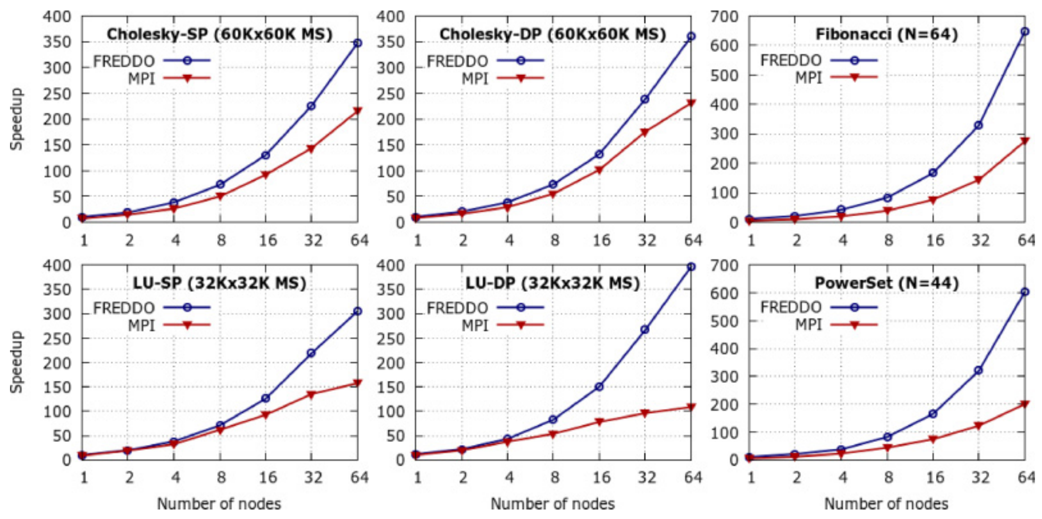


Fig. 9. FREDDO+MPI versus MPI on CyTera.

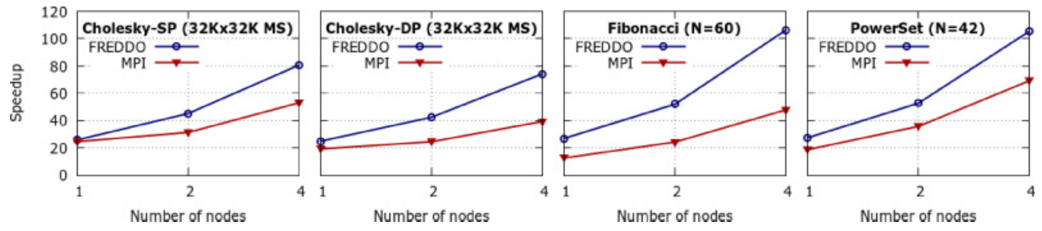


Fig. 10. FREDDO+MPI versus MPI on AMD.

algorithms were modified to support thresholds to improve the performance. For the comparisons between FREDDO and DDM-VM, we used *Cholesky* and *LU*. Finally, we compare FREDDO to OmpSs@Cluster using *Cholesky*, *LU*, and *QR*. For all frameworks, we choose the configurations that achieve the optimal performance (e.g., tile sizes, thresholds for the recursive algorithms, and grid configurations for the ScaLAPACK implementations).

OmpSs@Cluster was installed with GASNet [13] (V1.28.2) and OpenMPI (V1.8.4) on CyTera. We used the latest stable OmpSs package, which includes the Mercurium compiler V2.0.0-28-06-2017 and the Nanos++ runtime V0.13-29-06-2017. The OmpSs benchmarks were executed using GASNet's *ibv-conduit*. To have fair comparisons, FREDDO is compared to DDM-VM using the FREDDO+CNI implementation and with MPI and OmpSs@Cluster using the FREDDO+MPI implementation. The reason is that the MPI library incurs more overheads (as shown in Figure 8) compared to a CNI with fewer functionalities.

The comparison results of FREDDO and MPI on CyTera and AMD are depicted in Figures 9 and 10, respectively. *LU* is not included in our comparisons on AMD, as the system does not support remote direct memory access (RDMA), which is required by the MPI implementation of the algorithm. Our framework scales better than MPI. On average, FREDDO is 1.79× faster than MPI, on AMD, for the 4-node configuration and 2.35× faster, on CyTera, for the 64-node configuration. In the case of *Cholesky* and *LU*, FREDDO performs better since it exploits the advantages of having fine-grain threads/tasks executing asynchronously, in a data-driven manner. MPI relies on the

