Exploring HPC Parallelism with Data-Driven Multithreating

Constantinos Christofi, George Michael, Pedro Trancoso and Paraskevas Evripidou

Department of Computer Science, University of Cyprus
Nicosia,Cyprus
Email: {christofi.c,xeirwn,pedro,skevos}@cs.ucy.ac.cy

***Abstract*—The switch to Multi-core systems has ended the reliance on the single processor for increase in performance and moved into Parallelism. However, the exponential growth in performance of the single processor in the 80's and 90's had overshadowed the drive for efficient Parallelism and relegate it into a niche research area, mostly for High Performance Computing (HPC). Parallelism now is in the forefront and holds the burden for utilising the extra resources of Moore's law to maintain the exponential growth of the computing systems.In the drive to utilise parallel models of computation, Data-Flow models have recently been "re-visited" for exploiting parallelism in the multi and many core systems. Data-Driven Multithreading (DDM) is one such model which is based on Dynamic Data-Flow principles, that can expose the maximum parallelism of an application. DDM schedules Threads based on Data availability driven by a producer consumer graph. DDM enforces single assignments semantics on the data passed from producer to consumer.**

In this paper we present a preliminary evaluation of whether DDM can be viable candidate for HPC. We study the scalability of a small subset of the LINPACK benchmark using the Data-Driven Multithreading for a system with a 48 cores. We implement three test case operations: Matrix Multiplication, LU and Cholesky decompositions and use them to test their scalability and performance. We use optimized linear algebra kernel operation for the basic operations performed in the threads. We compare our DDM implementations against PLASMA, a state-of-the-art linear algebra library for HPC computing, and show that applications using the DDM model can scale efficiently and observe a performance improvement of up to $2\times$.

I. INTRODUCTION

Efficient and scalable Parallelism is the key to achieving high performance in Multi/Many-core systems. Sequential (control-flow) computing, also know and as von Neumann computing, has been the de-facto model of programming and computation since the advent of digital computers. The Power and Memory walls have forced the switch to multiple cores per chip thus elevating concurrency as the key challenge in achieving high performance. New parallel programming and computation models are needed in order to fully exploit the ever increasing number of cores per chip.

In this paper we investigate wether Data-Driven Multithreading (DDM) can achieve high performance in in HPC applications. DDM combines Data-Flow concurrency with the efficiency of sequential execution. DDM has been implemented and evaluated in both Heterogeneous (CELL) and Homogeneous Multi-core systems[6].

DDM is a multithreaded model that applies Dynamic Data-flow principles for communication among threads and exploits highly efficient control-flow execution within a thread. The core of DDM is the Thread Scheduling Unit that provides the Data-Driven scheduling of the Threads. DDM does not need traditional memory coherence because it enforces the single assignment semantics for data exchange among threads. Furthermore, it employs prefetching of input data before a thread is scheduled for execution by the TSU. DDM prefetching can be very near to optimal and deterministic because for each core the TSU knows at any time which threads can be executed and thus can initiate the necessary prefetching. DDM based processors can achieve high performance with simpler designs, as they do not need complex and expensive modules like out-of-order execution.

We believe that the DDM model can inspire the evolution of micro-architecture with the addition of a hardware TSU on-chip and the use of a data-driven hierarchy of scratch-pad memories that can replace the traditional multilevel cache hierarchy. Such memory hierarchies will be deterministic and smaller in size than current cache hierarchies. We have implemented such a system in software for the CELL processor. We are currently developing an FPGA based Distributed Multi-core system in which we are introducing these micro-architecture changes.

In this paper we are presenting a preliminary evaluation as to whether Data-Flow concurrency as implemented in DDM can be a viable candidate for future High Performance Computing (HPC). The initial results are very promising.

In DDM the programmer is responsible to find the portions of code that can execute in parallel, the Data-Driven Threads (DThreads), and the dependencies among themselves. These dependencies are expressed with the *input* and *output* values for each thread. The programmer just needs to express all of the above by augmenting the original code using TFlux DDM pragma directives, like the ones listed in Table I.

To investigate the benefits from the use of the DDM model we have implemented three linear algebra applications (Matrix Multiplication, LU decomposition and Cholesky decomposition) using the DDM model and tested their scalability for large number of cores. To achieve the best performance we use highly tuned linear algebra kernel operations in the threads. We compare our results against the advanced PLASMA[**?**][12]

linear algebra library. Our experiments show that the DDM implementations of the applications can benefit from increased performance around $2\times$ for all applications for 48 cores compared to the state-of-the-art PLASMA linear algebra library.

The rest of the paper is organized as follows. In Section II we provide the background and discuss the related work. The Data-Driven Multithreading model is presented in Section III. In Section IV we show the details of the implementation of our test case scenarios. In Section V we present the setup and the evaluation of our test cases. Finally in Section VI we outline the conclusions and future work.

II. Background and Related work

In order to program large-scale parallel systems, programmers may use a parallel library that is tuned to the system as to exploit the best performance. There are plenty of libraries available for the programmer to choose but the majority of them were originally designed for single processor machines. Even thought almost all the libraries are now supporting multi-core systems their parallel implementations exhibit limited scalability due to their original sequential design. In the subsections following we present some of the available libraries for linear algebra computations. In addition, we present some of the latest work in exploring performance applying the Data-Flow model of execution.

