From Serial Loops to Parallel Execution on Distributed Systems

Abstract

Programmability and performance portability have been, and continue to be, two major challenges in scientific computing. So far, auto-parallelization has provided sub-optimal solutions, as auto-generated code tends to under-perform and is commonly limited to shared memory environments. In this paper, we build upon an existing run-time system designed to efficiently schedule and execute fine-grain task-based applications on heterogeneous, distributed memory environments. We present an automatic compiler tool for analyzing the data flow of serial codes with imperfectly nested, affine loop nests containing if statements. This tool functions as the front-end, source code compiler of the run-time system by automatically converting input serial codes into the run-time's internal representation of the task system that represents the input code. We demonstrate how this representation captures the semantics of the serial code in a symbolic and problem-size independent manner. We describe the data flow analysis, and show how serial loops and dependence edges can be converted into the symbolic representation. We demonstrate the efficiency of the approach, through examples borrowed from dense linear algebra problems.

Keywords compiler analysis, symbolic data flow, distributed computing, task scheduling

1. Introduction and Motivation

Achieving scientific discoveries, 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 to become heterogeneous, with the introduction of accelerators, like GPUs. While all these changes are expected to happen, according to trends and necessities recognized by computer scientists and hardware vendors, it is not the ideal scenario for application and library developers. A developer, whether a domain scientist simulating physical phenomena, or a developer of a numerical library such as ScaLAPACK [6] or PLASMA [13], 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 and utilize the expertise of computer scientists when making decisions about the ordering of computations, their distributions on the computing resources, in order to increase performance, e.g. through cache reuse and communication/computation overlapping. Unfortunately, such systems commonly require application developers to use novel languages, outside their programming experience, or do not scale enough to satisfactorily utilize modern high performance systems.

Previous work resulted in a dynamic task scheduling, run-time system that is designed to address these limitations. In the remainder of this paper we will refer to this system as *xyz* for the purpose of the double-blind review process. The full name, as well as citations to the appropriate articles will be added in the final version. In this paper, we describe the compiler front end of *xyz* that automatically translates annotated C code, containing serial loops, into a task based representation of the problem, enabling the *xyz* run-time engine to schedule and execute the tasks on large scale, distributed memory, parallel systems. We explain the process and the tools used to perform this translation and we demonstrate through experimental results that high performance can be achieved with minimal user involvement.

[Copyright notice will appear here once 'preprint' option is removed.]

2. Toolchain

In contrast with parallelizing compilers, we are not trying to convert a serial program into a parallel program by statically addressing all the issues involved with parallel execution. Instead, we combine a compiler component, which is the subject of this paper, and a run-time component to achieve parallel execution. We rely on the compiler component to statically analyze the data flow of the input program and produce a symbolic representation of the program and delegate all other decisions to the run-time. The input program representation generated by the compiler, contains a collection of parameterized tasks, and symbolic information describing the data flow between them. In the rest of this article, we will refer to this representation as the Job Data Flow (JDF). Effectively, the compiler performs static data flow analysis to convert an 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.

Figure 1 shows a schematic outline of the overall system architecture. The compiler component is shown as a shaded box in the figure. We first briefly describe the other components in order to provide some context for the reader, and motivate and justify the existence of the compiler.

