From Serial Loops to Parallel Execution on Distributed Systems

George Bosilca[1], Aurelien Bouteiller[1] Anthony Danalis[1], Thomas Herault[1], and Jack Dongarra[1 2]

[1] University of Tennessee, Knoxville TN 37996, USA,
[2] University of Manchester, Manchester, UK
{bosilca,bouteill,adanalis,herault,dongarra}@eecs.utk.edu

Abstract. Programmability and performance portability are two major challenges in today's dynamic environment. Algorithm designers targeting efficient algorithms should focus on designing high-level algorithms exhibiting maximum parallelism, while relying on compilers and run-time systems to discover and exploit this parallelism, delivering sustainable performance on a variety of hardware. The compiler tool presented in this paper can analyze the data flow of serial codes with imperfectly nested, affine loop-nests and if statements, commonly found in scientific applications. This tool operates as the front-end compiler for the *DAGuE* run-time system by automatically converting serial codes into the symbolic representation of their data flow. We show how the compiler analyzes the data flow, and demonstrate that scientifically important, dense linear algebra operations can benefit from this analysis, and deliver high performance on large scale platforms.

Keywords: compiler analysis; symbolic data flow; distributed computing; task scheduling

1 Introduction and Motivation

Achieving scientific discovery through computing simulation puts such high demands on computing power that even the largest supercomputers in the world are not sufficient. Regardless of the details in the design of future high performance computers, few would disagree that a) there will be a large number of nodes; b) each node will have a significant number of processing units; c) processing units will have a non-uniform view of the memory. Moreover, computing units in a single machine have already started becoming heterogeneous, with the introduction of accelerators, like GPUs.

This creates a complex environment (the "jungle"[3]) for application and library developers. A developer, whether a domain scientist simulating physical phenomena, or a developer of a numerical library such as ScaLAPACK [4] or

[3] Herb Sutter, "Welcome to the Jungle", 12-29-2011, http://herbsutter.com/2011/12/29/welcome-to-the-jungle/

PLASMA [12], is forced to compromise and accept poor performance, or waste time optimizing her code instead of making progress in her field of science. A better solution would be to rely on a run-time system that can dynamically adapt the execution to the current hardware. *DAGuE* [10], which deploys dynamic micro-task scheduling, has been shown [11] to deliver portable high performance, on heterogeneous hardware for a class of regular problems, such as those occurring in linear algebra.

Unfortunately, dynamic scheduling approaches commonly require application developers to use unfamiliar programming paradigms which hinders productivity and prevents widespread adoption. As an example, in *DAGuE*, the algorithms are represented as computation tasks decorated by symbolic expressions that describe the flow of data between tasks.

In this paper, we describe a compiler tool that automatically analyzes annotated sequential C code and generates the symbolic, problem size independent, data flow used by *DAGuE*. Through polyhedral analysis, our compiler represents the data flow of the input code as parameterized, symbolic expressions. These expressions enable each task to independently compute, at run-time, which other tasks it has dependencies with, thus defining the communication that must be performed by the system. We explain the process and the tools used to perform this translation. To the best of our knowledge, it is the first time that state-of-the-art, handcrafted software packages are outperformed by automatic data flow analysis coupled with run-time DAG scheduling on large scale distributed memory systems.

2 Related Work

Symbolic dependence analysis has been the subject of several studies [14, 18–20], mainly for the purpose of achieving powerful dependence testing, array privatization and generalized induction variable substitution, especially in the context of parallelizing compilers such as Polaris [5] and SUIF [17]. This body of work differs from the work presented in this paper in that our compiler does not focus on dependence testing, or try to statically find independent statements, in order to parallelize them. Our compiler derives symbolic parameterized expressions that describe the data flow and synchronization between tasks. Furthermore, we focus on programs that consist of loops and *if* statements, with calls to kernels that operate on whole array regions (i.e. matrix tiles), rather than operating on arrays in an element by element fashion. This abstracts away the access patterns inside the kernels, and simplifies the data flow equations enough that we can produce exact solutions using the Omega Test.

The polyhedral model [1, 3, 23], of which the Omega Test is part, has drawn a lot of attention in recent years, and newer optimization and parallelization tools, such as Pluto [6], have emerged that take advantage of it. However, unlike the work currently done within the polyhedral model, we do not use the dependence abstractions to drive code transformations, but rather export them in symbolic notation to enable our run-time to make scheduling and message exchange decisions.

