Data-Driven Concurrency for High Performance Computing

George Matheou
Paraskevas Evripidou

TR-17-1 **May 2017**

University of Cyprus

Technical Report

Data-Driven Concurrency for High Performance Computing

George Matheou

UCY-CS-TR-17-1

May 2017

Table of Contents

Abstract

In this work we utilize data-flow/data-driven techniques in order 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, based on the Data-Driven Multithreading (DDM) model, that enables data-driven concurrency on HPC systems. DDM is a hybrid control-flow/data-flow model of execution that schedules threads based on data availability on sequential processors. The proposed framework is provided as an extension to FREDDO, a C++ implementation of the DDM model, that supports data-driven execution on single-node multi-core architectures. The proposed framework has been evaluated using a suite of eight 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 distributed FREDDO implementation scales well and tolerates scheduling overheads and memory latencies effectively. We also compare our framework with MPI/ScaLAPACK, SWARM and DDM-VM. The comparison results show that the proposed framework obtains comparable or better performance. Particularly, on the 768-core system, our framework is up to 1.61× *faster than the MPI/ScaLAPACK implementation which is based solely on the sequential model of execution. Additionally, on the 128-core system, the proposed framework is up to* 1.26× *faster than SWARM and up to* 1.42× *faster than DDM-VM.*

Index terms— Data-Driven Multithreading; Distributed Execution; Runtime System; High Performance Computing

1 Introduction

The end of the exponential growth of the sequential processors has facilitated the development of multi-core and many-core architectures. Such architectures have dominated the High Performance Computing (HPC) field, from shared memory systems to large-scale distributed memory clusters. Programming of such systems is mainly done through parallel extensions of the sequential model like MPI [1] and OpenMP [2]. These extensions do facilitate high productivity parallel programming, but also suffer from the inability to tolerate long memory and synchronization latencies [3, 4, 5, 6]. As a result, the high number of computational resources of the current HPC systems is not utilized efficiently [7].

Indeed, realistic applications running on current supercomputers typically use only 5%-10% of the machine's peak processing power at any given time [4]. Even worse, as the number of cores will inevitably be increased in the coming years, the fraction that can be kept busy at any given time can be expected to plummet [4]. As such, new programming/execution models need to be developed in order to utilize efficiently the resources of the current and future HPC systems. Such a model is the *data-flow model* of execution [8, 9, 10]. Data-flow is a formal model that is inherently parallel, functional and asynchronous [8, 11]. It enforces only a partial ordering, as dictated by the true data-dependencies, which is the minimum synchronization possible. This is very beneficial for parallel processing since it allows to exploit the maximum potential parallelism [12]. Moreover, programming models and architectures, based on the data-flow principles, can handle concurrency and tolerate memory and synchronization latencies efficiently [3].

In this work we present and evaluate a programming framework that supports data-flow/data-driven concurrency on HPC systems. The proposed framework is based on the Data-Driven Multithreading (DDM) model of execution [13]. DDM is a hybrid control-flow/data-flow model of execution 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. The proposed programming framework extends the FREDDO framework [14], a C++ implementation of DDM, that targets single-node multi-core systems.

The distributed FREDDO implementation provides (i) a software Distributed Shared Memory (DSM) system [15] with a shared Global Address Space, (ii) an implicit memory management based on data-driven semantics using *data forwarding* techniques [16] and (iii) an implicit distributed termination detection algorithm [17]. This allows the distributed DDM/FREDDO applications to be fundamentally the same as the single-node ones. This is because complicated tasks, such as, partitioning of data and computations, movement of data during execution and preserving coherency/consistency, are provided automatically. Furthermore, the proposed implementation provides an efficient distributed data-driven scheduling optimized for multithreaded architectures. This is achieved by implementing a lightweight two-layer scheduling mechanism based on the DDM's tagging system where scheduling operations, computations and network functionalities are operated asynchronously. The contributions presented in this paper are the following:

1. Provide an efficient, portable and robust distributed implementation of the DDM model. All the required data structures and mechanisms have been designed and implemented in order 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 local 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 multi-core systems with up to 24 cores using the DDM-VM implementation [18]. For the evaluation of the distributed FREDDO implementation we have used a benchmark suite consisting of eight benchmarks with different characteristics: embarrassingly parallel and recursive algorithms as well as benchmarks with high complexity Dependency Graphs. The performance evaluation shows that FREDDO scales well on both systems. Particularly, on the AMD system for the 4-node configuration and the largest problem size, FREDDO achieves up to 93% of the ideal speedup with an average of 82%. On the CyTera system, for the 64-node configuration and the largest problem size, FREDDO achieves up to 84% of the ideal speedup with an average of 67%.

4. The comparison of the results obtained from the FREDDO framework and two state-of-the-art frameworks, the ScaLAPACK [19] and SWARM [20], for the *Cholesky* benchmark. FREDDO is also compared with DDM-VM [18], the first distributed implementation of the DDM model, for the *Cholesky* and LU benchmarks. ScaLAPACK is an MPI-based library that provides high-performance linear algebra routines for parallel distributed memory machines while DDM-VM and SWARM are software runtime systems that allow data-driven concurrency on distributed multi-core systems. The comparison results show that FREDDO achieves similar or better performance.

5. Provide simple mechanisms/optimizations to reduce the network traffic of a distributed DDM execution. 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 is compared with the DDM-VM system.

6. The implementation of a connectivity layer with two different network interfaces: a Custom Network Interface (CNI) and MPI [1]. The CNI support allows to have a direct and fair comparison with frameworks that also utilize a custom network interface (e.g., DDM-VM and SWARM) where the MPI support provides portability and flexibility to the FREDDO framework. Finally, we provide comparison results for CNI and MPI for several benchmarks.

The remainder of this paper 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 paper.

2 Data-Driven Multithreading

DDM [13] is a non-blocking 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 allows multiple instances of the same DThread to co-exist 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) [21] which is responsible for scheduling DThreads at run-time based on data-availability. For each DThread, the TSU collects meta-data (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 its producer-instances have completed their execution. Finally, the DDM model was evaluated by several software [22, 23, 18, 14] and hardware [24, 25] implementations. Although hardware data-flow/data-driven implementations can achieve higher performance than software runtime systems [25, 26], in this work, we provide a software implementation in order to utilize the resources of the current/commercial HPC systems.

2.1 Single-node FREDDO implementation

FREDDO [14, 27] is an efficient object-oriented implementation of the DDM model [13]. It is a C++11 framework that supports data-driven execution on conventional single-node multi-core architectures. FREDDO provides a C++ API (Application Programming Interface) which includes a set of runtime functions and classes that help the programmers to develop DDM applications. The API provides the basic functionalities for parallelizing loops (up to three-level nested loops), recursive functions and simple function calls with data dependencies. In FREDDO, a program consists of DThreads which 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 pinned on a specific core until the end of the DDM execution. This eliminates the overheads of the context-switching between the Kernels in the system. The Kernel is responsible for executing the ready DThread instances that are received from the TSU. The Runtime system enables the communication between the 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.

The FREDDO's Dependency Graph The Dependency Graph is a directed graph in which the nodes represent the DThread instances and the arcs represent the data-dependencies amongst 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-T4). Notice that the number inside each node indicates its Context value. T1 is a SimpleDThread object, i.e. it has only one instance. T2 and T3 are MultipleDThread objects. A MultipleDThread object manages a DThread with multiple instances and can be used to parallelize one-level loops (or similar constructs). T4 is a MultipleDThread2D object, a special type of MultipleDThread, where the Context values consist of two parts, the outer and the inner. A MultipleDThread2D object can be used to parallelize a two-level nested loop where each Context value will hold an index of the outer and the inner loop. In this example T4 consists of 64 instances (with Contexts from <0,0> to <7,7>). Similarly, a user can parallelize a three-level nested loop by using a MultipleDThread3D object where its Context values consist of three parts: outer, middle and inner.