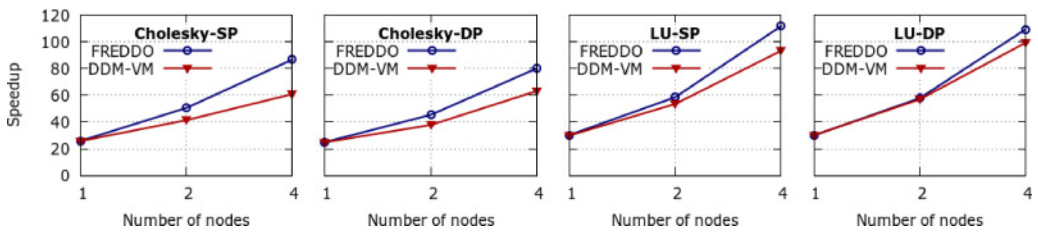


Fig. 11. FREDDO+CNI versus DDM-VM on AMD (MS: $32K \times 32K$).

fork-join paradigm and uses barriers to synchronize between phases of computation. Synchronization constructs, like barriers, can have a negative impact on performance, especially when the number of nodes increases. However, FREDDO does not use such synchronization constructs, as it relies solely on data-driven mechanisms for its operations. In the MPI implementations of the recursive algorithms, the *MPI_Comm_spawn* routine is used to spawn children recursive calls through new MPI processes. A parent instance waits for its children instances to complete before it proceeds with computations, thus increasing the runtime overheads (a parent instance blocked until all of its children instances finished their execution). As well, additional overheads are introduced by the dynamic allocation of new MPI processes. In FREDDO, such overheads are eliminated by executing recursive algorithms based on dataflow with continuations; a continuation DThread instance is activated to process the results of the children instances when all of them finished their execution. Moreover, all instances are executed by the kernels, and thus there is no need to allocate new resources (processes, threads, etc.).

The comparison results of FREDDO and DDM-VM, on the AMD system, are depicted in Figure 11. On average, FREDDO is 1.25× faster than DDM-VM, for the 4-node configuration. Although DDM-VM and FREDDO are based on the same execution model, FREDDO achieves better performance for three main reasons. First, DDM-VM follows a Context-based distribution scheme (similar to this work) where each DThread instance is mapped and executed on a specific core of the system. In FREDDO, the DThread instances are mapped to specific nodes and the TSU's scheduler distributes them to the kernels with the least workload. This approach can improve the load balancing in each node. Second, FREDDO provides an optimized TSU and Network Manager (e.g., it uses atomic variables to implement the distributed termination detection algorithm where DDM-VM uses lock/unlock operations, which incur more overheads). Third, in DDM-VM, the TSU and the receiving thread are pinned on the same core. This approach can affect the scheduling and network operations, thus increasing the runtime overheads. In FREDDO, the receiving thread is not pinned to any specific core, which gives the flexibility to the operating system to schedule it appropriately.

Figure 12 compares FREDDO to OmpSs@Cluster on CyTera. We used the *affinity* scheduling policy, which is used for running applications on a cluster. We executed the OmpSs benchmarks with different tile sizes (from 32 × 32 to 2,048 × 2,048) and found that the 512 × 512 tile size achieved the best performance in all cases. Our experiments show that OmpSs does not perform well on benchmarks with small tiles. Smaller tile sizes increase the number of tasks, which increase the Nanos++'s workload required to determine the data dependencies between those tasks. Notice that OmpSs@Cluster reserves one core for network functionalities. FREDDO outperforms OmpSs@Cluster, especially for the configurations with large number of nodes. For the 32-node configuration on CyTera, FREDDO is up to 3.18× faster than OmpSs@Cluster (2.20× faster on average). The reason is twofold. First, OmpSs determines the dependencies at runtime. This eases programmability since programmers only need to annotate the sequential code with compiler

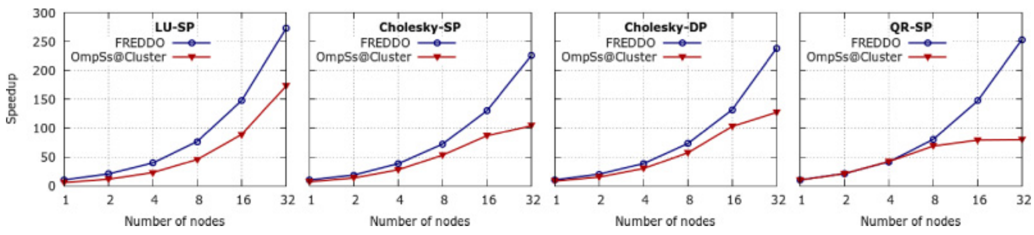


Fig. 12. FREDDO+MPI versus OmpSs@Cluster on CyTera (MS: $60K \times 60K$).