In our work we harness the theoretical framework set by Feautrier [15] and Vasilache et al. [25] to compute the symbolic expressions that capture the data flow. By coupling for the first time this compiler theory with a distributed memory DAG scheduling run-time, we assert experimentally the significance of this approach in the context of high performance computing.

Finaly, Baskaran et. al [2] performed compiler assisted dynamic scheduling using compiler analysis. In their approach, the compiler generates code that scans and enumerates all vertices of the DAG at the beginning of the run-time execution. This has the same drawbacks as approaches, such as StarSS [21] and TBlas [24], that rely on pseudo-execution of the serial loops at run-time to dynamically discover dependencies between kernels. The overhead grows with the problem size and the scheduling is either centralized of replicated. In contrast, the symbolic data-flow and synchronization expressions our compiler generates can be solved at run-time by each task instance independently, in $O(1)$ time, without any regard to the location of the given instance in the DAG.

3 Compiler & Run-time Synergy

The goal of traditional standalone parallelizing compilers is to convert a serial program into a parallel program by statically addressing all the issues involved with parallel execution. However, dynamic environments call for a run-time solution. In our toolchain, the compiler static analysis scope is reduced to producing a symbolic representation to be interpreted dynamically by the scheduler during execution. Effectively, the compiler performs static data flow analysis to convert an affine input serial program into a Direct Acyclic Graph (DAG), with program functions (kernels) as its nodes, and data dependency edges between kernels as its edges. Then, the run-time is responsible for addressing all DAG scheduling challenges, including background MPI data transfers between distributed resources [10].

To drive the scheduler decisions, the compiler needs to produce more than a boolean value regarding the existence or not of a dependency. It has to identify the exact, symbolic, dependence relations that exist in the source code. From those, it generates parameterized symbolic expressions with parameters that take distinct values for each task. The expressions are such that the run-time can evaluate them for each task T_i independently of the task's place in the DAG. Also, the evaluation of each expression costs constant time (i.e., it does not depend on the size of the DAG). The result of evaluating each symbolic expression is another task T_j, to which data must be sent, or from which data must be received[4].

[4] Therefore, the only parameters allowed in a symbolic expression are the parameters of the execution space of T_i, and globals used in the input code.

4 Input and Output Formats

4.1 Input Format: Annotated Sequential Code

The analysis methodology used by our compiler allows any program with regular control flow and side-effect free functions to be used as input. The current implementation focuses on codes written in C, with affine loops and array accesses. The compiler front-end is flexible enough to process production codes such as the PLASMA library [12]. PLASMA is a linear algebra library that implements tile-based dense linear algebra algorithms.

```
for (k = 0; k < A.mt; k++) {
  Insert_Task(zpotrf, A[k][k], INOUT);
  for (m = k+1; m < A.mt; m++) {
    Insert_Task(ztrsm, A[k][k], INPUT, A[m][k], INOUT);
  }
  for (m = k+1; m < A.mt; m++) {
    Insert_Task(zherk, A[m][k], INPUT, A[m][m], INOUT);

    for (n = k+1; n < m; n++) {
      Insert_Task(zgemm, A[m][k], INPUT, A[n][k], INPUT, A[m][n], INOUT);
    }
  }
}
```

Fig. 1: Cholesky factorization in PLASMA

Figure 1 shows the PLASMA code that implements the Tiled Cholesky factorization [12] (with some preprocessing and simplifications performed on the code for improving readability). The figure shows the operations that constitute the Cholesky factorization POTRF, TRSM, HERK, and GEMM. The data matrix “A” is organized in tiles, and notation such as “A[m][k]” refers to a block of data (a tile), and not a single element of the matrix. Our compiler uses a specialized parser that can process hints in the API of PLASMA. We made this choice because in the PLASMA API the following is true: a) for every matrix tile passed to a kernel as a parameter, the parameter that follows it specifies whether this tile is read, modified, or both, using the special values `INPUT`, `OUTPUT` and `INOUT`; b) all PLASMA kernels are side-effect free. This means that they operate only on memory pointed to by their arguments, and that memory is not aliased.