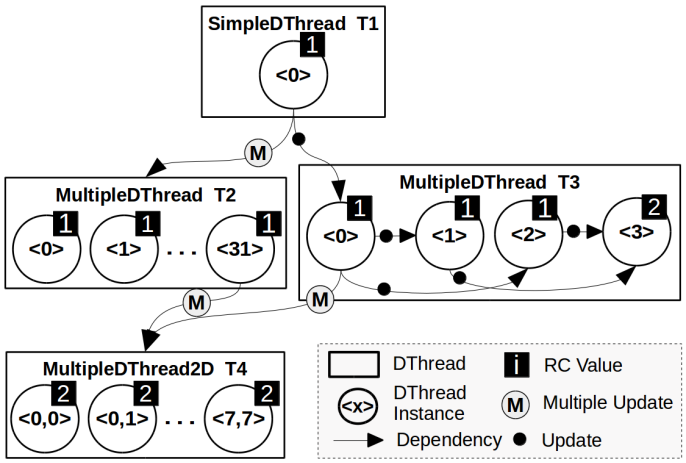


Figure 1: Example of a Dependency Graph.

The RC values are depicted as shaded values next to the nodes. For example, the instance <3> of T3 has RC=2 because it has two producers, the instances <1> and <2> of T3. All the instances of T2 have RC=1 because they are waiting for only one producer, the instance <0> of T1. 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 FREDDO, the operation used for decreasing the RC value is called *Update*. Update operations can be considered as tokens that are moving from the producer to consumer instances through the arcs of the graph. *Multiple Updates* are introduced in order to decrease multiple RC values of a DThread at the same time. This reduces the number of tokens in the graph. For instance, DThread T1 sends a Multiple Update command to DThread T2 in order to spawn all its instances, instead of sending 32 single Updates. A DThread instance can send single and multiple Updates to any other instance of any type, including itself (the only requirement is to have data dependencies). Thus, it is possible to build very complex Dependency Graphs.

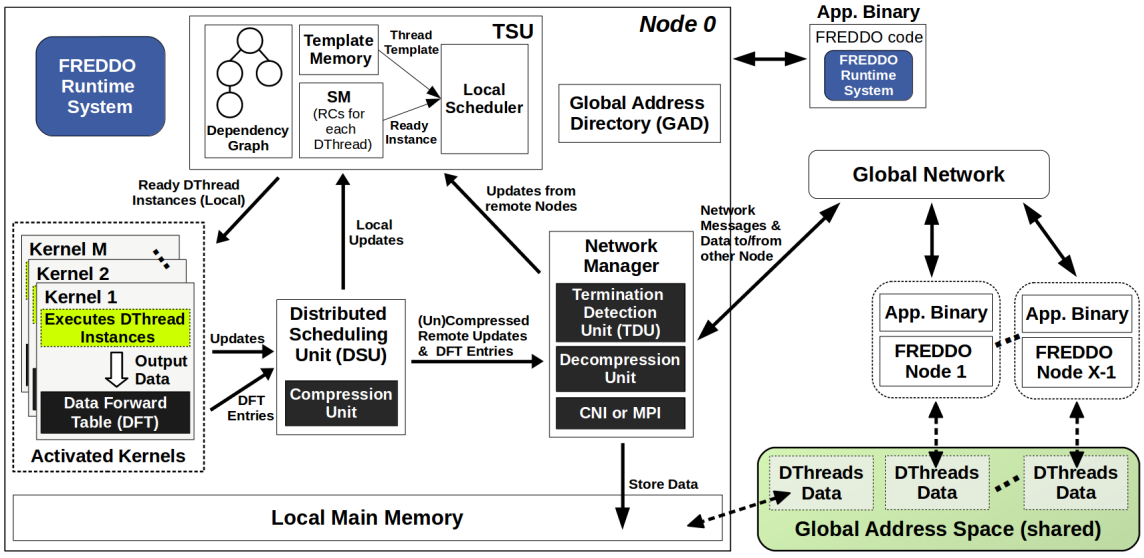


Figure 2: The FREDDO's Distributed Architecture.

3 FREDDO: Distributed Support

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

3.1 Distributed Architecture

Figure 2 depicts the distributed architecture of FREDDO. It is composed by multi-core nodes connected by a global network interconnect (e.g., Ethernet, InfiniBand, etc.). A *Network Manager* is implemented in each node, that abstracts the details of the network interconnect and allows the inter-node communication. The FREDDO's runtime system is responsible for handling the communication and data management across the nodes, for managing the applications' Dependency Graphs and for scheduling/executing the ready DThread instances on the cores of the entire system. The same application binary is executed on all the 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 Distributed Shared Memory (DSM) system [15] with a shared Global Address Space (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, 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 the nodes (from one local main memory to another) by using the *GAS_IDs*.

For preserving memory consistency we employ *data forwarding* [16]. Particularly, the produced 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 the data transfers are completed. To support this functionality, each Kernel is associated with a Data Forward Table (DFT). A DFT keeps track of the produced data segments of the currently executed DThread instance. When the instance finishes its execution, the runtime uses the DFT entries to transfer the produced data to the remote nodes, implicitly. The data-driven semantics of our model allow to have a lightweight memory consistency mechanism where expensive coherence operations 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 multi-core processor.

3.3 Scheduling Mechanisms

FREDDO provides a lightweight *distribution scheme*, based on the DDM's tagging system [13], to distribute and execute the DThread instances on the system's nodes. Particularly, the *mapping* of the DThread instances to the nodes is determined at compile time based on their Context values (the tags). However, the execution of the ready DThread instances is determined at runtime, based on data-availability. This approach simplifies the scheduling and data management operations as well as it reduces the runtime overheads. The FREDDO's distribution scheme is implemented by two different scheduling mechanisms, the *intra-node* and the *inter-node*.

3.3.1 Inter-node Scheduling Mechanism

The *inter-node* mechanism is handled by the *Distributed Scheduling Unit (DSU)* which decides if the 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 the data (Updates and consumed data) as network messages, via the FREDDO's Network Manager, to the corresponding remote node(s). In the latter case, the Update operations are sent to the local TSU.

3.3.2 Intra-node Scheduling Mechanism

The *intra-node* mechanism is handled by the TSU. The TSU 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). 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 and decrements the RC value in the Synchronization Memory. If the RC value of any DThread's instance reaches zero, then it is deemed executable and it is sent to the TSU's Local Scheduler. The Local Scheduler distributes the ready DThread instances to the Kernels in order 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 [14, 27].

3.4 Distributed Execution Termination

Detecting termination of data-driven programs in distributed execution environments it's not a straightforward procedure, since the availability of data governs the order of execution. In this work we have implemented an implicit distributed termination algorithm based on the Dijkstra and Scholten's parental responsibility algorithm [17], which requires minimal message exchange. The algorithm assumes termination when: *the state of all the 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 and pending ready DThread instances that are waiting for execution. When the parent node (*RootNode*) detects termination, it broadcasts a termination message to the other nodes and waits for their acknowledgements in order 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 in order to reduce the programming effort as well as to avoid introducing extra dependencies compared to explicit termination approaches [28]. Reducing dependencies can improve the performance of the data-driven programs since these extra dependencies result in extra TSU work and in more traffic in the network.

3.5 Network Manager

The Network Manager is responsible for handling the inter-node communication. It is implemented as a software module that relies on the underlying network hardware interface. The Network Manager establishes connections between the system's nodes, it exchanges network messages between the nodes and it processes the incoming network messages appropriately (e.g., it sends the Updates to the local TSU). It also supports data forwarding across the global address space.