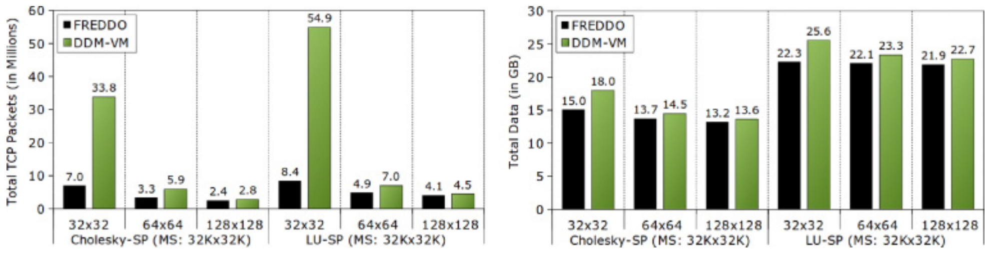


Fig. 13. Network traffic analysis: FREDDO versus DDM-VM on AMD using four nodes.

directives. However, it increases runtime overheads. In FREDDO, the dependencies are provided by programmers (through Update commands), which increases the programming effort but runtime overheads are reduced. Second, the execution model of OmpSs@Cluster is based on a master-worker design where the master is responsible for (i) assigning tasks to the remote nodes and (ii) preserving data coherency. OmpSs@Cluster aims to create an identical address space on each node, which gives the view of a single distributed address space [16], as in FREDDO. However, a master-worker scheme suffers from scalability issues on large clusters due to the bottleneck constituted by the master. Although OmpSs supports task nesting on cluster nodes to reduce the pressure on the master node, our results show that FREDDO scales better. FREDDO implements a peer-to-peer network (a node can send data and Updates to any other node) and a lightweight distribution scheme based on static mapping. It also uses data forwarding to reduce latencies.

5.5 Network Traffic Analysis

To study the efficiency of the proposed mechanisms for reducing the network traffic, during a distributed data-driven execution, we performed a traffic analysis for both FREDDO and DDM-VM. We are comparing our system to DDM-VM since the latter does not provide any mechanisms for reducing the network traffic in DDM applications. The experiments were conducted on the AMD system for two benchmarks from our benchmark suite: *Cholesky* and *LU* (SP versions). For our experiments, a root access was required for capturing the network traffic. Thus, the AMD system was used since it is the only system where we have root access. Figure 13 depicts the total TCP packets (in millions) and the total data (in gigabytes) that are exchanged between the nodes of the AMD system, for the 4-node configuration and the largest problem size ($32K \times 32K$). The benchmarks were executed with three different tile sizes: 32×32, 64×64, and 128×128. For the traffic analysis experiments, we used the *TShark* tool [38] (V2.2.3) and configured it to capture the traffic that is exchanged between the TCP ports that were reserved for the internode communication. It is important to note that in FREDDO, the size of the Context values is set to 64 bit, whereas DDM-VM supports only 32-bit Context values. Larger Context values allow execution of benchmarks

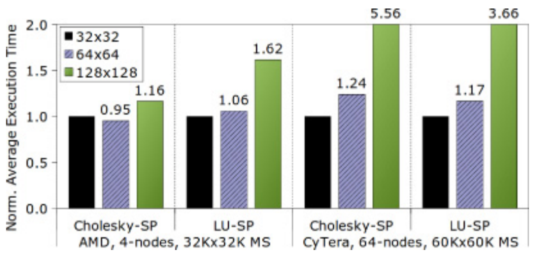


Fig. 14. Tile size effect on the AMD and CyTera systems using FREDDO.

with large problem sizes and fine-grain threads [46] (e.g., the *LU* benchmark on the CyTera system with $60K \times 60K$ matrix size and 32×32 tile size).

The comparison results show that FREDDO reduces the total TCP packets and data, especially for the smallest tile size where the frequency of the communication is increased between the nodes of the system. In the case of *Cholesky*, FREDDO reduces the total TCP packets by 4.85×, 1.79×, and 1.16× for the 32×32, 64×64, and 128×128 tile sizes, respectively. In the case of *LU*, FREDDO reduces the total TCP packets by 6.55×, 1.44×, and 1.11× for the 32×32, 64×64, and 128×128 tile sizes, respectively. Furthermore, FREDDO reduces the total amount of data by 16.7%, 5.5%, and 2.9% for *Cholesky* and 12.9%, 5.2%, and 3.5% for *LU* for the 32×32, 64×64, and 128×128 tile sizes, respectively. It is easy to observe that the total number of TCP packets and the total amount of data are not reduced with the same ratio. This is because the largest percentage of the total amount of data consists of the computed matrix tiles that are forwarded from the producer to consumer nodes. This percentage is approximately the same in both frameworks, as FREDDO reduces the network traffic mainly through optimizations on sending Update operations. The number of Update operations is high in benchmarks with high-complexity dependency graphs, and thus a higher number of TCP packets is used to carry such operations in DDM-VM.