A. BLAS - LAPACK

The Basic Linear Algebra Subprograms *(BLAS)* are routines that provide the standard building blocks for performing basic vector and matrix operations[17][18]. They are considered a standard and there are implementations from various developers for a number of architectures. The BLAS consists of three levels of functions each one performing different operations. The Level 1 BLAS perform scalar, vector and vector-vector operations, the Level 2 BLAS perform matrix-vector operations and finally Level 3 BLAS perform matrix-matrix operations. BLAS is considered to be efficient, portable and widely available due to the numerous free implementations available. Because of these characteristics it is commonly used for the development of high quality linear algebra software, like Linear Algebra PACKage (LAPACK) for example.

The *LAPACK* is a software library for numerical linear algebra[1][4]. It provides routines for solving systems of simultaneous linear equations, least-square solutions of linear system equations, eigenvalue problems and singular value problems. It also includes matrix factorizations like LU, Cholesky and QR. LAPACK implementations are available by various developers or vendors optimized for specific systems included in free or commercial software packages. In Figure 1 we show how LAPACK's function for performing the LU factorization is "constructed" using calls to other LAPACK or BLAS functions.

Among the most popular packages providing BLAS and LAPACK commercial products like Intel's MKL (for Math Kernel Library) offer good performance for various architectures. MKL includes highly optimized routines for BLAS and

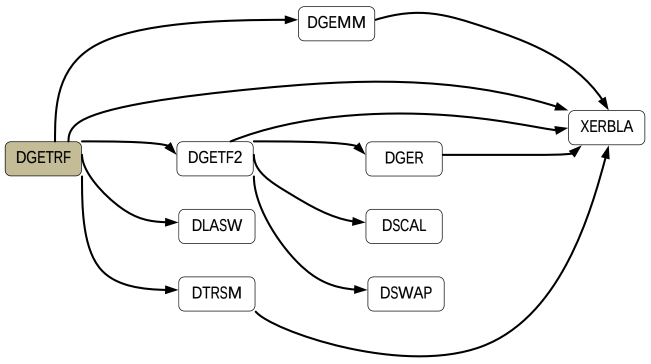


Fig. 1. Call graph for LU factorization from LAPACK[4]

LAPACK and also functions for sparce solvers, fast Fourier transforms and vector math.

B. PLASMA

Parallel Linear Algebra Software for Multi-core Architectures[3][2] (PLASMA) although was developed to solve the same problems as LAPACK is designed so it can deliver high performance on multi-socket systems by using a different strategy for extracting parallelization. PLASMA is also built on top of optimized BLAS libraries but instead of relying on them for the parallelization it uses a combination of newly design tiled algorithms and dynamic scheduling of parallel tasks. The tiled algorithms developed for the PLASMA project lay out the matrices in square tiles with relatively small size so that each tile occupies a continuous memory region. This enables the maximizing of data reuse as the tile is loaded into the processors cache and is not evicted back to main memory until is completely processed. PLASMA also benefits for the runtime dynamic scheduling of parallel tasks which is based on the idea of scheduling by enforcing data dependencies between tasks[19] (QUARK), in other words assigning work to cores based on the availability of data to be processed in any given time of the execution. All the parallel tasks (usually BLAS kernels) are expressed in a Directed Acyclic Graph (DAG) and they depend on each other only on the availability of the data that are going to be processed. This producer-consumer relationship is able to synchronize the tasks and extract the highest degree of parallelism between them.

C. Data-Flow Programming and Execution Models

1) SMP Superscalar: SMP superscalar (SMPSs)[21] is a programming environment focused on the ease of programming, portability and exibility. Is based on Cell superscalar (CellSs) but instead of targeting Cell processor, SMPSs tailored for multi-cores and Symmetric Multiprocessors (SMP) in general. SMPSs is based on the Data-Flow model and it achieves parallelism through the use of pragmas (given by the programmer) that identify atomic parts of the code that operate over a set of parameters. These parts of the code are encapsulated in the form of functions (called tasks). SMPSs

TABLE I
TFLUX DDM PRAGMA DIRECTIVES

`#pragma ddm startprogram`	Define the start and the end of a DDM program
`#pragma ddm endprogram`	
`#pragma ddm block` *ID*	Define the start and the end of a block of threads with identifier *ID*
`#pragma ddm endblock`	
`#pragma ddm thead` *ID* `kernel` *NUMBER*	Define the boundaries of a DDM thread with identifier *ID* and the kernel *NUMBER* to execute on
`#pragma ddm endthread`	
`#pragma ddm for thread` *ID*	Define the boundaries of a DDM loop thread with identifier *ID*
`#pragma ddm endfor`	
`#pragma ddm kernel` *NUMBER*	Declare the number of kernels to be used
`#pragma ddm var` *TYPE NAME*	Declare a shared variable with *NAME* and *TYPE*
`#pragma ddm private var` *TYPE NAME*	Declare a private variable with *NAME* and *TYPE*

compiler and runtime library use the information from these pragmas to build a parallel application that detects the task calls and their interdependencies. A task-graph is dynamically generated, scheduled and run in parallel by using a different number of threads depending on the architecture.