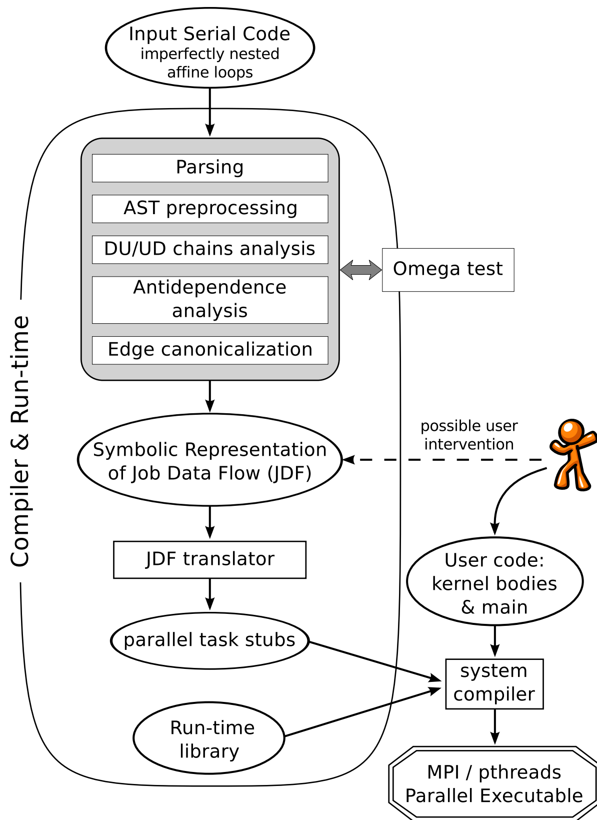


Figure 1: Toolchain schematic

2.1 Run-time

The JDF, generated by the compiler, is a human-readable representation of the Direct Acyclic Graph of tasks, one example being given in Figure 4 below. As illustrated in Figure 1, the application developer may edit this representation to tune some parameters (like the association between a task and the data distribution, or to introduce priority hints to the scheduler). Then, the JDF is processed by the JDF translator to build a set of libraries holding the parallel job stubs. These stubs enable the application developer to instantiate, at run-time, DAG generators with fixed parameters. A DAG generator is an opaque object used by the run-time engine to A) compute, given a specific task, its successors and predecessors; B) generate tasks on demand when they become ready to be scheduled; C) call the corresponding kernels.

The stubs and the DAG generator instantiated from them for given parameters, are problem-size independent: they use symbolic algebraic representations to evaluate the expressions that appear in the JDF, when needed.

The run-time schedules the work on the computing resources, using different components: 1) the DAG generator, to compute the successors and predecessors of each task when they are completed; 2) hardware information (like memory affinity, levels and sizes of caches) that is collected at run-time using the hwloc library [12]; 3) progress status and data requests received by other processes of the run-time, running on remote nodes.

The system uses MPI as its underlying communication system. All processes are launched by the MPI run-time. User programs can be entirely processed and executed by our toolchain, or they can be written as normal MPI applications that make calls into code that has been been processed by our system. When an application needs to execute some operations that have been processed by our toolchain, all MPI processes of a communicator will call the generated stub to instantiate a local DAG generator with the desired parameters. Then, our run-time will orchestrate the scheduling of all tasks on all available computing units of all nodes belonging to the MPI communicator. Processing units include all cores of the machine running the MPI process, and GPUs if some are available. Providing the kernels to compute the operation of the task on a particular hardware is the responsibility of the user. When the DAG is completed (the last task that runs locally on this MPI process is done, and this MPI process does not need to pass along more data), the main function of the run-time engine returns, and the user can resume the classical MPI behavior.

The run-time defines where a task should be executed through an affinity between data and tasks, expressed algebraically. To specify where a specific task should run, when doing a distributed run, the user needs to specify how the data is distributed between the processors. Each of the data referenced in the JDF is specified by a couple of user-defined functions to compute what processor holds the data, and where in its memory. Our framework provides such data localization functions for most of the classical data distributions (e.g. Block-cyclic).

The distributed run-time engine is able to schedule work in a fully distributed manner, because all processes have the capacity to examine any part of the whole DAG at any time during execution, thanks to the symbolic representation. The successors or predecessor of any task can be discovered in bounded time, and the memory requirements to do so do not depend on the problem size, for the same reason. Exploration of the DAG can be done either by following the successor relationship of tasks (to compute which task on which processes need to receive the result of the execution of a local task), or the predecessor relationship (to compute which task, on which process, will generate the awaited data). As a consequence, the distributed run-time relies on the symmetry of the dependence relationship as it is expressed in the JDF, and built from the symbolic data flow analysis.