Although the proposed mechanisms for reducing the network traffic are performing better for relatively small tile sizes (fine-grain threads), we expect to have a high positive impact on benchmarks that run on HPC systems with a large number of cores/nodes. Future HPC systems (e.g., exascale architectures) are projected to have up to billions of fine-grain threads executing asynchronously [5, 33, 56]. As a very small indication, in Figure 14, we provide the normalized average execution time of the *LU* and *Cholesky* benchmarks that are executed on both systems for three different tile sizes. The timings are normalized based on the execution time of the 32×32 tile size. The results show that larger tile sizes (i.e., coarse-grain threads) can negatively affect the performance, especially in the CyTera system with the 64-node configuration. We would like to note that we have tested experiments with even smaller tile sizes (e.g., 16×16) and the performance was not good, as expected, especially on the AMD system. This is because smaller tile sizes can increase the runtime overheads since more DThread instances will be created that stress the TSU. However, smaller tile sizes significantly increase the Update operations, and thus our mechanisms may further reduce the network traffic compared to the DDM-VM system.

6 RELATED WORK

MPI [26] is a de facto standard for programming distributed systems. It provides low-level communication primitives to transfer data among a set of processes running on the nodes of a distributed system. Although MPI allows the achievement of high performance, it requires a lot of effort from the programmer (partitioning of data and computations, movement of data during program execution, coherence, etc.). This is in contrast to our approach, which automates most of these tasks.

Thread Building Blocks (TBB) is an API developed by Intel that relies on C++ templates to facilitate parallel programming [55]. Since TBB 4.0, unstructured parallelism, dependency graphs, and dataflow algorithms can be expressed with *flow graph* classes and functions [30]. Unlike FREDDO, TBB's thread management does not naturally extend across the network.

Gupta and Sohi [27] introduced a software system that allows dataflow execution of sequential imperative programs on multicore systems. They implemented a C++ runtime library that exploits functional-level parallelism by executing functions on the cores in a dataflow fashion. Its execution model determines data dependences between functions dynamically and executes them concurrently in a dataflow fashion. In contrast, FREDDO applies its techniques at nonblocking threads. In addition, Gupta and Sohi's system does not support distributed data-driven execution.

Trebuchet [4] is a hybrid Von Neumann/dataflow execution system based on TALM. TALM—an architecture and language for multithreading—provides a user-defined coarse-grain parallel dataflow model, where programmers identify code blocks, called *superinstructions*, to be run in parallel and connect them in a dataflow graph. Couillard [41] is a C-compiler designed to create, based on an annotated C-program, a dataflow graph and C-code corresponding to each *superinstruction*. Currently, Trebuchet does not allow dataflow execution on distributed systems.

Although hardware dataflow implementations can achieve higher performance than software runtime systems [49], in this work we provide a software implementation to utilize the resources of commodity multicore systems. Examples of hardware implementations are WaveScalar [59], DDM's architectural support [43, 44], and Maxeler's dataflow engines [51].

FastFlow [3] is a structured parallel programming framework originally designed for shared-memory multicore/manycore platforms. It is implemented as a stack of C++ template libraries that allow programmers to use predefined and customizable parallel design patterns (also known as algorithmic skeletons). They include stream-oriented, data-parallel, and dataflow patterns. FastFlow was extended to support distributed systems [2]. In a distributed FastFlow application, multiple lower-tier graphs are connected together using a communication pattern that implements a network channel (point-to-point, broadcast, etc.). In FREDDO, we have a global graph, distributed across the nodes, where the internode communication is managed implicitly based on data-driven principles.

SWARM [40] is a software runtime that uses an execution model based on codelets [61]. It divides a program into tasks with runtime dependencies and constraints that can be executed when all runtime dependencies and constraints are met. SWARM's runtime schedules the tasks for execution based on resource availability and utilizes a work-stealing approach for on-demand load balancing. Both SWARM and FREDDO allow data-driven execution on distributed systems and adopt the static dependency resolution. FREDDO, in contrast to SWARM, handles the flow of data between producers and consumers running on different nodes, automatically.

OmpSs [16, 17] is a variant of OpenMP extended to support asynchronous task parallelism on clusters of heterogeneous architectures. It aims programming productivity and uses an annotation-based programming model to move data across a disjoint address space. Programmers annotate the sequential code with compiler directives that are translated into calls to the Nanos++ runtime library. Nanos++ schedules tasks on the available resources of a cluster and manages data coherence and data movement transparently. In OmpSs, the task dependency graph is always built at runtime, and thus this approach may introduce extra overheads. Moreover, OmpSs exposes only a part of the dependency graph available to the runtime; consequently, only a fraction of the concurrency opportunities in the applications is visible at any time.