2) Scheduled Data-Flow: Scheduled Data-Flow (SDF) execution paradigm is an other model that is based on coarse grained Data-Flow and multithreading[22]. SDF partitions a program into non-blocking threads where all memory accesses are decoupled from the execution. For each thread the model identifies three main phases: pre-load phase (where thread reads input data), execution phase (where all threads are executed without performing any memory accesses) and post-store phase (where results are written back). Decoupled Threaded Architecture - Clustered (DTA-C) is based on the execution paradigm of SDF while adding the concept of clustering resources[23]. A cluster consists of one or more Processing Elements (PEs) and a Distributed Scheduler Element (DSE). The set of all DSEs constitutes the Distributed Scheduler (DS).

3) Function-Level Parallelism: Gupta and Sohi in [20] implement a runtime library that dynamically parallelizes the execution of suitably-written sequential programs, in a Data-Flow fashion, on multiple processing cores. While traditional Data-Flow models exploit instruction level parallelism (ILP), this work raise the granularity to functions and seeks to exploit Function-Level Parallelism (FLP) by executing functions on the cores in a Data-Flow fashion. A function's input operands (read set) and its output operands (write set) collectively called the data set. Data sets are used describe the data dependencies among functions. Functions are "shelved" until their dependencies are resolved.

III. DATA-DRIVEN MULTITHREADING

Recently there is a lot of research interest in the Data-Flow model as a means to exploit efficient parallelism on large-scale systems such as multi-core machines with large number of cores and many-core systems[5]. Although the analysis of a program to identify the data dependencies between instructions may be a difficult task, the Data-Flow execution model is designed to expose the maximum available parallelism from a code. Based on a producer-consumer relationship the execution of a thread is schedule only when all its data are produced thus avoiding the need of any additional synchronization. DDM is a Data-Flow model in which the granularity of the model is a collection of instructions called Data-Driven Threads (DThreads)[7][6][8]. DDM programs consist of a number of DTheads which are connected to each other with a producer-consumer relationship. A Synchronization Graph is constructed by combining the DThreads and their dependencies where each node of the graph represents a DThread and each arch a data dependency between them. The scheduling of the execution of each DThread is done dynamically at runtime in a data-flow manner by the Thread Synchronization Unit (TSU). The TSU is responsible to schedule a DThread for execution, only if it's producers have finished executing and produced the required data, based on the Synchronization Graph. Although the dependency analysis of a program is a relatively difficult task if done correctly can expose the maximum available parallelism and the DDM model can achieve the optimal parallelization of the DThreads. This work is based on the DDM model as described in Section II. DDM is not build for a specific system but rather acts as a virtualization platform for DDM program execution on a variety of systems. DDM applications can be programed using the TFlux DDM directives[15] and make use of the Data-Driven Multithreading Preprocessor (DDMCPP)[16]. DDMCPP requires from the programmer a C source file augmented with TFlux DDM directives and it creates as an output a C program that includes all the runtime support needed to execute the program under the DDM model on an unmodified hardware. In the following Subsections we include some information about the TFlux DDM Directives that can be used while programming the DDM model and the TSU which is the unit responsible for scheduling the execution of the DThreads

A. TFlux DDM Directives

The TFlux DDM directives as described in Table I are used by the programmer of the application to define the boundaries of DThreads and the dependencies between them. These dependencies will later be used to create the synchronization graph needed by the Thread Synchronization Unit in order to synchronize the execution. The directives also support parallel `for` loops with cross iteration dependencies.

B. Thread Scheduling Unit

TSU is the unit that is responsible to load the synchronization graph of a DDM application and schedule the execution of the DThreads, according to the dependencies, in a Data-Flow manner. Each DDM implementation uses it own TSU but we can abstract the functionality to the scheduling of the execution of a DThread only when all the data it need are produced.

IV. DDM Test Cases

To prove the concept that DDM applications are able to scale well as the number of processors increases we implement and optimized the matrix-matrix multiplication algorithm found in Level 3 BLAS libraries and the LU and Cholesky decomposition algorithms found in LAPACK. The functions were build using DDM and were optimized for better performance.

A. Matrix-Matrix Multiplication

The evaluation of the product of two matrices can be very computationally expensive. Following the "naive" matrix multiply algorithm (shown in Figure 2) we see that multiplying two $n \times n$ matrices requires $O(n^3)$ Floating Point Operations (FLOPS) while operating on $O(n^2)$ data elements.

```
for i = 1 to N
  for j = 1 to N
    for k = 1 to N
      C(i,j) = C(i,j) + A(i,k) * B(k,j)
```

Consider A, B and C to be N by N matrices

Fig. 2. "Naive" matrix multiply algorithm.

For our DDM code to be able to fulfill its goals the algorithm has to be as optimized as possible before implemented with DDM code. The difference in the operations performed and the data used by the algorithm make it obvious that as a whole, matrix multiplication exhibits very good reuse of data. However, in general, matrices are large thus not fitting in the cache hierarchy of a processor. As a solution for this, the work must therefore be broken into smaller chunks of computation, so that each one of them uses a small enough piece of the data to fit into caches. The smaller chunks of the matrices are called blocks and the matrix multiplication can then by carried out block by block as outlined in the algorithm shown in Figure 3. The blocked matrix multiplication algorithm is explained in detail in [13] and [14].