2.2 Source Code Compiler

The compiler accepts, as input, serial programs with imperfectly nested, affine loops that make calls to side-effect-free functions and can include *if* statements with conditions that involve constants and induction variables of enclosing loops. In the rest of this paper, we will refer to these functions as *kernels*. To analyze the input program, the compiler needs information about the behavior of the kernels with respect to their arguments. The arguments of interest are expected to be compound entities, such as tiles or blocks of a matrix, and can have any shape, as long as they can be described with an MPI data-type. The current implementation of the compiler uses a custom C parser that we built on-top of an openly available ANSI-C grammar implemented in yacc[1]. Since dense linear algebra is a natural application domain of a DAG system like ours, we have specialized our parser to take advantage of library specific syntax used by the PLASMA project [2] developed at the University of Tennessee. The codes developed by PLASMA are open source and include hints that reveal whether a kernel parameter is *used*, *defined*, or both *used* and *defined* by the kernel (`IN`, `OUT` and `INOUT` respectively). An example of such syntax will be explained in Section 3, and can be seen in Figure 2. As can be seen in Figure 1, the parser, which is the only annotation specific and language specific component, is a modular front-end of the compiler and can be adapted to accept different annotations such as SMPSS-like [25] pragma directives. In principle, the front-end could perform interprocedural analysis and discover this information without the need

[1] Jeff Lee, 1985, draft version of the ANSI C standard

for annotations, but this is much harder technically and outside the scope of this paper.

The parser generates an Abstract Syntax Tree (AST) representing the program and creates a simple symbol table. The node structure of the AST is a simplified version of the one used by the Open64 compiler [1].

Before starting the data flow analysis, the compiler performs some preprocessing of the AST. The goal of this step is to detect early potential failures in the subsequent steps due to the presence of poorly formatted code and collect information that will be used in the data flow analysis. Loop canonicalization is an example transformation performed in this step, and we are planning to add generalized induction variable recognition and substitution [30]. In addition, this step collects information about the different "tasks" that exist in the code. From the perspective of the compiler, a task is a region of the code, that has been annotated by the user as such. The parser does not need special annotations for identifying tasks, as it can extract this information from PLASMA-specific syntax. In PLASMA, kernels are passed to the scheduler, QUARK [32], for execution, using a specific API as shown in Figure 2. The custom parser used by our compiler interprets these API calls and records which parts of the code constitute tasks. The information that the compiler collects about every task includes the arguments passed to the kernels, the task's *execution space*, or in other words the polyhedron defined by the iterations space of its enclosing loops, and whether the arguments are *used* or *defined* by the kernel.

The following and final steps are the symbolic data flow analysis steps. The symbolic analysis is the most important part of the compiler and is discussed in greater detail in Section 4 below.

3. Input and Output Formats

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.

3.1 Input

PLASMA is a linear algebra library similar in functionality with LAPACK, but it implements tile-based algorithms instead of panel-based ones. Tile-based algorithms express a higher degree of parallelism, on blocks of data that enable a longer duration for the kernels, allowing the run-time system to recover more recovery of the communication time with computation.

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

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

Figure 2: Cholesky factorization in PLASMA

Figure 2 shows the PLASMA code that implements the Tiled Cholesky factorization [13] (with minor preprocessing and simplifications performed on the code for improving readability). The code consists of four imperfectly nested loops with a maximum nesting depth of three. In the body of each loop there are calls to the kernels that implement the four mathematical operations that constitute the Cholesky factorization POTRF, TRSM, HERK, and GEMM. As the name of the algorithm suggests, the data matrix "A" is organized in tiles, and notations such as "A[m][k]" refer to a block of data (a tile), and not a single element of the matrix. As mentioned earlier, our compiler currently uses a specialized parser that can process hints in the API of PLASMA. This choice was influenced by the following two facts:

1. For every parameter passed to a kernel, that corresponds to a matrix tile, the parameter that follows it specifies whether this tile is read, modified, or both, using the special values `INPUT`, `OUTPUT` and `INOUT`.
2. All PLASMA kernels are side-effect free. This means that they operate on, and potentially change, only memory pointed to by their arguments. Also, this memory does not contain overlapping regions, i.e. the arguments are not aliased.