Figure 1 contains four kernels, that correspond to the aforementioned operations. In the rest of this article we will use the terms *task* and *task class*. A *task class* is a specific kernel in the application that can be executed several times, potentially with different parameters, during the life-time of the application. `zpotrf` and `zgemm` are examples of task classes in Figure 1. A *task* is a particular, and unique, instantiation of a kernel during the execution of the application, with given parameters. In the example of the figure, task class `zpotrf` will be instantiated as many times as the outer loop `for(k)` will iterate, and thus we define the task class's *execution space* to be equal to the iteration space of the loop.

4.2 Compiler Output: Job Data Flow

```
for (k = 0; k < N; k++) {
    Insert_Task( Ta, A[k][k], INOUT );
    for (m = k+1; m < N; m++) {
       Insert_Task( Tb, A[k][k], INPUT, A[m][m], INOUT );
    }
}
```

Fig. 2: Pseudocode example of input code

The compiler outputs a collection of task classes and their dependency relation in a format we refer to as the Job Data Flow (JDF). Consider the simpler input defined in Figure 2. The compiler extracts (as described in Section 5.1) data flows between T_a and T_b in a symbolic way and outputs them in the definitions of task classes T_a and T_b in the JDF. The symbolic representation of each edge is such that every task $T_a(k)$ is able to determine the tasks $T_b(k, m)$ that need to use the data defined by $T_a(k)$ and vice-versa. Consider the particular edge due to `A[k][k]` flowing from $T_a(k)$ to $T_b(k, m)$. In the JDF, we use the following notation to store this flow edge in task class T_a:

```
A[k][k] -> ( k < N-1 ) ? A[k][k] Tb( k, (k+1)..(N-1) )
```

Conversely, tasks of the class T_b must be able to determine which task they depend on for input. In this case the same edge has the following form:

```
A[k][k] <- A[k][k] Ta(k)
```

The full JDF that the compiler produces to represent the example code of Figure 2 is shown in Figure 3. As can be seen in the figure, in addition to the execution space and the data flow edges, there are two more elements in a JDF file. First, there is an affinity definition of the form "`:A[k][k]`" which signifies that the corresponding task should be run in the MPI process that owns the corresponding data element. Second, there is a `BODY` that consists of C-language code that the run-time will invoke in order to execute the actual kernel that constitutes the body of a task.

From interpreting that JDF output, the *DAGuE* run-time can handle distributed memory execution efficiently, the scheduler can identify which tasks must communicate with which other, without consulting a centralized entity or traversing the whole problem DAG.

```
Ta(k)
  k = 0..N-1
  : A[k][k]

  A[k][k] <- (k==0)  ? A[k][k] : A[m][m] Tb(k-1, k)
          -> (k<N-1) ? A[k][k] Tb(k, (k+1)..(N-1))
          -> A[k][k]
BODY
  Ta(A[k][k]);
END

Tb(k,m)
  k = 0..N-1
  m = k+1..N-1
  : A[m][m]

  A[k][k] <- A[k][k] Ta(k)
  A[m][m] <- (k==0) ? A[k][k] : A[m][m] Tb(k-1, m)
          -> (m==k+1) ? A[k][k] Ta(m) : A[m][m] Tb(k+1, m)
BODY
  Tb(A[k][k], A[m][m]);
END
```

Fig. 3: Example Job Description Format

5 Extracting Symbolic Data Flow and Data Exchange

5.1 Omega Relations

The Omega test [22] is the library we use for manipulating the sets of affine constraints over integer variables that arise when performing the symbolic data-flow analysis necessary when converting from sequential code to JDF. An Omega *Relation* is a mapping between two tuples, defining the execution space of the source and sink task classes, as well as the conjunction of constraints for both execution spaces. Consider the example of compiler input given in Figure 2. The iteration space of T_a is the iteration space of outer loop `for(k)`; We denote this iteration space with the following Omega notation:

```
{[k] : 0 <= k <= N-1}
```

Such notation `{ [T] : C }`, where `T` is a tuple, and `C` is a conjunction of constraints, defines the ranges of values for the elements of `T` for which `C` is true. Similarly, we define the execution space of task class T_b to be:

```
{[k,m] : 0 <= k < N-1 && k+1 <= m <= N-1}
```

Here, the tuple has two elements, since T_b is enclosed by two loops. By examining the data-flow of the code, we can see that `A[k][k]` for example, will be modified (*defined*, in compiler parlance) by kernel T_a and then read (*used*, in compiler parlance) by kernel T_b. The corresponding relation due to `A[k][k]` flowing from $Ta(k)$ to $Tb(k,m)$ is:

```
{[k] -> [k',m] : 0 <= k < N-1 && k+1 <= m <= N-1 && k == k'}
```

In the example above, the term "`[k]`" represents the execution space of the source, T_a, and the term "`[k',m]`"[5] represents the execution space of the sink, T_b. In Omega parlance, this Relation has an *input variable* count of one and *output variable* count of two.