After improving the algorithm's data reuse by using the blocked variation of the algorithm we focus on two approaches for implementing our matrix multiplication function. In the first approach the matrix multiplication between two blocks is calculated by manually traversing the blocks and multiplying them. In the second approach we replace the "manual" method with the optimized `dgemm` kernel of Level 3 BLAS library. The second implementation allowed us to benefit from all the

```
for i = 1 to B
  for j = 1 to B
    {load block C(i,j) into cache}
    for k = 1 to B
      {load block A(i,k) into cache}
      {load block B(k,j) into cache}
      C(i,j) = C(i,j) + A(i,k) * B(k,j)
      {do a matrix multiply on blocks}
    {write block C(i,j) back to memory}
```

Consider A,B and C to be N by N matrices of b by b blocks where $b = N/B$

Fig. 3. Blocked matrix multiply algorithm.

optimizations that are already included into BLAS library and also benefit from future even more optimized libraries without any changes to the code.

Matrix multiplication is by nature embarrassing parallel and that allows us to calculate the product of each block in parallel with out any dependencies. As the dependency graph of Figure 4 shows each sub-block of the matrix can be executed in parallel on any available core. The only synchronization point is at the end of the computation. At that point we have to wait for all the blocks to finish processing. At the return of the function we have the result of whole matrix as each core completes it's part of the result.

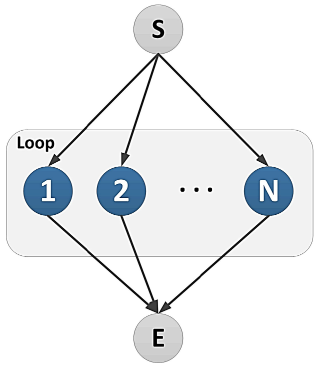


Fig. 4. Dependency graph of Matrix multiplication

B. LU Decomposition

The LU decomposition of a matrix is yet another, more complex, linear algebra operation. It computes a factorization of the form: $A = P \times L \times U$ where P is a permutation matrix, L is a lower triangular matrix with unit diagonal elements and U is an upper triangular matrix. This operation is often used in the process of solving linear systems as for example in the LINPACK benchmark. For our implementation we used a block algorithm that is implemented, with LAPACK's implementation as a baseline, to use level 3 BLAS and LAPACK functions. The LU decomposition algorithm, shown by Figure 5, consists of five nested loops that perform four basic operations in three steps. During the first step the upper left block of the matrix is factorized using the dgetrf function from LAPACK. The factorization of this block enables the second step during which all the blocks of the column bellow the factorized block can be updated with row swaps using LAPACK's dlaswp function. In the same step all the blocks that belong to the remaining row, right of the factorized block, are passed through the BLAS function dtrsm which performs a triangular solve over the blocks. In the third and final step a matrix-matrix multiply of the form $C = A \times B$ is performed where the A matrix is ts the part of the column updated with the swap and the B matrix is the row updated with the triangular solve. All columns to the right of A and rows bellow the B matrix hold the C matrix. The three steps are repeated with the factorized block moving down and to the right until all the matrix has been processed.

```
for k = 1 to B
  {factorize block A(k,k)}
  for i = k+1 to B
    {triangular solve block A(k,i)}
  for i = k+1 to B
      {apply swap on block A(k,i)}
  for i = k+1 to B
    for j = k+1 to B
      A(i,j) = A(i,j) + A(i,k) * A(k,j)
      {do a matrix multiply on blocks}
```

Consider A to be a N by N matrix of b by b blocks

Fig. 5. Blocked LU Decomposition algorithm.

The LU decomposition has a more complex dependency graph than the matrix multiply implemented in the previous subsection, thus, it requires more elaborate dependencies analysis. In Figure 6 we show the dependencies graph for a 4×4 block matrix. In every iteration of the outermost loop of the original code (Figure 5) the corresponding diagonal block is been factorized. The factorization of that block enables the execution of the "swap" and "triangular solve" operations on all blocks in the same column and row accordingly. All the column and row blocks can be updated in parallel since there are no dependencies among them. Blocks of the inner part of the matrix can be updated with a matrix multiplication operation as soon as their dependencies are solved. This means that for each of the blocks their corresponding "swap" and "triangular solve" blocks must both finish executing before they are enable for execution. The iteration of this algorithm over all the diagonal blocks produces the final decomposition of the matrix.