3.2 Tasks and task instances

Our system comprises a DAG based task scheduling engine. This means that applications need to be represented as collections of tasks and the data dependencies between the tasks. In order to be generic and problem size independent, the symbolic representation generated by our compiler and used by the run-time of our system, JDF, must store the tasks and the dependencies between them in a succinct, symbolic way that can be interpreted quickly at run-time.

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

Figure 3: Pseudocode example of input code

As an example of compiler input, let us consider the simpler code of Figure 3. It contains two kernels: `Ta` and `Tb`. In the rest of this article we will use the terms *task* and *task instance*. A *task* is a specific kernel in the application that can be executed several times, potentially with different parameters, during the life-time of the application. `Ta` and `Tb` are examples of tasks in Figure 3. A *task instance* is a particular, and unique, instantiation of a kernel during the execution of the application, with given parameters. In the example of Figure 3 task `Ta` will be instantiated as many times as the outer loop `for(k)` will be executed, and thus we define the task's *execution space* to be equal to the iteration space of the loop. We denote this iteration space with the following 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 `Tb` to be:

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

Here, the tuple has two elements, since `Tb` is enclosed by two loops. By examining the data-flow of the two tasks, we can see that `A[k][k]` for example, will be modified (*defined*, in compiler parlance) by task `Ta` and then read (*used*, in compiler parlance) by task `Tb`. 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}
```

The reason why k is limited to strictly less than $N-1$ and not equal to $N-1$ (as the execution space of `Ta` would suggest) is because, in the last iteration of the outer loop (when $k = N-1$), task `Tb` will not execute (because $m = k+1 \implies m \nless N$). For the iterations of the outer loop where $k < N-1$, task `Tb` will use `A[k][k]` in all iterations of the inner loop.

3.3 Output

The run-time system targets distributed memory execution, where task instances have to identify which other task instances they must communicate with, in a fast and problem size independent way. Therefore, the information we associate with task `Ta` must contain a symbolic representation of this edge, such that every task instance `Ta(k)` is able to determine the instances `Tb(k,m)` that need to use the data defined by `Ta(k)`. The JDF uses the following notation to represent this flow edge in task `Ta`:

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

Conversely, the instances of task `Tb` must be able to determine which task instances they will depend on for input. However, in this case the same edge has the following much simpler form in JDF:

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

The full JDF that the compiler produces to represent the example code of Figure 3 is shown in Figure 4. 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 a task.

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

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

Figure 4: Example Job Description Format

4. Symbolic Data Flow Analysis

The ultimate goal of the compiler presented in this paper is to provide the run-time of the system with information about the data-flow of the input program. In particular, the compiler analyzes the data flow between the kernels (tasks) of the input program to extract information that can be used to create a Direct Acyclic Graph where the nodes are task instances and the edges represent data exchanges between the nodes. This information is then used by the run-time to enable parallel execution of the tasks, while preserving the semantics of the serial execution. It is important to note that no part of the run-time ever creates, or traverses the DAG. That would cause inefficiencies, since the DAG size is problem size dependent. Rather, the information generated by the compiler is in the form of parameterized symbolic expressions with parameters that take distinct values for each task instance. Thus the run-time is able to evaluate the expressions of each task instance independently of the task's place in the DAG and do so in constant time. In essense, every task instance $T(...)$ can discover all the task instances $T^p(...)$ that produce data that are used by $T(...)$ and all task instances $T^c(...)$ that consume data generated by $T(...)$ in $O(1)$ time per task instance. In graph terms, each node of the DAG can discover its immediate neighborhood in constant time per neighboor.

4.1 Omega Relations