PaRSEC [14, 22] is a task-based dataflow-driven framework designed to achieve HPC on heterogeneous HPC systems. It supports the parameterized task graph (PTG) model where dependencies between tasks and data are expressed using a domain-specific language named *JDF*. The PaRSEC runtime combines the information included in the PTG with supplementary information provided by the user (distribution of data onto nodes, priorities, etc.) to allow efficient data-driven scheduling. Both PaRSEC and FREDDO perform a static work distribution between nodes. PaRSEC performs dynamic work stealing within each node where FREDDO distributes the DThread instances to the kernels with the least amount of work.

Charm++ [34] is a C++-based parallel object-oriented language in which a runtime system decides what computation to execute next based on the availability of data for various objects, called *chares*. In a Charm++ application, the computation consists of a large number of chares distributed on available processors of the system and interacting with each other via asynchronous method invocations. The parallel components of the Charm++ applications have to be described/declared in separate *charm interface* (ci) description files. These files adopt a C++-like declaration syntax with several additional keywords and are compiled by a compiler wrapper called *charmc*.

Concurrent Collections (CnC) [35] is a declarative parallel programming language, with similar semantics to DDM, which allows programmers who lack experience in parallelism to express their parallel programs as a collection of high-level computations called *steps* that communicate via single-assignment data-structures called *items*. Steps and items are uniquely identified by *tags*. CnC steps correspond to DThreads, as both represent the unit of execution and apply single-assignment across steps/threads while allowing side effects locally within a step/thread. The control and data dependence relationships among the steps, manifested in the items and tags that are produced and consumed, correspond to the producer-consumer relationships of DThreads. Recently, CnC was extended to support distributed execution on clusters of multicore architectures [57].

Open Community Runtime (OCR) [39, 47] and HPX [33] are task-based frameworks aimed at HPC systems. In OCR, a program is a collection of asynchronously executing tasks operating on relocatable data blocks where *events* coordinate the execution of OCR tasks. Tasks, data blocks, and events are globally addressable throughout the system via globally unique IDs (GUIDs). HPX is a C++11 runtime system based on the ParalleX execution model [32]. It facilitates distributed operations and enables fine-grain task parallelism. HPX provides an active global address space (AGAS), where globally addressable objects can be tracked by a unique global ID as they are migrated throughout the system.

Sucuri [58] is a Python library that allows transparent dataflow execution on computer clusters. Each node of the cluster has its own scheduler that uses a ready queue with a first-come first-served policy to serve idle workers. The dataflow graph of Sucuri applications is partitioned before execution, using an algorithm based on list scheduling. In FREDDO, the dataflow graph is partitioned based on the Context values of the DThread instances.

Several programming models have been proposed that try to facilitate the programmability of distributed systems by providing a global address space view of the aggregated memories of all nodes in a system. Such a view facilitates porting sequential applications and reduces the complexity of distributed applications. Examples of such models include UPC [18], X10 [21], and Chapel [20]. Although such models ease the burden of the programmer, achieving good performance still requires a nontrivial effort from programmers to deal with data distribution and coherency. Moreover, such models require the support of special compilers.

7 CONCLUSIONS AND FUTURE WORK

In this work, we presented a portable and efficient implementation of the DDM model that enables efficient data-driven scheduling on distributed multicore architectures. The proposed framework was implemented as an extension to FREDDO—a C++ implementation of the DDM model. Experiments were performed on two distributed systems with 128 and 768 cores, respectively. Our evaluation analysis shows that the distributed FREDDO implementation scales well and achieves comparable or better performance when compared to other systems, such as MPI, DDM-VM, and OmpSs@Cluster. Furthermore, we proposed simple and efficient techniques for reducing network traffic during the execution of DDM applications. Our experiments on the 128-core system show that FREDDO can reduce the total amount of TCP packets by up to 6.55× and the total amount of data by up to 16.7% when compared to the DDM-VM system.

Future work will be focused on applying data-driven scheduling on heterogeneous HPC systems. We are interested in manycore accelerators with software-controlled scratch-pad memories such as the Sunway SW26010 processor [24]. Deterministic data prefetching into scratch-pad memories using data-driven techniques can improve the locality of sequential processing [8]. We believe that this approach can further improve the performance of HPC systems. We are also interested in other types of manycore accelerators, such as the Intel Xeon Phi and GPUs. Currently, FREDDO provides an API for programmability. We plan to provide high-level tools for better programmability, such as compilers (e.g., DDMCPP [60]) and declarative parallel programming languages (e.g., CnC [35]). Finally, it would be interesting to evaluate the DSM overheads in FREDDO; run experiments in larger HPC systems; and evaluate FREDDO using larger scientific applications, such as LULESH, NWChem, and Nek5000.