3.5.1 Connectivity Layer

The Network Manager handles the low-level connectivity by utilizing two different network interfaces: a Custom Network Interface (CNI) which is an optimized implementation with TCP sockets, and the widely used MPI library [1]. As a first step, we have implemented the CNI in order to identify all the required functionalities and mechanisms that are needed for the inter-node communication. Based on these functionalities/mechanisms we have implemented the Network Manager

on top of MPI. The major benefits of providing MPI support are portability and flexibility. On the other hand, the CNI implementation allows to have a direct and fair comparison with similar frameworks that utilize a custom network interface (e.g., DDM-VM [18] and SWARM [20]). Furthermore, it allows us to explore the performance penalties of using MPI instead of CNI.

3.5.2 Sending/Receiving Functionalities

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 utilized by the Kernels (through the DSU) concurrently. This removes the cost of such operations from the TSU's critical path. For the receiving functionalities, an *auxiliary thread* is used, which continuously retrieves the incoming network messages from the other nodes.

3.6 Reducing the Network Traffic

Reducing the network traffic in HPC systems is critical since it can help to avoid network saturation and reduce the power consumption. The US Department of Energy (DOE) [29] 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.

3.6.1 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, like in [18]. 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 the basic types of commands. As a result, the number of sending network messages can be reduced to half.

3.6.2 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 3:❶ shows an example where a Multiple Update, with Contexts from <0> to <47>, is distributed to a 4-node system (each node has 4 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 3:❷). 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*.

3.6.3 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

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

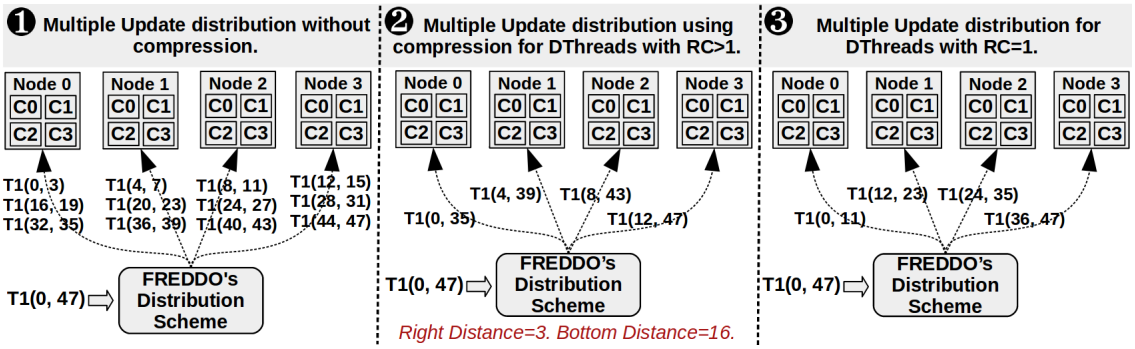


Figure 3: Example of reducing the network traffic generated by a Multiple Update. T1(X,Y) denotes a Multiple Update for DThread T1.

allocation [14]. Instances of such DThreads are scheduled immediately when Updates are received for them. This approach allows to schedule 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 3:❶ 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 3:❸. This methodology does not require compression and it is possible to reduce the number of messages by 75% for this specific example.

3.6.4 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, the 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

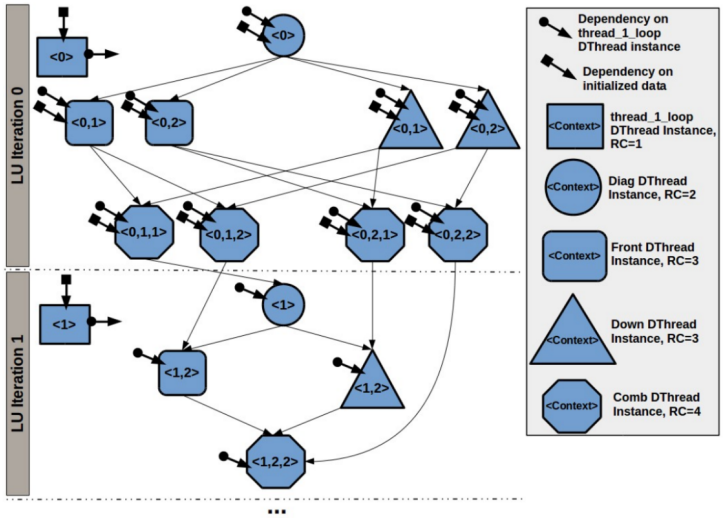
For supporting distributed execution of recursive algorithms in a data-driven manner, we have extended the functionalities of two FREDDO's DThread classes: *RecursiveDThread* and *ContinuationDThread* [14]. RecursiveDThread is a special class that allows parallelizing different types of recursive functions (linear, tail, etc.). It allocates/deallocates the arguments and the return values of the 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. In this case, the Network Manager will send the *DistRData* objects and the children-instances' Context values to the remote nodes in order 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 of the distributed FREDDO implementation by using a benchmark with a complex Dependency Graph, the tile LU Decomposition algorithm. The algorithm is based on an earlier version developed in StarSs [30] 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 4a 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*.

```
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]);
}
```

(a) Original Code.



(b) DDM Dependency Graph for the first two iterations of a 3×3 tile matrix (N=3).

Figure 4: Tile LU Decomposition.

In every iteration of the outermost loop, the *diag* operation takes as input the diagonal tile that corresponds to the iteration number to produce its new value. The *front* operation produces the remaining tiles on the same row as the diagonal tile. For each one of those tiles, it takes as input the result of the *diag* in addition to the current tile to produce its new value. Similarly, the *down* operation produces the remaining tiles on the same column as the diagonal tile. The *comb* operation produces the rest of the tiles for that LU iteration. For every tile it produces, it takes as input three tiles: the current tile, the tile produced by the front operation and the tile produced by the down operation. It multiplies the second and third tiles and adds the result to the first tile to produce the final resulting tile. This computational pattern is repeated in the next LU iteration on a subset of the resulting matrix that excludes the first row and column and continues for as much iterations as the diagonal tiles of the matrix.

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 below:

- 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. Particularly, 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 Dependency Graph shown in Figure 4b illustrates 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) has been selected. Each DThread's instance is labeled with the value of its Context.

```
#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();
}
```

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

4.2 FREDDO Code

Figure 5 depicts the FREDDO code for the tile LU Decomposition algorithm. In FREDDO, each DThread object is associated with a *DFunction* [14]. 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 the DThread class. In this example the code of each DThread has been placed in standard C/C++ functions.

Each call of an Update command in the DThreads' code corresponds to one dependency arrow in Figure 4b. A DThread object (DObject) provides several Single and Multiple Update methods that target itself or its consumers [14]. For example, the first Update operation of the loop_1_code DFunction indicates a Single Update which decrements the RC value of the instance kk of the diag_thread (diagDT) by one. The second Update operation of the same DFunction indicates a Multiple Update which decrements the RC value of multiple instances of the front_thread (frontDT) by one. Moreover, the Update operations at the end of the comb_code DFunction implement a switch actor, 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, the FREDDO's runtime registers the address of the tile matrix A in the *Global Address Directory* (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 the 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 4b that describe dependencies on initialized data. The initial Updates have to be executed one time only. In this example the *RootNode* has been selected to execute these updates which 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 the DDM scheduling and waits until the FREDDO's runtime detects the distributed execution termination (see Section 3.4). When the *run* function returns, all the resources allocated by the FREDDO framework are deallocated using the *finalize* function.

The 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 the user has 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. Additionally, for gathering the results in the *RootNode*, the user has to utilize 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's a part of the tile matrix A. In Figure 5, the required code for the distributed execution is highlighted.

As previously mentioned, the FREDDO's memory model creates an identical address space on each node, which gives the view of a single distributed address space. At the moment, 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 the distributed implementation of 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 the performance results and comparisons with other systems. Finally, we present network traffic analysis results.