5.2 Interprocess Data Exchange

The symbolic data edges are associated with task classes, so that the run-time can use them to determine what messages need to be exchanged between tasks. In particular, for each task the run-time must determine the tasks that produced the input of this task and the tasks that will consume the output of this task. Therefore, the expressions stored in the JDF may contain only a) the parameters of the source task, b) symbolic and numeric constants, c) the logical constants "TRUE" and "FALSE".

Outgoing Messages After the compiler has finished processing the input source code, it will have a collection of Omega Relations describing the data flow edges from each task class to each other task class in the code. To produce the information needed by the run-time regarding the outgoing edges of a task T_i, we need to process all Relations of flow edges that have as source the task T_i. For every parameter that appears in the execution space of a Relation's destination, we solve the equality constraints in the conjunction of constraints for this parameter. Consider, as an example, the Relation:

```
{[k,m] -> [k'] : k' = m && 1+k = m && 1 <= m < N}
```

which describes the flow edge from `A[m][m]` in T_b to `A[k][k]` in T_a. This edge will be stored in the JDF of T_b as:

```
A[m][m] -> ((1+k)==m) ? A[k][k] Ta(m)
```

This way, when the run-time is processing task $T_b(7,8)$ for example, it can compute in $O(1)$ time that it needs to send tile `A[8][8]` to task $T_a(8)$. Also, when processing task $T_b(7,11)$, the run-time can compute that `A[11][11]` should not be sent to any instance of T_a, since the condition $(1+k) == m$ is not true (clearly, $1+7 \neq 11$).

If a destination parameter does not appear in any equality constraints in the conjunction, we determine the lower bound and upper bound of this parameter by solving the inequality constraints, and create a range of tasks that should be the receiver of this message. As an example, consider the flow edge from `A[k][k]` of T_a to `A[k][k]` of T_b which is described by the Relation:

```
{[k] -> [k',m] : k' = k && 0 <= k < m < N}
```

[5] Although both task classes share a common enclosing loop, we use different variables in the execution spaces (`k` and `k'`) because the dependency could be a loop carried dependency, so we have to allow the two iteration spaces to be independent.

In order to store this edge in the JDF expression of T_a, we need to express k' and m in terms of k (and constants), since k is the only parameter in the execution space of T_a. Therefore, this edge will be translated to the following information in JDF notation:

```
A[k][k] -> (k < N-1) ? A[k][k] Tb(k, k+1..N-1)
```

since the output parameter "m" does not appear in any equality constraints. In JDF syntax, expressions with ranges signify to the run-time that a broadcast operation must be performed.

Incoming Messages To produce the information regarding the incoming edges of a task T_i, we traverse the flow edges of all tasks searching for edges that have task T_i as the destination. For each such Relation, we compute the inverse, and then proceed with solving the inverse Relation for the output parameters, as we do for the outgoing edges.

5.3 Anti-dependence Edges

An anti-dependence edge exists between tasks T_{src} and T_{dst} if T_{src} uses a variable that T_{dst} defines, and T_{src} executes before T_{dst}[6]. In parallel execution, anti-dependence edges must be translated to synchronization edges, to avoid using wrong versions of the data. Ostensibly, anti-dependencies are not relevant in a distributed memory execution environment due to data copying. However, *DAGuE* can run on distributed memory machines, shared memory machines, or distributed memory clusters of shared memory nodes. Therefore handling anti-dependencies in a uniform and systematic way is important for preserving the semantics of the input serial algorithm.

Our compiler starts by recording all potential anti-dependencies as Omega Relations. Then, Algorithm 1 is used to minimize the number of synchronization edges by using data flow edges between tasks to eliminate the need for additional synchronization, wherever possible. This is possible because a data flow edge imposes a message exchange between tasks and therefore explicit synchronization.