ACKNOWLEDGMENTS

The authors would like to thank the Cyprus Institute and the CyTera HPC facility for providing the hardware resources (Intel nodes) used in the evaluation of this work.

REFERENCES

[1] Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Stanimire Tomov. 2009. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *Journal of Physics: Conference Series* 180, 1, 012037.

[2] Marco Aldinucci, Sonia Campa, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. 2013. Targeting distributed systems in FastFlow. In *Euro-Par 2012: Parallel Processing Workshops*. Lecture Notes in Computer Science, Vol. 7640. Springer, 47–56.

[3] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. 2012. FastFlow: High-level and efficient streaming on multi-core. In *Programming Multi-Core and Many-Core Computing Systems*, S. Pllana (Ed.). John Wiley & Sons, 13.

[4] Tiago A. O. Alves, Leandro A. J. Marzulo, Felipe M. G. França, and Vítor Santos Costa. 2011. Trebuchet: Exploring TLP with dataflow virtualisation. *International Journal of High Performance Systems Architecture* 3, 2-3, 137–148.

[5] Saman Amarasinghe, Mary Hall, Richard Lethin, Keshav Pingali, Dan Quinlan, Vivek Sarkar, John Shalf, et al. 2011. Exascale programming challenges. In *Proceedings of the Workshop on Exascale Programming Challenges*.

[6] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, Jack J. Dongarra, J. Du Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen. 1999. *LAPACK Users' Guide* (3rd ed.). Society for Industrial and Applied Mathematics, Philadelphia, PA.

[7] Samer Arandi and Paraskevas Evripidou. 2010. Programming multi-core architectures using data-flow techniques. In *Proceedings of the 2010 International Conference on Embedded Computer Systems (SAMOS'10)*. IEEE, Los Alamitos, CA, 152–161.

[8] Samer Arandi and Paraskevas Evripidou. 2011. DDM-VMc: The data-driven multithreading virtual machine for the cell processor. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*. ACM, New York, NY, 25–34.

[9] Arvind and Gostelow. 1982. The U-interpreter. *Computer* 15, 2, 42–49.

[10] Arvind and Robert A. Iannucci. 1988. Two fundamental issues in multiprocessing. In *Proceedings of the 4th International DFVLR Seminar on Foundations of Engineering Sciences on Parallel Computing in Science and Engineering*. Springer, New York, NY, 61–88.

[11] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*. ACM, New York, NY, 72–81.

[12] L. S. Blackford, J. Choi, A. Cleary, E. D'Azeuedo, J. Demmel, I. Dhillon, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. 1997. *ScaLAPACK User's Guide.* Society for Industrial and Applied Mathematics, Philadelphia, PA.

[13] Dan Bonachea. 2002. *GASNet Specification, v1.1.* Technical Report UCB/CSD-02-1207. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2002/5764.html.

[14] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Herault, Pierre Lemarinier, and Jack Dongarra. 2012. DAGuE: A generic distributed DAG engine for high performance computing. *Parallel Computing* 38, 1, 37–51.

[15] BSC. 2017. BSC Application Repository. Retrieved November 21, 2017, from https://pm.bsc.es/projects/bar/wiki/Applications.

[16] Javier Bueno, Luis Martinell, Alejandro Duran, Montse Farreras, Xavier Martorell, Rosa M. Badia, Eduard Ayguade, and Jesús Labarta. 2011. Productive cluster programming with OmpSs. In *Proceedings of the European Conference on Parallel Processing*. 555–566.

[17] Javier Bueno, Xavier Martorell, Rosa M. Badia, Eduard Ayguadé, and Jesús Labarta. 2013. Implementing OmpSs support for regions of data in architectures with multiple address spaces. In *Proceedings of the 27th International ACM Conference on Supercomputing.* ACM, New York, NY, 359–368.

[18] William W. Carlson, Jesse M. Draper, David E. Culler, Kathy Yelick, Eugene Brooks, and Karen Warren. 1999. *Introduction to UPC and Language Specification.* Technical Report. CCS-TR-99-157, IDA Center for Computing Sciences, Bowie, MD.

[19] Márcia C. Cera, João V. F. Lima, Nicolas Maillard, and Philippe Olivier Alexandre Navaux. 2010. Challenges and issues of supporting task parallelism in MPI. In *Proceedings of the 2010 EuroMPI Conference.* Springer, 302–305.

[20] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. 2007. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications* 21, 3, 291–312.