C. Cholesky Decomposition

Our third case study application is the Cholesky decomposition mainly used for the numerical solution of linear equations of the form $Ax = b$, where A is a symmetric and positive definite matrix. Cholesky decomposition (or Cholesky factorization) is very common in scientific applications especially in modeling physical phenomenons. The Cholesky factorization of a $n \times n$ matrix A has the form $A = LL^T$ where L in a

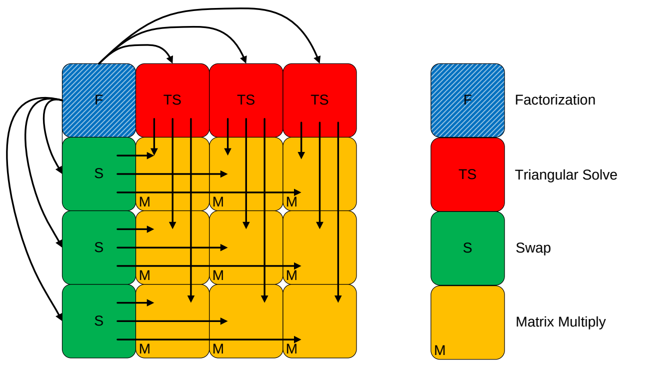


Fig. 6. LU Decomposition dependency graph for a 4×4 block matrix

lower triangular matrix with positive diagonal elements. Our implementation follows the LAPACK implementation model shown by Figure 7. The algorithm is built using four operations implemented by LAPACK and BLAS functions. It consists of a outermost loop that traverses the matrix from the top left corner block to the bottom right one processing several columns and rows per iteration.The Cholesky factorization of diagonal blocks is performed using the dpotrf function, the triangular solve, applied to the column bellow the diagonal block, is performed by the dtrsm function, symmetric rank-k update if performed by dsyrk on blocks positioned right of the diagonal block and finally dgemm applies the matrix multiplication operation on several blocks from the sub-matrix formed by the block bellow and righter of the diagonal blocks. The dependencies analysis graph of the Cholesky decomposition on a 5×5 block matrix is shown by Figure 8. After all the diagonal blocks have been traversed the lower triangular part of matrix A is overwritten by the L matrix.

```
for k = 1 to B
  {factorize block A(k,k)}
  for i = k+1 to B
    {triangular solve A(i,k)<--(k,k)}
  for i = k+1 to B
    {symmetric rank-k upadate A(i,i)<--A(i,k)}
    for j = i+1 to B
        A(j,i) = A(j,i) + A(j,k) * A(i,k)
        {do a matrix multiply on blocks}
```

Consider A to be a N by N matrix of b by b blocks

Fig. 7. Blocked Cholesky Decomposition algorithm.

V. Experiments

The performance of our implementations was measured with the experiments described in this section on a 48-core machine consisting of 4 twelve-core AMD Opteron chips running at 2.1 GHz. The machine has 32 GByte of main memory available for each chip totaling 128 GByte. More details on the system used for the experiments are given in Table II. During the performance measurements the machine

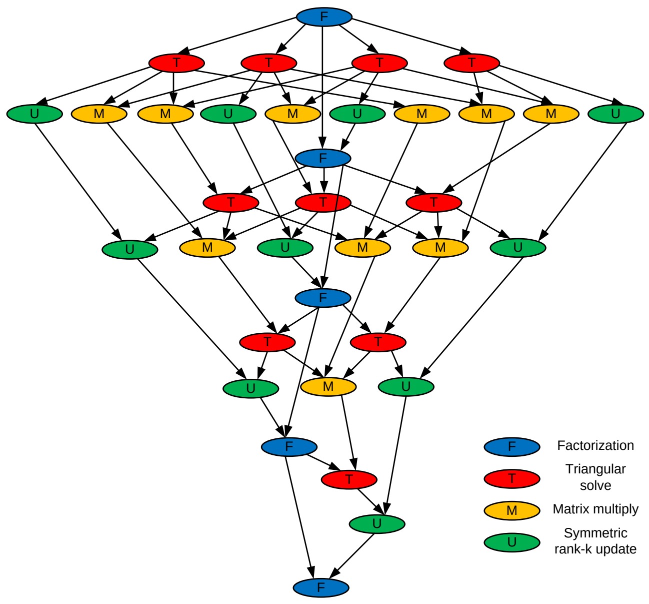


Fig. 8. Cholesky Decomposition dependency graph for a 5×5 block matrix

was running Linux OS kernel 2.6.35.7 and all the executables were compiled using the gcc 4.1.2 compiler using level 3 (O3) optimization flag. The BLAS and LAPACK kernels used during the experiments were linked from Intel's MKL 11.1. The measured performance for our implementation was compared with the performance of PLASMA library's solution for the same problems on the same machine and set up described in Table II. For the execution of the DDM code we used a DDM model implementation designed to run on shared memory multi-core systems. From all the cores available in the system one is reserved by the DDM model to serve as the TSU that schedules the execution of the DDM code. We need to note at this point that since for the DDM model a core is reserved for the TSU all the comparisons made in the charts are modified to reflect this. For example the 16-cores scenario means that for the PLASMA library 16 cores were used for execution but for the DDM implementations 15 cores were solving the problem with the remaining core acting as the TSU.

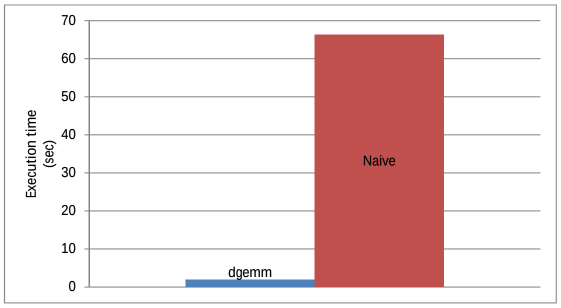


Fig. 9. Matrix Multiply execution time for the "naive" and the "dgemm" approaches

Our first experiment is performed to investigate the performance gains observed by using a highly optimized library for the BLAS and LAPACK kernels used in the DDM implementations. We executed our blocked DDM implementation for the matrix multiplication problem while running on 48-cores (47 + 1 TSU) and measure it's execution time for two scenarios. For the first scenario the product of two blocks was calculated by manually traversing the blocks and multiplying them, in the second approach we replace the "manual" method with the optimized `dgemm` kernel of Level 3 BLAS library. Figure 9 shows that using the optimized kernels available through BLAS and LAPACK our implementations can benefit from all the optimizations included in the libraries. As we can see from the chart the execution time is reduced by tens of times when using the optimized libraries.