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: 48 GB)	
- Clock Frequency	1.4 GHz	2.67 GHz
- Cores	16	12
- Hardware Threads/Core	2	1
- Total Hardware Threads	32	12
- L1-I$, L1-D$, L2$/Core	32 KB, 16 KB, 1 MB	32 KB, 32 KB, 256 KB
- L3$	24 MB	12 MB
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

Table 1: Systems used for benchmark evaluation.

5.1 Experimental Setup

For the experimental evaluation we have used two different systems, the AMD and CyTera. The AMD is a 4-node local system. The CyTera [31] is an open-access HPC system which provides up to 64 nodes per user. The specifications of the systems are shown in Table 1. Our benchmark suite contains three applications which require low communication between the nodes (*BMMULT*, *Blackscholes* and *Swaptions*), three benchmarks with complex Dependency Graphs that require heavy inter-node communication (*LU*, *QR* and *Cholesky*) and two recursive algorithms (*Fibonacci* and *PowerSet*) that require medium inter-node 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) [32].

3. *Swaptions* uses the Heath-Jarrow-Morton (HJM) framework to price a portfolio of swaptions [32]. 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 has been based on an earlier version written in StarSs [30].

5. *QR* implements the right-looking tile QR factorization as described in [33]. The algorithm uses LAPACK [34] (V3.6.1) and PLASMA [35] (V2.8.0) routines.

6. *Cholesky* calculates the lower triangular matrix L of a symmetric positive definite matrix A, such that $A = LL^T$ [36]. Operations on the tiles are performed using LAPACK [34] (V3.6.1) and BLAS [37] routines.

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 has been retrieved from the BSC Application Repository [38].

For the benchmarks working on tile/block matrices we have used both single-precision (SP) and double-precision (DP) floating-point dense matrices. All the source codes and libraries/packages were compiled using the *-O3* optimization flag. For the performance results which 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 have executed each benchmark (both sequential and parallel) five times. In the parallel execution time of each execution we have included the time needed for gathering the results to the *RootNode*.

We have executed benchmarks using FREDDO with Custom Network Interface support (called FREDDO+CNI) and with MPI support (called FREDDO+MPI). Currently, FREDDO+CNI supports only Ethernet-based interconnects. The default FREDDO implementation for the AMD system is the 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 the FREDDO+CNI. For the FREDDO+MPI implementation we are using the OpenMPI library (V1.8.4 for the CyTera system and V2.0.1 for the AMD system). Notice that for both implementations, the size of the Context values is set to 64-bit.

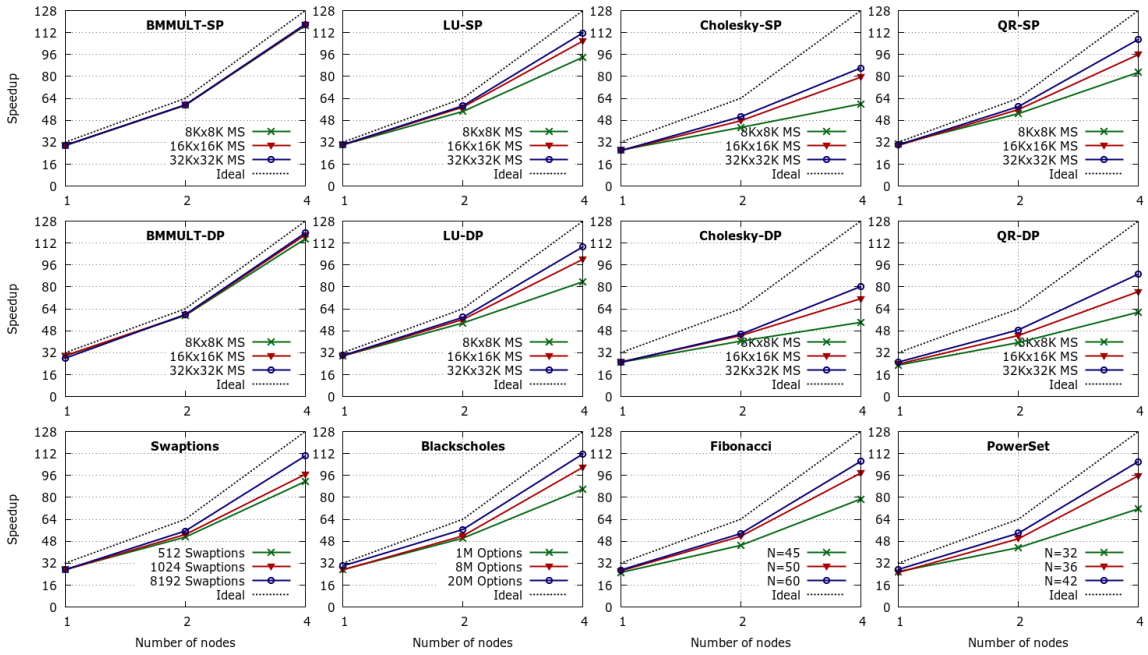


Figure 6: Strong scalability and problem size effect on the AMD system using FREDDO+CNI (MS=Matrix Size, SP=Single-Precision, DP=Double-Precision, $K = 2^{10}$, $M = 10^6$).

5.2 Performance Evaluation

We have performed a scalability study in order 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 while 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 FREDDO's runtime disables the Network Manager's *receiving thread*.

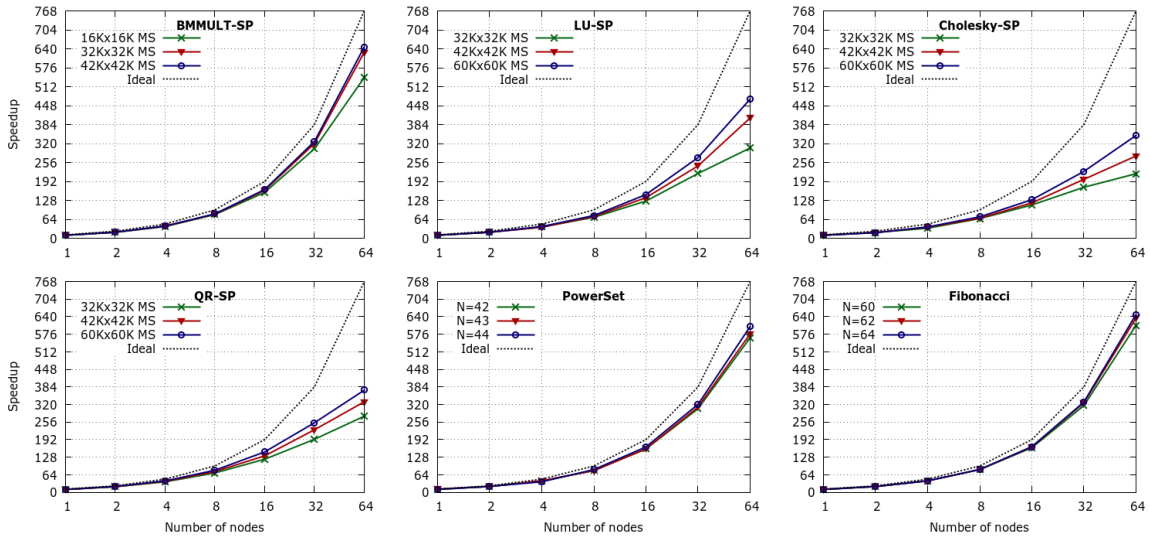


Figure 7: Strong scalability and problem size effect on the CyTera system using FREDDO+MPI (MS=Matrix Size, SP=Single-Precision, $K = 2^{10}$).