6 Performance

Two metrics of performance are relevant in the context of this work. First, the performance of the compiler tool itself, and second, the performance of applications running under our system. We have tested the performance of our compiler tool by processing the dense linear algebra operations found in the PLASMA library, on hardware commonly found on average personal computers. The compilation time we have observed is in the order of 100ms when the anti-dependence minimization algorithm is not being used, and in the order of a few seconds when it is being used.

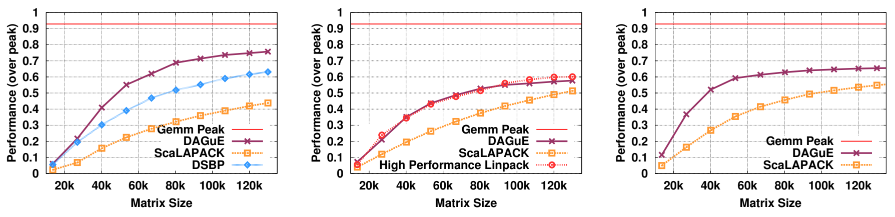
[6] Or more accurately, if there exists an execution path from T_{src} to T_{dst}.

```
Function $FinalizeAntiDependencies(I_G)$
Input: $I_G$, Input graph.
Result: Modifies $I_G$ by finalizing antidependencies.
begin
    foreach anti-dependence edge $E_a \in I_G$ do
        Let $G$ be a copy of $I_G$
        /* Unless otherwise specified all nodes and edges belong    */
        /* to $G$, and all operations are done on $G$.               */
        foreach pair of nodes $N_1, N_2$ do
            $\mathcal{R} \leftarrow \bigcup\{R_i : N_1 \xrightarrow[R_i]{} N_2\}$
            Replace all edges from $N_1$ to $N_2$ with single edge $N_1 \xrightarrow[\mathcal{R}]{} N_2$
        foreach Node $N_0$ do
            Let $(p_1, \ldots)$ be the parameters of the task that correspond to $N_0$
            /* Initiate $Cycle(N_0)$ with an empty (tautologic)        */
            /* Relation to self.                                         */
            $Cycle(N_0) \leftarrow \{[p_1, \ldots] \rightarrow [p_1, \ldots]\}$
        foreach Node $N_0$ do
            foreach $N_0 \xrightarrow[R_0]{} N_1 \ldots \xrightarrow[R_{n-1}]{} N_0$ do
                /* $N_0, N_1 \ldots N_0$ is a Cycle formed following flow,   */
                /* and/or anti-dependence edges.                        */
                $C \leftarrow R_0 \circ R_1 \circ \ldots \circ R_{n-1}$
                $T \leftarrow$ transitive closure of $C$
                $Cycle(N_0) \leftarrow Cycle(N_0) \bigcup T$
        $A \leftarrow FindTransitiveEdge(Source(E_a), \emptyset, \emptyset)$
        /* May remove $E_a$ if empty                                  */
        Change $E_a$ to $(E_a - A)$ in $I_G$
```

Algorithm 1: *FinalizeAntiDependencies*(I_G)

```
Function $FindTransitiveEdge(N_c, T, A)$
Input: $N_c$, the current node in the transitive edge; $T$ the transitive edge being
       built; $A$ the union of all transitive edges found until now.
Result: Union of the transitive edges that start at $N_c$ and end at Sink($E_a$)
begin
    /* Scope inlcludes the variables of $FinalizeAntiDependencies()$  */
    /* in Algorithm 1. This algorithm operates on $G$.                  */
    Mark $N_c$ as visited
    $T \leftarrow Cycle(N_c) \circ T$
    foreach Edge $N_c \xrightarrow[R_i]{} N_i$ s.t. $N_i$ is not visited do
        $T_{tmp} \leftarrow R_i \circ T$
        $A \leftarrow FindTransitiveEdge(N_i, T_{tmp}, A)$
    if $N_c = Sink(E_a)$ then
        return $A \bigcup T$
    else
        return $A$
```

Algorithm 2: *FindTransitiveEdge*(N_c, T, A)



(a) Cholesky Factorization (b) LU Factorization (c) QR Factorization

Fig. 4: Performance comparison on the Griffon platform (on 648 cores)

The performance of the DAGuE run-time has been extensively studied in related publications [8, 9, 7]. The goal of this paper is to present the compiler front-end of the system, so we present only a summary of performance results to demonstrate that our toolchain can automatically analyze, schedule and execute non-trivial algorithms, and deliver high performance at scale. Application performance results are relevant, because the scalability achieved by our run-time is enabled by the problem size independent algebraic expressions that our compiler generates to describe inter-task dependence edges.