For our second experiment we evaluated the scalability of our solutions as the number of cores increases, for two matrix sizes $4k \times 4k$ and $8k \times 8k$. The scalability of our DDM implementations is compared with the scalability shown by the PLASMA library routines. We executed the scenarios for a number of cores ranging from one up to 48. From the charts of Figure V we can see the performance for the Matrix multiplication routine. Our DDM implementation is compared against the PLASMA's `PLASMA_dgemm()` routine for the same number of cores. The matrix multiplication as described in Section IV-A has a large number of DThreads (each loop iteration is considered to be a DThread in the DDM model) and no dependencies. As we can see from the charts of Figure V the DDM solution achieves similar performance with PLASMA for the small size scenario. When we increase the problem size the DDM solution shows better scalability than the PLASMA routines. For the 48-cores scenario DDM achieves more than double the performance observed for PLASMA.

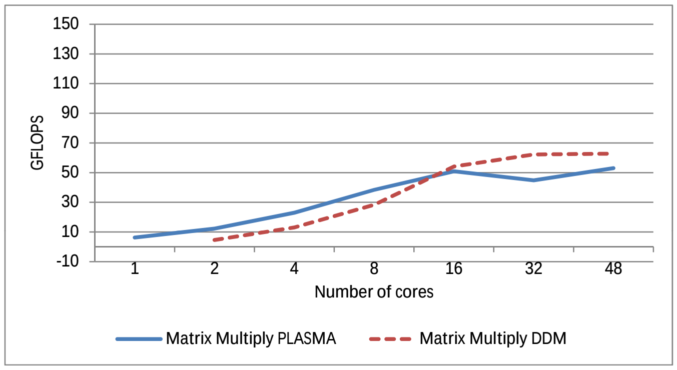
Our two next test cases LU and Cholesky decompositions have a more complicated dependencies graph and that is because of the data dependencies among their DThreads as shown by Figures 6 and 8 respectively. For the LU decomposition our solution was tested against PLASMA's `PLASMA_dgetrf()` routine running on the same scenarios. From the results of Figure 11 we see that our implementation manages to outperform PLASMA for large core counts (over 32 cores) for both the sizes small and large. Finally for the Cholesky decomposition our implementation was tested against the PLASMA's `PLASMA_dpotrf()` routine. We observe from the results shown in Figure 12 that the DDM implementation shows significantly better performance and scalability that the PLASMA solution for almost all the tested scenarios.

This preliminary performance analysis show that the DDM concurrency mechanism combined with the optimized linear algebra kernels can achieve high performance, especially when the core count increases.

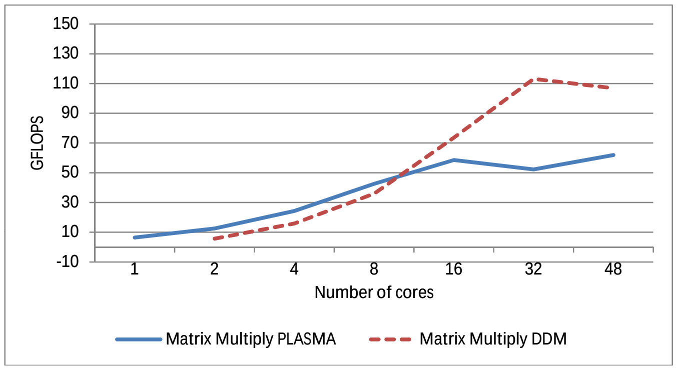
PLASMA achieves slightly better results than DDM in some cases when the number of cores is low. We believe this is due to the ability of PLASMA to adjust the block size according

TABLE II
DETAILS OF THE SYSTEMS USED FOR THE FOLLOWING PERFORMANCE RESULTS

Number of Cores	Architecture	Frequency (GHz)	L2 cache (KBytes)	L3 cache (MBytes)	RAM per core	Compiler	BLAS and LAPACK
48	AMD Opteron	2.1	512	12	2.6	gcc 4.1.2	MKL 11.1



(a)



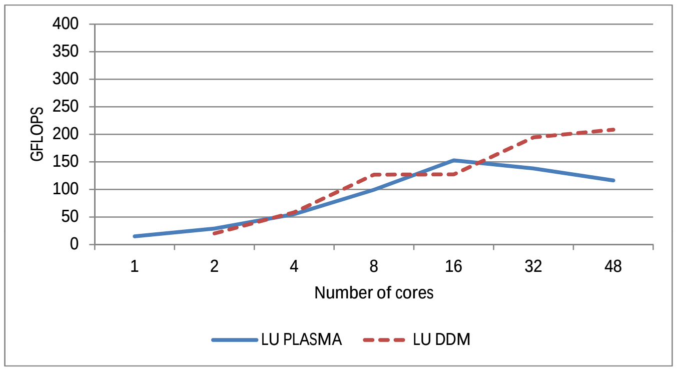
(b)