Figures 6 and 7 depict the results for the AMD and CyTera systems, respectively. On the former system we have executed all the benchmarks, including both single-precision and double-precision versions of the algorithms working on tile/block matrices. On the latter system we have executed the two recursive algorithms and the single-precision versions of the tiled algorithms. *Blackscholes* and *Swaptions* as well as the double-precision versions of the tiled algorithms are excluded from our performance evaluation on the CyTera system in order to save computational resources (CPU hours[2]). From the performance results we observe that generally as the input size increases, the system scales better (especially for the benchmarks with the complex Dependency Graphs). This is expected, as larger problem sizes allow for 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 double-precision versions of the algorithms achieve slightly lower speedups compared to the single-precision ones, since in the former case the data exchanged in the network is doubled.

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 2: Average sequential execution time (in seconds) of the sequential version of the benchmarks.

The *BMMULT*, *Blackscholes* and *Swaptions* benchmarks achieve very good speedups due to the low data sharing and low data exchange between the nodes. Particularly, they achieve up to 93%

[2] CyTera allows up to 50,000 CPU hours per user.

of the ideal speedup, on the AMD system, for the largest problem size, when all the nodes are used. For the same configuration, *BMMULT* achieves 84% of the ideal speedup, on the CyTera system. *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 inter-node communication and the complexity of the algorithms. When utilizing all the nodes on the CyTera system, 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 the AMD system 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 the *Fibonacci* and 79% (604 out of 768) for the *PowerSet* of the ideal speedup, also for the largest problem size. For minimizing the overheads of the parallel recursive implementations we have used *thresholds* in order 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 we choose the one that provides the best performance.

To conclude, distributed FREDDO scales well and effectively leverages the decoupling of synchronization and execution. Table 3 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 the available nodes are used, FREDDO achieves an average of 82% of the ideal speedup on the AMD system and 67% on the CyTera system.

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

Table 3: Minimum, Maximum and Average speedup results along with the utilization percentage of the available cores in each case.

5.3 FREDDO: CNI vs MPI

In this section we study the performance penalties of using MPI instead of CNI, for the benchmarks that have medium and heavy inter-node communication. For our experiments we have used the AMD system with all the 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 6% faster than FREDDO+MPI on average, for the smallest, medium and largest problem sizes, respectively. This indicates that MPI has more overheads which affect the performance of the Network Manager's receiving thread as well as the sending operations of the Kernels. MPI has more overheads than CNI since it's a much larger library which contains

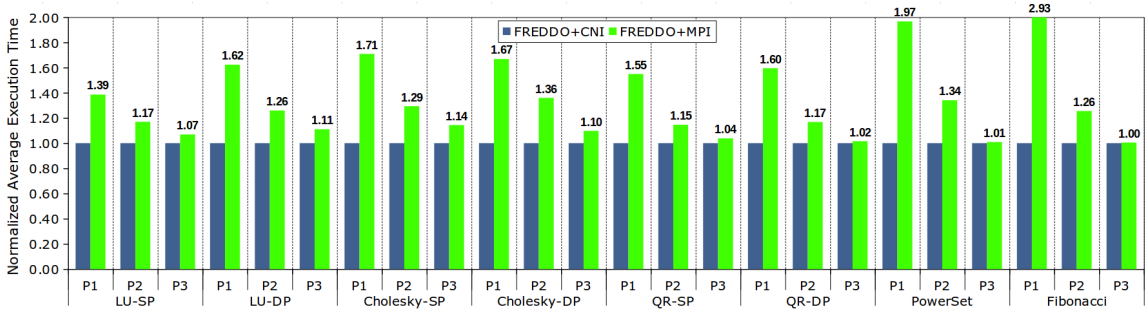


Figure 8: FREDDO+CNI vs. FREDDO+MPI on AMD for the 4-node configuration.

more functionalities than CNI. However, the MPI's overheads are hidden as the benchmark's execution time increases. Thus, the 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 the FREDDO applications, especially when targeting large-scale HPC systems with different architectures. Furthermore, it allows the programmers to combine MPI code with FREDDO code.

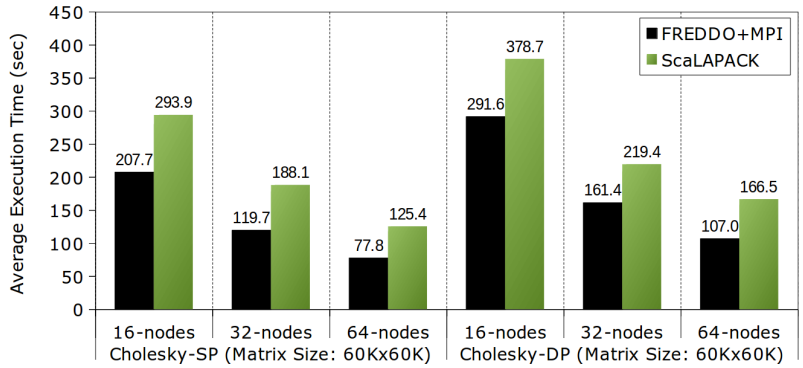


Figure 9: FREDDO vs. ScaLAPACK on CyTera for the Cholesky benchmark.

5.4 Performance comparisons with other systems

FREDDO is compared with two state-of-the-art frameworks for the *Cholesky* benchmark: ScaLAPACK [19] (V2.0.2) and SWARM [20] (V0.14). ScaLAPACK (Scalable LAPACK) is an MPI-based library which includes a subset of LAPACK [34] routines redesigned for distributed-memory parallel systems. In this work ScaLAPACK has been installed using the OpenMPI library. We have also compared our implementation with DDM-VM [18], the first distributed DDM implementation, for the *Cholesky* and *LU* benchmarks. For all frameworks we choose the configurations that achieve the optimal performance (e.g., the tile size for the tiled algorithms and the grid configuration for the ScaLAPACK implementation).

Figure 9 compares FREDDO with ScaLAPACK on the CyTera system for the Cholesky benchmark using $60K \times 60K$ matrices. The results are shown as average execution times for three different node configurations (16, 32 and 64). In Figure 10 we compare FREDDO with ScaLAPACK and SWARM for the *Cholesky* benchmark and with DDM-VM for the *Cholesky* and *LU* benchmarks, on the AMD system. For both benchmarks we have used the largest problem size ($32K \times 32K$ matrices). In order to have fair comparisons on the AMD system, FREDDO is compared with DDM-VM and SWARM using the FREDDO+CNI implementation and with ScaLAPACK using the FREDDO+MPI implementation. The reason is that ScaLAPACK utilizes the MPI library for

the inter-node communication which incurs more overheads (as shown in Figure 8) compared to a custom network network interface with less functionalities.

The comparison results show that FREDDO outperforms the other frameworks in all cases. Table 4 indicates how many times FREDDO is faster than the other frameworks, in each system, for each node configuration. For instance, on the CyTera system for the 64-node configuration, FREDDO is 1.61× faster than ScaLAPACK for the Cholesky-SP and 1.56× faster for the Cholesky-DP. Notice that for the ScaLAPACK's Cholesky implementations the *pspotrf* and *pdpotrf* routines were used. On the AMD system, FREDDO is up to 1.89× faster than ScaLAPACK, 1.26× faster than SWARM and up to 1.42× faster than DDM-VM.