For the experiments we present here, we used 81 dual socket Intel Xeon L5420 quad core processors at 2.5GHz for a total of 648 cores. Each node has 16GB of memory, and is interconnected to the others by a 20Gbs Infiniband network and runs Linux 2.6.24 (Debian Sid).

The benchmarks consist of three popular dense matrix factorizations: Cholesky, LU and QR. All three operations are implemented in the ScaLAPACK numerical library [4]. Moreover, the Cholesky factorization has been implemented in a more optimized way in the DSBP software [16], using static scheduling of tasks, and a data distribution more efficient. The LU factorization with partial pivoting is also solved by the well known High Performance Linpack benchmark (HPL [13]), used to measure the performance of supercomputers.

For our comparison, we implemented these operations within *DAGuE* by using the compiler presented in this paper to generate the JDF symbolic representation from the corresponding PLASMA files. The data distribution is not generated by automatic tools, but rather chosen by the human developer. For our experiments, we have distributed the initial data following the classical 2D-block cyclic distribution used by ScaLAPACK, and used our run-time engine to schedule the operations on the distributed data. The kernels consist of the BLAS operations referenced by the sequential codes, and their implementation was the most efficient available on this machine. The same kernel implementations for ScaLAPACK, HPL, DSBP, and our engine were used on each run.

Figure 4 presents the performance measured using our system (labeled as *DAGuE*) and ScaLAPACK, and when applicable DSBP and HPL, as a function of the problem size. All data is normalized to the theoretical floating point peak of the machine. A total of 648 cores participated in the distributed run, and the data was distributed according to a 9x9 2D block-cyclic grid. Tile size was tuned to provide the best performance on each setup. As the figures illustrate, on all benchmarks and for all problem sizes, our framework outperforms ScaLAPACK, and performs as well as the state of the art, hand-tuned codes for specific problems. Our system goes from the sequential code to the parallel run automatically, with very limited human involvement, but is still able to outperform DSBP, and competes with the HPL implementation on this machine.

7 Conclusion

In this paper we presented the compiler front end of the *DAGuE* system, more precisely how the compiler extracts the Symbolic Data Flow and Data Exchanges from the input code in order to expose additional information to the run-time. We outlined JDF, *DAGuE*'s internal problem-size independent representation of task generated by the compiler and used by the run-time to make all task scheduling and communication decisions. We showed how Relations produced using the Omega test can be converted into message and synchronization requests for the run-time, and how the synchronization edges can be reduced to the minimum necessary set. Using this critical information exposed by the compiler, the run-time can take more effective decisions about inter-nodes data transfers and about how to schedule tasks in order to maximize the available parallelism not only locally but remotely. Experimental results confirm that serial codes processed by our system can match, or outperform, highly optimized, state of the art, hand tuned, distributed linear algebra codes, such as Scalapack, libSCI and HPL.

References

1. Ancourt, C., Irigoin, F.: Scanning polyhedra with do loops. In: Proceedings of ACM PPoPP '91. pp. 39–50. Williamsburg, VA (1991)
2. Baskaran, M.M., Vydyanathan, N., Bondhugula, U.K.R., Ramanujam, J., Rountev, A., Sadayappan, P.: Compiler-assisted dynamic scheduling for effective parallelization of loop nests on multicore processors. In: Proceedings of ACM PPoPP '09. pp. 219–228. Raleigh, NC (2009)
3. Bastoul, C.: Code Generation in the Polyhedral Model Is Easier Than You Think. In: Proceedings of IEEE PACT '04. pp. 7–16. Antibes Juan-les-Pins, France (2004)
4. Blackford, L.S., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., Whaley, R.C.: ScaLAPACK Users' Guide. Society for Industrial and Applied Mathematics, Philadelphia, PA (1997)
5. Blume, W., Doallo, R., Eigenmann, R., Grout, J., Hoeflinger, J., Lawrence, T., Lee, J., Padua, D., Paek, Y., Pottenger, B., Rauchwerger, L., Tu, P.: Parallel programming with polaris. IEEE Computer 29, 78–82 (December 1996)
6. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: A practical automatic polyhedral parallelizer and locality optimizer. In: Proceedings of ACM PLDI '08. pp. 101–113. Tucson, AZ (2008)
7. Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Haidar, H., Herault, T., Kurzak, J., Langou, J., Lemarinier, P., Ltaief, H., Luszczek, P., YarKhan, A., Dongarra, J.: Distributed-Memory Task Execution and Dependence Tracking within DAGuE and the DPLASMA Project. Tech. Rep. 232, LAWN (Sep 2010)

8. Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Haidar, A., Herault, T., Kurzak, J., Langou, J., Lemarinier, P., Ltaief, H., Luszczek, P., YarKhan, A., Dongarra, J.: Flexible development of dense linear algebra algorithms on massively parallel architectures with DPLASMA. In: IEEE PDSEC-11. Anchorage, AK (2011)
9. Bosilca, G., Bouteiller, A., Danalis, A., Herault, T., Lemarinier, P., Dongarra, J.: DAGuE: A generic distributed dag engine for high performance computing. In: HIPS-11. Anchorage, AK (2011)
10. Bosilca, G., Bouteiller, A., Danalis, A., Herault, T., Lemarinier, P., Dongarra, J.J.: DAGuE: A generic distributed DAG engine for high performance computing. Parallel Computing (2011), to appear, http://dx.doi.org/10.1016/j.parco.2011.10.003
11. Bosilca, G., Bouteiller, A., Hérault, T., Lemarinier, P., Saengpatsa, N.O., Tomov, S., Dongarra, J.J.: Performance portability of a gpu enabled factorization with the dague framework. In: IEEE CLUSTER. pp. 395–402 (2011)
12. Buttari, A., Langou, J., Kurzak, J., Dongarra, J.J.: A class of parallel tiled linear algebra algorithms for multicore architectures. Parallel Comput. Syst. Appl. 35, 38–53 (2009)
13. Dongarra, J.J., Luszczek, P., Petitet, A.: The LINPACK benchmark: Past, present and future. Concurrency Computat.: Pract. Exper. 15(9), 803–820 (2003)
14. van Engelen, R.A., Birch, J., Shou, Y., Walsh, B., Gallivan, K.A.: A unified framework for nonlinear dependence testing and symbolic analysis. In: Proceedings of ACM ICS '04. pp. 106–115. Malo, France (2004)
15. Feautrier, P.: Dataflow analysis of array and scalar references. International Journal of Parallel Programming 20, 23–53 (1991), 10.1007/BF01407931
16. Gustavson, F.G., Karlsson, L., Kågström, B.: Distributed SBP cholesky factorization algorithms with near-optimal scheduling. ACM Trans. Math. Softw. 36(2), 1–25 (2009)
17. Hall, M.W., Anderson, J.M., Amarasinghe, S.P., Murphy, B.R., Liao, S.W., Bugnion, E., Lam, M.S.: Maximizing multiprocessor performance with the SUIF compiler. IEEE Computer 29, 84–89 (December 1996)
18. Kyriakopoulos, K., Psarris, K.: Data dependence analysis techniques for increased accuracy and extracted parallelism. International Journal of Parallel Programming 32, 317–359 (August 2004)
19. Kyriakopoulos, K., Psarris, K.: Nonlinear Symbolic Analysis for Advanced Program Parallelization. IEEE Transactions on Parallel and Distributed Systems 20, 623–640 (May 2009)
20. Maydan, D.E., Hennessy, J.L., Lam, M.S.: Efficient and exact data dependence analysis. In: Proceedings of ACM PLDI '91. pp. 1–14. Toronto, Ontario (1991)
21. Perez, J., Badia, R., Labarta, J.: A dependency-aware task-based programming environment for multi-core architectures. In: Proceedings of IEEE Cluster Computing. pp. 142 –151 (2008)
22. Pugh, W.: The omega test: a fast and practical integer programming algorithm for dependence analysis. In: Proceedings of the ACM/IEEE SC'91. pp. 4–13
23. Quilleré, F., Rajopadhye, S., Wilde, D.: Generation of efficient nested loops from polyhedra. Int. J. Parallel Program. 28, 469–498 (October 2000)
24. Song, F., YarKhan, A., Dongarra, J.: Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems. In: Proceedings of ACM/IEEE SC'09
25. Vasilache, N., Bastoul, C., Cohen, A., Girbal, S.: Violated dependence analysis. In: Proceedings of ACM ICS '06. pp. 335–344. Cairns, Queensland, Australia (2006)