Fig. 10. Scalability of Matrix multiplication as the number of cores increases for: (a) $4k \times 4k$ matrix and (b) $8k \times 8k$ matrix

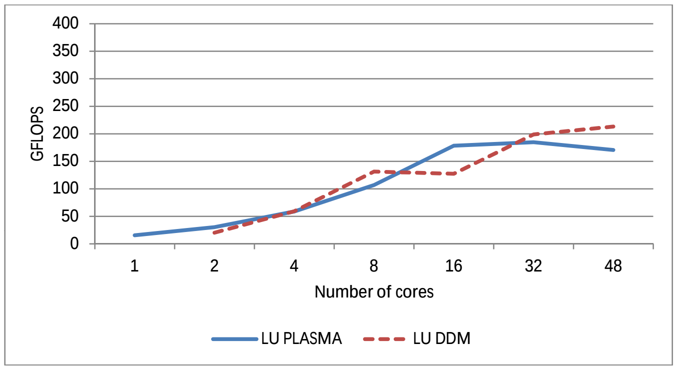
to the number of cores available, DDM on the hand uses fixed block size. We plan to add this feature to DDM as well.

DDM and PLASMA use the same highly optimized kernel operations, Thus the DDM benefit is mostly a result of the efficient scheduling of DDM Threads. DDM metadata allows it to construct the entire dynamic data-flow graph, with all the dynamic dependencies, at compile time. PLASMA, on the other hand, is determining the data dependencies at runtime which is time consuming. Thus, the higher performance for DDM is due to the creation of the entire Dynamic Data-Flow graph at compile time.

DDM has some erratic behavior after 8 cores for LU and Cholesky. DDM has been ported and tested in many multi-core systems and did not exhibit such behaviours before. Our test machine consists of four 12-core Multi-cores. Going from 8 cores to 16 cores we are utilizing a second Multi-core and then a third and fourth for 32 and 48. This transition creates some problems to the DDM which we are currently investigating. However, overall we achieve high performance for all three applications which reaches around $2\times$ better than the PLASMA for all three applications. Thus, we are confident that can make the case that Data-Flow concurrency as implemented in DDM could be a viable candidate for future HPC platforms.



(a)

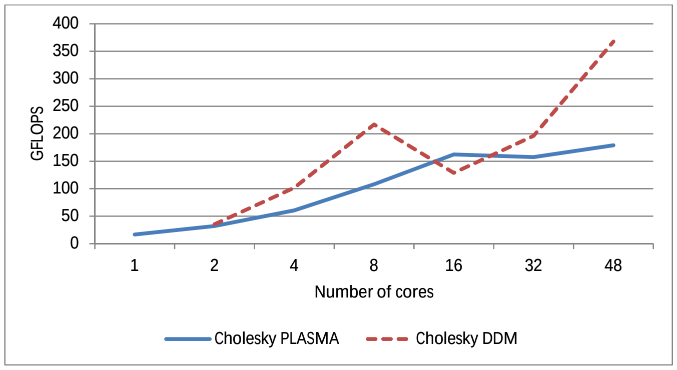


(b)

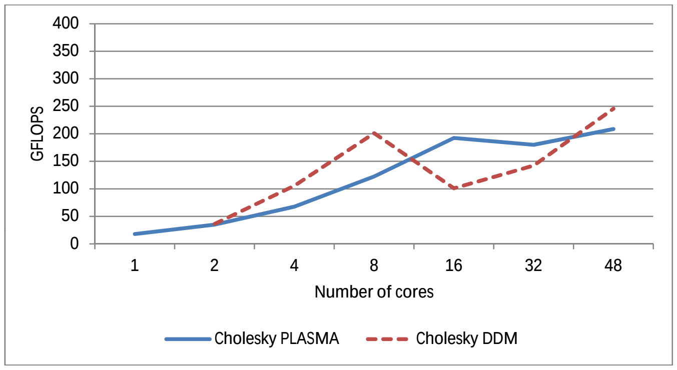
Fig. 11. Scalability of LU Decomposition as the number of cores increases for (a) $4k \times 4k$ matrix and (b) $8k \times 8k$ matrix

VI. Conclusions and Future Work

In order for the Data-flow model of computation to be regarded as a promising model for High Performance Computing (HPC) it has to first demonstrate that it can do as well or better than other existing approaches. The top 500 supercomputer ranking [24] is determined by the LINPACK benchmark. We implemented in DDM three applications (Matrix Multiple, LU and Cholesky) from the LINPACK benchmark and evaluated them against the performance and scalability of PLASMA, a state-of-the-art library. Our solutions for Matrix multiply, LU and Cholesky decompositions showed comparable scalability performance with PLASMA's, as the number of cores



(a)



(b)

Fig. 12. Scalability of Cholesky Decomposition as the number of cores increases for (a) $4k \times 4k$ matrix and (b) $8k \times 8k$ matrix

increased. For the 48 core configuration all three applications outperform the same PLASMA applications by a factor of 2X.