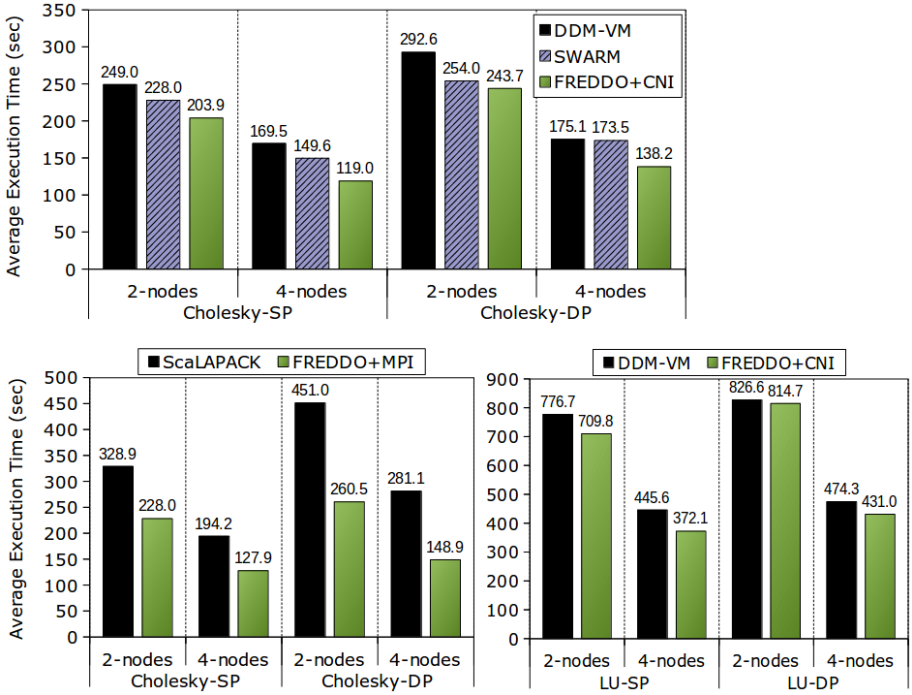


Figure 10: Comparing FREDDO with ScaLAPACK, DDM-VM and SWARM on the AMD system for the largest problem size ($32K \times 32K$ matrix size).

The ScaLAPACK ended up with lower performance due to its inability to fully exploit thread-level parallelism on shared-memory multi-core systems [39, 36]. The main reason for this is that the ScaLAPACK's algorithms, like *Cholesky*, rely on the fork-join paradigm and the MPI's programming model for the distributed execution [40, 41]. On the other hand, FREDDO allows data-driven concurrency using non-blocking threads, thus it achieves better performance on distributed shared-memory multi-core systems. The main reason that FREDDO outperforms SWARM is that the latter utilizes work-stealing for load-balancing across the nodes where FREDDO utilizes a simpler two-layer scheduling policy (see Section 3.3). Work-stealing increases the runtime overheads which can be avoided for benchmarks with regular Dependency Graphs, like *Cholesky*.

FREDDO vs. ScaLAPACK, SWARM and DDM-VM on AMD (Matrix Size: 32Kx32K)									FREDDO vs. ScaLAPACK on CyTera (Matrix Size: 60Kx60K)					
	Cholesky-SP		Cholesky-DP		LU-SP		LU-DP		Cholesky-SP			Cholesky-DP		
	2-nodes	4-nodes	2-nodes	4-nodes	2-nodes	4-nodes	2-nodes	4-nodes	16-nodes	32-nodes	64-nodes	16-nodes	32-nodes	64-nodes
ScaLAPACK	1.44X	1.52X	1.73X	1.89X	-	-	-	-	1.42X	1.57X	1.61X	1.30X	1.36X	1.56X
SWARM	1.12X	1.26X	1.04X	1.26X	-	-	-	-	-	-	-	-	-	-
DDM-VM	1.22X	1.42X	1.20X	1.27X	1.09X	1.20X	1.01X	1.10X	-	-	-	-	-	-

Table 4: Performance improvement ($Xavg/Favg$) of FREDDO compared to the other frameworks ($Favg$ is the average execution time of FREDDO and $Xavg$ is the average execution time of a compared framework).

Although DDM-VM and FREDDO are based on the same execution model, FREDDO achieves better performance for three main reasons:

1. DDM-VM follows a Context-based distribution scheme (similarly to this work) where each DThread instance is mapped and executed on a specific core of the distributed system. In FREDDO, the DThread instances are mapped to specific nodes and the TSU's Scheduler distributes them to the Kernels with the least work-load. This approach can better improve the load-balancing in each node.

2. FREDDO provides optimized TSU and Network Manager modules. For example, the FREDDO's Network Manager uses *atomic variables* to count the number of outgoing and incoming messages in each node, which is required for the distributed execution termination algorithm. In DDM-VM, the same functionality is implemented using lock/unlock operations which incur more overheads.

3. In DDM-VM, the TSU and the *receiving thread* of the Network Interface Unit (similar to the *receiving thread* of the FREDDO's Network Manager) 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.

5.5 Network Traffic Analysis

In order to study the efficiency of the proposed mechanisms for reducing the network traffic, during a distributed data-driven execution, we have performed a traffic analysis for both FREDDO and DDM-VM. We are comparing our system with 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* (single precision versions). For our experiments a root access was required for capturing the network traffic. Thus, the AMD system has been used since it's the only system where we have root access. Figure 11 depicts the total TCP packets (in Millions) and the total data (in GB) that are exchanged between the nodes of the AMD system, for the 4-node configuration and the largest problem size ($32K \times 32K$). Additionally, the benchmarks have been executed with three different tile sizes (32×32, 64×64 and 128×128).

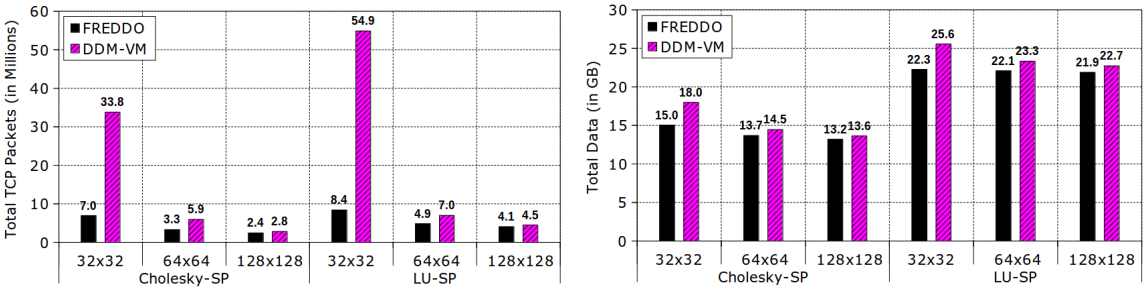


Figure 11: Network traffic analysis: FREDDO against DDM-VM on the AMD system, for the 4-node configuration and the largest problem size ($32K \times 32K$).

For the traffic analysis experiments we have used the *TShark* tool [42] (V2.2.3) and we configured it to capture the traffic that is exchanged between the TCP ports that were reserved for the inter-node communication. It is important to note that in FREDDO the size of the Context values is set to 64-bit while DDM-VM supports only 32-bit Context values. Larger Context values allow to execute benchmarks with large problem sizes and fine-grained threads [27] (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. Additionally, 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 since 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, 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, we expect to have a high positive impact on benchmarks that run on HPC systems with a large number of cores/nodes. In such systems usually fine-grained threads (e.g., with small tile sizes) are used in order to better utilize the large number of computation cores. In order to justify this, in Figure 12, we provide the normalized average execution time of the *LU* and *Cholesky* benchmarks that are executed on both systems (using FREDDO) 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-grained threads) can decrease the performance, especially in the CyTera system with the 64-node configuration.

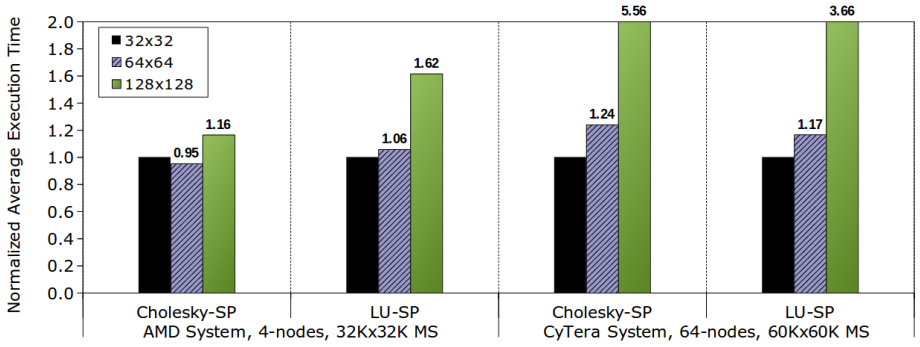


Figure 12: Tile size effect on the AMD and CyTera systems using FREDDO.