[21] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph Von Praun, and Vivek Sarkar. 2005. X10: An object-oriented approach to non-uniform cluster computing. *ACM SIGPLAN Notices* 40, 519–538.

[22] Anthony Danalis, Heike Jagode, George Bosilca, and Jack Dongarra. 2015. PaRSEC in practice: Optimizing a legacy chemistry application through distributed task-based execution. In *Proceedings of the 2015 IEEE International Conference on Cluster Computing (CLUSTER'15).* IEEE, Los Alamitos, CA, 304–313.

[23] Edsger W. Dijkstra and Carel S. Scholten. 1980. Termination detection for diffusing computations. *Information Processing Letters* 11, 1, 1–4.

[24] Haohuan Fu, Junfeng Liao, Jinzhe Yang, Lanning Wang, Zhenya Song, Xiaomeng Huang, Chao Yang, et al. 2016. The Sunway Taihulight supercomputer: System and applications. *Science China Information Sciences* 59, 7, 072001.

[25] Samuel H. Fuller and Lynette I. Millett. 2011. Computing performance: Game over or next level? *Computer* 44, 1, 31–38.

[26] William Gropp, Ewing Lusk, and Anthony Skjellum. 1999. *Using MPI: Portable Parallel Programming With the Message-Passing Interface*, Vol. 1. MIT Press, Cambridge, MA.

[27] Gagan Gupta and Gurindar S. Sohi. 2011. Dataflow execution of sequential imperative programs on multicore architectures. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture.* ACM, New York, NY, 59–70.

[28] J. R. Gurd, C. C. Kirkham, and I. Watson. 1985. The Manchester prototype dataflow computer. *Communications of the ACM* 28, 1, 34–52.

[29] The Cyprus Institute. 2017. Cy-Tera. Retrieved November 21, 2017, from http://web.cytera.cyi.ac.cy.

[30] Intel. 2017. Flow Graph. Retrieved November 21, 2017, from https://software.intel.com/en-us/node/506211.

[31] Wesley M. Johnston, J. R. Hanna, and Richard J. Millar. 2004. Advances in dataflow programming languages. *ACM Computing Surveys* 36, 1, 1–34.

[32] Hartmut Kaiser, Maciek Brodowicz, and Thomas Sterling. 2009. ParalleX: An advanced parallel execution model for scaling-impaired applications. In *Proceedings of the 2009 International Conference on Parallel Processing Workshops (ICPPW'09).* IEEE, Los Alamitos, CA, 394–401.

[33] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. 2014. HPX: A task based programming model in a global address space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models.* ACM, New York, NY, 6.

[34] Laxmikant V. Kale and Sanjeev Krishnan. 1996. Charm++: Parallel programming with message-driven objects. In *Parallel Programming Using C++,* G. V. Wilson and P. Lu (Eds.). MIT Press, Cambridge, MA, 175–213.

[35] Kathleen Knobe. 2009. Ease of use with Concurrent Collections (CnC). In *Proceedings of the 1st USENIX Conference on Hot Topics in Parallelism (HotPar'09).* 17.

[36] David A. Koufaty, Xiangfeng Chen, David K. Poulsen, and Josep Torrellas. 1996. Data forwarding in scalable shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems* 7, 12, 1250–1264.

[37] C. Kyriacou, P. Evripidou, and P. Trancoso. 2006. Data-driven multithreading using conventional microprocessors. *IEEE Transactions on Parallel and Distributed Systems* 17, 10, 1176–1188.

[38] Ulf Lamping and Ed Warnicke. 2004. Wireshark user's guide. *Interface* 4, 6.

[39] Joshua Landwehr, Joshua Suetterlein, Andrés Márquez, Joseph Manzano, and Guang R. Gao. 2016. Application characterization at scale: Lessons learned from developing a distributed open community runtime system for high performance computing. In *Proceedings of the ACM International Conference on Computing Frontiers*. ACM, New York, NY, 164–171.

[40] Christopher Lauderdale, Mark Glines, Jihui Zhao, Alex Spiotta, and Rishi Khan. 2013. *SWARM: A Unified Framework for Parallel-For, Task Dataflow, and Distributed Graph Traversal*. ET International Inc., Newark, NJ.

[41] Leandro A. J. Marzulo, Tiago A. O. Alves, Felipe M. G. França, and Vítor Santos Costa. 2014. Couillard: Parallel programming via coarse-grained data-flow compilation. *Parallel Computing* 40, 10, 661–680.

[42] George Matheou. 2017. FREDDO Project. Retrieved November 21, 2017, from https://github.com/george-matheou/freddo-project.