The results are very encouraging and show that DDM can handle the parallelization required for linear algebra applications for present and future multi and many-core systems. Thus, DDM should be further investigated as viable candidate for HPC. We are currently investigating the cause of the performance loss when we expand to more than one multi-core in LU and Cholesky.

In the future work we are planing to build a library that will contain such linear algebra functions as to improve the programmability and thus allow a broader set of programmers to exploit the large-scale parallelism by only changing the parallel library used to link with their codes. We plan to test this library in existing supercomputer class of systems.

References

[1] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammerling, J. Demmel, C. Bischof, and D. Sorensen. 1990. LAPACK: a portable linear algebra library for high-performance computers. In Proc. of the 1990 ACM/IEEE conference on Supercomputing. IEEE Computer Society Press, Los Alamitos, CA, USA, 2-11.

[2] Hadri, B.; Ltaief, H.; Agullo, E.; Dongarra, J.; , "Tile QR factorization with parallel panel processing for multicore architectures," Parallel & Distributed Processing (IPDPS), 2010 , vol., no., pp.1-10, 19-23 April 2010

[3] E. Agullo, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, J. Langou, and H. Ltaief. PLASMA Users Guide. Technical report, ICL, UTK, 2009.

[4] E. Anderson, Z. Bai, C. Bischof, L. Blackford, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. LAPACK Users' Guide. SIAM, 1992.

[5] Teraflux project. http://teraflux.eu

[6] 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 (HiPEAC '11). ACM, New York, NY, USA, 25-34.

[7] C. Kyriacou, P. Evripidou, and P. Trancoso. Data-driven multithreading using conventional microprocessors. IEEE Trans. Parallel Distrib. Syst., pp. 1176 - 1188, Oct. 2006.

[8] K. Stavrou, P. Evripidou, and P. Trancoso. Ddm-cmp: data-driven multithreading on a chip multiprocessor. In Proceedings of the 5th international conference on Embedded Computer Systems: architectures, Modeling and Simulation, SAMOS'05, pages 364 - 373, Berlin, Heidelberg, 2005. Springer-Verlag.

[9] J. R. Gurd, C. C. Kirkham, and I. Watson. The manchester prototype data-flow computer. Commun. ACM, 28(1):34-52, Jan. 1985.

[10] BLAS Technical Forum Standard. The International Journal of High Performance Computing Applications, 2001.

[11] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. 2008. Parallel tiled QR factorization for multicore architectures. Concurr. Comput. : Pract. Exper. 20, 13 (September 2008), 1573-1590.

[12] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. 2009. A class of parallel tiled linear algebra algorithms for multicore architectures. Parallel Comput. 35, 1 (January 2009), 38-53.

[13] Jack Dongarra and Robert Schreiber. 1990. Automatic Blocking of Nested Loops. Technical Report. University of Tennessee, Knoxville, TN, USA.

[14] D. Parello, O. Temam and Jean-Marie Verdun, On increasing architecture awareness in program optimizations to bridge the gap between peak and sustained processor performance Matrix-Multiply revisited., Supercomputing, 2002.

[15] Kyriakos Stavrou, Marios Nikolaides, Demos Pavlou, Samer Arandi, Paraskevas Evripidou, and Pedro Trancoso. 2008. TFlux: A Portable Platform for Data-Driven Multithreading on Commodity Multicore Systems. In Proceedings of the 2008 37th International Conference on Parallel Processing (ICPP '08). IEEE Computer Society, Washington, DC, USA, 25-34.

[16] P. Trancoso, K. Stavrou, and P. Evripidou. Ddmcpp: The data-driven multithreading c pre-processor. In Proceedings of the 11th Annual Workshop on the Interaction between Compilers and Computer Architecture (Interact-11), pages 3239, 2007.

[17] C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh, Basic Linear Algebra Subprograms for FORTRAN usage, ACM Trans. Math. Soft., 5 (1979), pp. 308–323.

[18] J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling, A set of Level 3 Basic Linear Algebra Subprograms, ACM Trans. Math. Soft., 16 (1990)

[19] A. YarKhan, J. Kurzak, and J. Dongarra. QUARK Users' Guide: QUeueing And Runtime for Kernels. University of Tennessee Innovative Computing Laboratory Technical Report ICL-UT-11-02, 2011.

[20] 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 (MICRO-44 '11)

[21] Perez J.M., Badia R.M., Labarta J.:A dependency-aware task-based programming environment for multi-core architectures. In Proceedings of 2008 IEEE International Conference on Cluster Computing, 2008.

[22] K. Kavi, J. Arul, and R. Giorgi, "Execution and Cache Performance of the Scheduled Dataflow Architecture," SPRINGER Journal of Universal Computer Science, vol. 6, pp. 948-967, 2000.

[23] R. Giorgi, Z. Popovic, N. Puzovic,"DTA-C: A Decoupled multi-Threaded Architecture for CMP Systems" ,Proc. IEEE SBAC-PAD, ISBN:0-7695-23014-1, Gramado, Brasil, Oct. 2007, pp. 263-270.

[24] Top500 Supercomputing Sites. 2012. top500.org. 20 Jul. 2012 "http://top500.org".