6 Related Work

The Message Passing Interface (MPI) [1] has been 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 achieving high-performance it requires a lot of effort from the programmer (partitioning/distributing data, exchanging data during execution, synchronizing nodes, etc.). On the other hand, FREDDO allows distributed data-driven execution and automatic management of a software Distributed Shared Memory which simplifies programmability. However, FREDDO uses MPI as a communication medium in order to provide portability to the DDM programs.

Thread Building Blocks (TBB) is an API developed by Intel that relies on C++ templates to facilitate parallel programming [43]. It provides a set of data-structures and algorithmic skeletons that supports the execution of tasks. TBB also provides a set of concurrent containers (hashmaps, queues, etc.) and synchronization constructs (mutex constructs, atomic operations). The TBB

runtime implements a tasks-stealing scheduling policy and adopts a fork-join approach for the creation and management of tasks, similarly to Cilk approach [44]. On the contrary, FREDDO relies on data-dependencies for the scheduling of threads. Also, unlike FREDDO, TBB's thread management does not naturally extend across the network.

Gupta and Sohi [45] have also produced a software system that allows data-flow/data-driven execution of sequential imperative programs on multi-core systems. Particularly, they have implemented a C++ runtime library that exploits Functional-Level Parallelism (FPL) by executing functions on the cores in a data-flow fashion. The execution model of this work determines data dependences between computations/functions dynamically and executes them concurrently in a data-flow fashion. In contrast, FREDDO applies its techniques at non-blocking threads. Furthermore, like TBB, the Gupta and Sohi's runtime system does not support distributed data-driven execution.

The Data-Driven Multithreading Virtual Machine (DDM-VM) [22, 18] is a parallel software platform that supports data-driven execution on homogeneous and heterogeneous multi-core systems. A DDM-VM program consists of ANSI-C with a set of C macros that expand into calls to the DDM-VM runtime. In FREDDO, a program consists of C++ objects, thus, the programming interface provides the benefits of object-oriented programming (Data Encapsulation, Data Abstraction, etc.). Furthermore, FREDDO achieves better performance than DDM-VM and it provides more functionalities such as: recursion support, mechanisms for reducing the network traffic and MPI-based inter-node communication for better portability.

SWift Adaptive Runtime Machine (SWARM) [20] is a software runtime that uses an execution model based on codelets [5]. SWARM divides a program into tasks with runtime dependencies and constraints that can be executed when all runtime dependencies and constraints are met. The SWARM's runtime schedules the tasks for execution based on resource availability and it 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, i.e. the programmer/compiler is responsible for constructing the Dependency Graph. FREDDO, in contrast to SWARM, handles the flow of data between producers and consumers running on different nodes, automatically.

OmpSs [46, 47] is a variant of OpenMP [2] 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. The programmer annotates 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 it 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, and consequently, only a fraction of the concurrency opportunities in the applications is visible at any time.

The PaRSEC framework [48, 49] is a task-based data-flow-driven runtime designed to achieve high performance computing 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 (e.g., distribution of data onto nodes, priorities, etc.) in order to allow efficient data-driven scheduling. Both PaRSEC and FREDDO performed a static work distribution between nodes. Additionally, PaRSEC performs dynamic work stealing within each node where FREDDO distributes the DThread instances to the Kernels with the least amount of work. Finally, PaRSEC provides a C-based programming interface where the *PaRSEC Compiler* is used to produce the JDF representation. FREDDO provides an object-oriented C++ programming interface where the dependency graph consists of DThread objects and it is executed using the DDM's Update operations.

The Decoupled Threaded Architecture - Clustered (DTA-C) [50] is an SDF architecture [51] with the addition of the concept of clusterizing resources. The architecture is composed of a set of clusters where each cluster consists of one or more Processing Elements (PEs) and a Distributed

Scheduler Element (DSE). The Distributed Scheduler (DS) consists of all the system's DSEs and it's responsible for assigning threads at runtime. In SDF architectures, like DTA-C, the computation is carried out by a custom designed processor while in DDM architectures, like FREDDO, the computation is carried out by an off-the-shelf processor.

7 Conclusions and Future Work

The data-flow/data-driven model is an alternative model of execution that can be used to tolerate long memory and synchronization latencies on HPC systems. In this work we have presented a portable and efficient implementation of the Data-Driven Multithreading (DDM) model that enables efficient data-driven scheduling on distributed multi-core architectures. The proposed framework has been 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 has shown that the distributed FREDDO implementation scales well and it achieves comparable or better performance when is compared with other systems, such as ScaLAPACK, SWARM and DDM-VM. Particularly, on the 768-core system, FREDDO is up to 1.61× faster than ScaLAPACK. On the 128-core system, FREDDO is up to 1.89× faster than ScaLAPACK, 1.26× faster than SWARM and up to 1.42× faster than DDM-VM. Furthermore, we have proposed simple and efficient techniques for reducing network traffic during the execution of DDM applications. Our experiments on the 128-core system have shown 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 is compared with the DDM-VM system.

Future work will be focused on applying data-driven scheduling on heterogeneous HPC systems. Particularly, we are interested on many-core accelerators with software-controlled scratch-pad memories. An example of this architecture is the Sunway SW26010 processor which is the basic building block of the Sunway TaihuLight [52] (ranked 1st in TOP500). Deterministic data prefetching into scratch-pad memories using data-driven techniques can improve the locality of sequential processing. We believe that this approach can further improve the performance of HPC systems. Additionally, we would like to evaluate the distributed FREDDO implementation on larger HPC systems which currently are not available to us.

Acknowledgment

This work was partially funded by the IKYK foundation through a scholarship for George Matheou.

References

[1] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: portable parallel programming with the message-passing interface*. MIT press, 1999, vol. 1.

[2] OpenMP Architecture Review Board, "OpenMP application program interface version 4.5," Nov. 2015. [Online]. Available: http://www.openmp.org/mp-documents/openmp-4.5.pdf

[3] Arvind and R. A. Iannucci, "Two fundamental issues in multiprocessing," in *4th International DFVLR Seminar on Foundations of Engineering Sciences on Parallel Computing in Science and Engineering*. New York, NY, USA: Springer-Verlag New York, Inc., 1988, pp. 61–88.

[4] P. Kogge, "Next-generation supercomputers," *IEEE Spectrum, February*, 2011.

[5] S. Zuckerman, J. Suetterlein, R. Knauerhase, and G. R. Gao, "Position paper: Using a codelet program execution model for exascale machines," in *EXADAPT Workshop*, 2011.

[6] G. Matheou, P. Evripidou, and C. Kyriacou, "Paradigm shift for exascale computing," in *Proceedings of the 3rd International Conference on Exascale Applications and Software*. University of Edinburgh, 2015, pp. 109–114.

[7] A. Yarkhan and J. Dongarra, "Lightweight superscalar task execution in distributed memory," 2014.

[8] J. B. Dennis, "First version of a data flow procedure language," in *Programming Symposium*. Springer, 1974, pp. 362–376.

[9] J. B. Dennis and D. P. Misunas, "A preliminary architecture for a basic data-flow processor," in *ACM SIGARCH Computer Architecture News*, vol. 3, no. 4. ACM, 1975, pp. 126–132.

[10] J. R. Gurd, C. C. Kirkham, and I. Watson, "The manchester prototype dataflow computer," *Commun. ACM*, vol. 28, no. 1, pp. 34–52, Jan. 1985. [Online]. Available: http://doi.acm.org/10.1145/2465.2468

[11] B. Lee and A. R. Hurson, "Dataflow architectures and multithreading," *Computer*, vol. 27, no. 8, pp. 27–39, 1994.