[43] George Matheou and Paraskevas Evripidou. 2013. Verilog-based simulation of hardware support for data-flow concurrency on multicore systems. In *Proceedings of the 2013 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII)*. IEEE, Los Alamitos, CA, 280–287.

[44] George Matheou and Paraskevas Evripidou. 2015. Architectural support for data-driven execution. *ACM Transactions on Architecture and Code Optimization* 11, 4, Article 52, 25 pages.

[45] George Matheou and Paraskevas Evripidou. 2016. FREDDO: An efficient framework for runtime execution of data-driven objects. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'16)*. 265–273.

[46] George Matheou and Paraskevas Evripidou. 2016. *FREDDO: An Efficient Framework for Runtime Execution of Data-Driven Objects*. Technical Report TR-16-1. Department of Computer Science, University of Cyprus, Nicosia, Cyprus. https://www.cs.ucy.ac.cy/docs/techreports/TR-16-1.pdf.

[47] Timothy G. Mattson, Romain Cledat, Vincent Cavé, Vivek Sarkar, Zoran Budimlić, Sanjay Chatterjee, Josh Fryman, et al. 2016. The open community runtime: A runtime system for extreme scale computing. In *Proceedings of the 2016 IEEE High Performance Extreme Computing Conference (HPEC'16)*. IEEE, Los Alamitos, CA, 1–7.

[48] George Michael, Samer Arandi, and Paraskevas Evripidou. 2013. Data-flow concurrency on distributed multi-core systems. In *Proceedings of the 2013 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'13)*.

[49] Andrea Mondelli, Nam Ho, Alberto Scionti, Marco Solinas, Antoni Portero, and Roberto Giorgi. 2015. Dataflow support in x86_64 multicore architectures through small hardware extensions. In *Proceedings of the 2015 Euromicro Conference on Digital System Design (DSD'15)*. IEEE, Los Alamitos, CA, 526–529.

[50] mpich.org. 2017. LU Factorization. Retrieved November 21, 2017, from https://trac.mpich.org/projects/armci-mpi/browser/tests/contrib/lu/lu.c.

[51] Oliver Pell, Oskar Mencer, Kuen Hung Tsoi, and Wayne Luk. 2013. Maximum performance computing with dataflow engines. In *High-Performance Computing Using FPGAs*. Springer, 747–774.

[52] Judit Planas, Rosa M. Badia, Eduard Ayguadé, and Jesus Labarta. 2009. Hierarchical task-based programming with StarSs. *International Journal of High Performance Computing Applications* 23, 3, 284–299.

[53] David K. Poulsen and Pen-Chung Yew. 1994. Data prefetching and data forwarding in shared memory multiprocessors. In *Proceedings of the 1994 International Conference on Parallel Processing (ICPP'94)*, Vol. 2. IEEE, Los Alamitos, CA, 280.

[54] Jelica Protic, Milo Tomasevic, and Veljko Milutinović. 1998. *Distributed Shared Memory: Concepts and Systems*, Vol. 21. John Wiley & Sons.

[55] James Reinders. 2007. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly Media, Inc.

[56] S. Ashby, P. Beckman, J. Chen, P. Colella, B. Collins, D. Crawford, J. Dongarra, et al. 2010. *The Opportunities and Challenges of Exascale Computing*. Summary Report of the Advanced Scientific Computing Advisory Committee (ASCAC) Subcommittee. U.S. Department of Energy Office of Science, Washington, DC.

[57] Frank Schlimbach, James C. Brodman, and Kath Knobe. 2013. Concurrent Collections on distributed memory theory put into practice. In *Proceedings of the 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP'13)*. IEEE, Los Alamitos, CA, 225–232.

[58] Rafael J. N. Silva, Brunno Goldstein, Leandro Santiago, Alexandre C. Sena, Leandro A. J. Marzulo, Tiago A. O. Alves, and Felipe M. G. França. 2016. Task scheduling in Sucuri dataflow library. In *Proceedings of the 2016 International Symposium on Computer Architecture and High Performance Computing Workshops*. IEEE, Los Alamitos, CA, 37–42.

[59] Steven Swanson, Ken Michelson, Andrew Schwerin, and Mark Oskin. 2003. WaveScalar. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, Los Alamitos, CA, 291.

[60] Pedro Trancoso, Kyriakos Stavrou, and Paraskevas Evripidou. 2007. DDMCPP: The data-driven multithreading C pre-processor. In *Proceedings of the 11th Workshop on the Interaction Between Compilers and Computer Architectures*. 32.

[61] Stéphane Zuckerman, Joshua Suetterlein, Rob Knauerhase, and Guang R. Gao. 2011. Using a "codelet" program execution model for exascale machines: Position paper. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era (EXADAPT'11)*. 64–69.

Received May 2017; revised October 2017; accepted November 2017