[12] W. M. Johnston, J. Hanna, and R. J. Millar, "Advances in dataflow programming languages," *ACM Computing Surveys (CSUR)*, vol. 36, no. 1, pp. 1–34, 2004.

[13] C. Kyriacou, P. Evripidou, and P. Trancoso, "Data-driven multithreading using conventional microprocessors," *IEEE Trans. on Parallel and Distributed Systems*, vol. 17, no. 10, pp. 1176–1188, Oct. 2006.

[14] G. Matheou and P. Evripidou, "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)*, 2016, pp. 265–273.

[15] J. Protic, M. Tomasevic, and V. Milutinović, *Distributed shared memory: Concepts and systems*. John Wiley & Sons, 1998, vol. 21.

[16] D. A. Koufaty, X. Chen, D. K. Poulsen, and J. Torrellas, "Data forwarding in scalable shared-memory multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 12, pp. 1250–1264, 1996.

[17] J. Matocha and T. Camp, "A taxonomy of distributed termination detection algorithms," *Journal of Systems and Software*, vol. 43, no. 3, pp. 207–221, 1998.

[18] G. Michael, S. Arandi, and P. Evripidou, "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)*, 2013.

[19] 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, *ScaLAPACK User's Guide*, J. J. Dongarra, Ed. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1997.

[20] C. Lauderdale, M. Glines, J. Zhao, A. Spiotta, and R. Khan, "Swarm: A unified framework for parallel-for, task dataflow, and distributed graph traversal," *ET International Inc., Newark, USA*, 2013.

[21] P. Evripidou, "Thread synchronization unit (tsu): A building block for high performance computers," in *International Symposium on High Performance Computing*. Springer, 1997, pp. 107–118.

[22] S. Arandi and P. Evripidou, "Programming multi-core architectures using data-flow techniques," in *Embedded Computer Systems (SAMOS), 2010 International Conference on*. IEEE, 2010, pp. 152–161.

[23] ——, "DDM-VMc: the data-driven multithreading virtual machine for the cell processor," in *Proc. of the 6th Int. Conf. on High Performance and Embedded Architectures and Compilers*. ACM, 2011, pp. 25–34.

[24] G. Matheou and P. Evripidou, "Verilog-based simulation of hardware support for data-flow concurrency on multicore systems," in *SAMOS XIII, 2013*. IEEE, 2013, pp. 280–287.

[25] ——, "Architectural support for data-driven execution," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 4, pp. 52:1–52:25, Jan. 2015. [Online]. Available: http://doi.acm.org/10.1145/2686874

[26] A. Mondelli, N. Ho, A. Scionti, M. Solinas, A. Portero, and R. Giorgi, "Dataflow support in x86_64 multicore architectures through small hardware extensions," in *Digital System Design (DSD), 2015 Euromicro Conference on*. IEEE, 2015, pp. 526–529.

[27] G. Matheou and P. Evripidou, "FREDDO: an efficient framework for runtime execution of data-driven objects," Department of Computer Science, University of Cyprus, Nicosia, Cyprus, Tech. Rep. TR-16-1, January 2016. [Online]. Available: https://www.cs.ucy.ac.cy/docs/techreports/TR-16-1.pdf

[28] S. Arandi, "The data-driven multithreading virtual machine," Ph.D. dissertation, University of Cyprus, 2011.

[29] S. Amarasinghe, M. Hall, R. Lethin, K. Pingali, D. Quinlan, V. Sarkar, J. Shalf, R. Lucas, K. Yelick, P. Balanji, *et al.*, "Exascale programming challenges," in *Proceedings of the Workshop on Exascale Programming Challenges, Marina del Rey, CA, USA. US Department of Energy, Office of Science, Office of Advanced Scientific Computing Research (ASCR)*, 2011.

[30] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta, "Hierarchical task-based programming with starss," *International Journal of High Performance Computing Applications*, vol. 23, no. 3, pp. 284–299, 2009.

[31] T. C. Institute, "Cy-Tera," http://web.cytera.cyi.ac.cy, 2017, [Online; accessed 25-Mar-2017].

[32] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM, 2008, pp. 72–81.

[33] F. Song and J. Dongarra, "Scaling up matrix computations on shared-memory manycore systems with 1000 cpu cores," in *Proceedings of the 28th ACM international conference on Supercomputing*. ACM, 2014, pp. 333–342.

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

[35] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov, "Numerical linear algebra on emerging architectures: The plasma and magma projects," in *Journal of Physics: Conference Series*, vol. 180, no. 1. IOP Publishing, 2009, p. 012037.

[36] J. Kurzak, H. Ltaief, J. Dongarra, and R. M. Badia, "Scheduling linear algebra operations on multicore processors," 2009, lAPACK Working Note 213.

[37] L. S. Blackford, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, *et al.*, "An updated set of basic linear algebra subprograms (blas)," *ACM Transactions on Mathematical Software*, vol. 28, no. 2, pp. 135–151, 2002.

[38] BSC, "BSC Application Repository," https://pm.bsc.es/projects/bar/wiki/Applications, 2017, [Online; accessed 25-Mar-2017].

[39] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, J. Kurzak, P. Luszczek, S. Tomov, and J. J. Dongarra, "Scalable dense linear algebra on heterogeneous hardware," *Advances in Parallel Computing*, 2013.

[40] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz, "Communication-optimal parallel and sequential cholesky decomposition," *SIAM Journal on Scientific Computing*, vol. 32, no. 6, pp. 3495–3523, 2010.

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

[42] U. Lamping and E. Warnicke, "Wireshark user's guide," *Interface*, vol. 4, no. 6, 2004.

[43] J. Reinders, *Intel threading building blocks: outfitting C++ for multi-core processor parallelism.* " O'Reilly Media, Inc.", 2007.

[44] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *Journal of parallel and distributed computing*, vol. 37, no. 1, pp. 55–69, 1996.

[45] G. Gupta and G. S. Sohi, "Dataflow execution of sequential imperative programs on multicore architectures," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2011, pp. 59–70.

[46] J. Bueno, L. Martinell, A. Duran, M. Farreras, X. Martorell, R. M. Badia, E. Ayguade, and J. Labarta, "Productive cluster programming with ompss," in *European Conference on Parallel Processing*. Springer, 2011, pp. 555–566.

[47] J. Bueno, X. Martorell, R. M. Badia, E. Ayguadé, and J. Labarta, "Implementing ompss support for regions of data in architectures with multiple address spaces," in *Proceedings of the 27th international ACM conference on International conference on supercomputing*. ACM, 2013, pp. 359–368.

[48] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra, "Dague: A generic distributed dag engine for high performance computing," *Parallel Computing*, vol. 38, no. 1, pp. 37–51, 2012.

[49] A. Danalis, H. Jagode, G. Bosilca, and J. Dongarra, "Parsec in practice: Optimizing a legacy chemistry application through distributed task-based execution," in *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*. IEEE, 2015, pp. 304–313.

[50] R. Giorgi, Z. Popovic, and N. Puzovic, "Dta-c: A decoupled multi-threaded architecture for cmp systems," in *Computer Architecture and High Performance Computing, 2007. SBAC-PAD 2007. 19th International Symposium on*. IEEE, 2007, pp. 263–270.

[51] K. M. Kavi, R. Giorgi, and J. Arul, "Scheduled dataflow: Execution paradigm, architecture, and performance evaluation," *IEEE Transactions on Computers*, vol. 50, no. 8, pp. 834–846, 2001.

[52] H. Fu, J. Liao, J. Yang, L. Wang, Z. Song, X. Huang, C. Yang, W. Xue, F. Liu, F. Qiao, *et al.*, "The sunway taihulight supercomputer: system and applications," *Science China Information Sciences*, vol. 59, no. 7, p. 072001, 2016.