After the compiler preprocesses the AST built by the front-end, it collects information about the tasks that exist in the program as well as all the uses and definitions of all relevant variables. Consequently, it considers that each pair of occurrences of a variable, such that at least one is a definition, is a potential data dependence edge. For example, when analyzing the code of Figure 3, the compiler will record a potential edge from `A[k][k]` in task `Ta` to `A[m][m]` in task `Tb`. To test if a potential edge is indeed a data dependence, the compiler takes into account the execution spaces of the tasks that are involved and formulates an Omega [26] *Relation*. Therefore, the aforementioned edge, will be formulated as the following Omega Relation:

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

An Omega Relation is a mapping between two tuples, defining the execution space of the source and sink tasks, as well as the conjunction of constraints for both execution spaces. In the example above, the term "`[k]`" represents the execution space of the source task, `Ta`, and the term "`[k',m]`" represents the execution space of the sink task, `Tb`. In Omega parlance, this Relation has an *input variable* count of one and *output variable* count of two. Although both tasks 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. The first two constraints: "`0 <= k < N`" and "`k'+1 <= m < N`" are imposed by the loop bounds. The constraints "`k <= k'`" is imposed by the fact that the source and destination are both enclosed by the `for(k)` loop and the destination is after the source in the body of the loop. This means that the results of the source are visible by the destination in the iteration that generated them, or any other future

† Viewed as a polyhedron, this execution space includes all the points with integer coordinates, in the X, Y plane, that are enclosed by the triangle defined by the Y axis, the $y = N-1$ line and the $y = x+1$ line.

iteration. Finally, the constraint "`k == m`" is a result of the demand that "`A[k][k] == A[m][m]`". After the compiler has formulated this Relation, it calls into the Omega library to simplify it. In this particular example, Omega will simplify it into the Relation: `{[k] -> [k',m] : FALSE }` since, clearly, `m` cannot be greater than `k` and equal to `k` at the same time and therefore there does not exist a true data flow edge from `A[k][k]` in task `Ta` to `A[m][m]` in task `Tb` at any point in the execution of this example code.

4.2 Loops & Data Dependence

More formally, there is a data dependence from task T_s to task T_d if and only if both following conditions hold:

1. Both tasks access (for definition or use) the same memory location, and at least one access is a definition.
2. In the serial code, task T_s executes before T_d.

To refer to specific iterations, when dealing with loop nests, we use iteration vectors[2]. The iteration vector, $\mathbf{I}$, of a specific iteration of a loop nest of depth n, is a vector of n integers, $\mathbf{I} = \{i_1, i_2, ..., i_n\}$, such that the loop at depth k ($1 \leq k \leq n$) is at iteration i_k. We say that iteration $\mathbf{I}$ executes not later than iteration $\mathbf{J}$ (which we hereafter denote as: $\mathbf{I} \leq \mathbf{J}$) if and only if:

$$\forall k : i_k > j_k \ \exists m : m < k \ \wedge \ i_m < j_m$$

Also, we say that iteration $\mathbf{I}$ executes before iteration $\mathbf{J}$ (which we hereafter denote as: $\mathbf{I} < \mathbf{J}$) if and only if:

$$\mathbf{I} \leq \mathbf{J} \ \wedge \ \exists k : 1 \leq k \leq n \ \wedge \ i_k < j_k$$

Using iteration vectors, we can refine the second condition, mentioned above, for the existence of a data dependence between tasks that share n enclosing loops. Specifically, There is a dependence edge between task T_s in loop $\mathbf{I_s}$ and task T_d in loop $\mathbf{I_d}$ if and only if either of the following conditions is true:

- $\mathbf{I_s} \leq \mathbf{I_d}$ and there is a path from T_s to T_d in the body of the inner most common loop.
- $\mathbf{I_s} < \mathbf{I_d}$

4.3 Potential Data Dependence Edges

Parallelizing compilers analyze serial programs trying to identify, for any two statements of interest, whether, or not, a data dependence exists. This information is important, because statements with data dependencies cannot be executed in parallel. However, our compiler is not an automatic parallelizer. Instead, it identifies the exact, symbolic, parameterized relation between tasks. This information is then used by the run-time to schedule individual instances of tasks based on the data flow restrictions of the individual task instances, and not based on the restrictions of the tasks as they appear in the serial code. The compiler forms Omega Relations about all potential data dependence edges. That is, *flow edges* (use following definition), *output edges* (definition following definition) and *anti-dependencies* (definition following use). It does so by traversing the uses and definitions of the input program according to the following steps:

1. Create a fake `ENTRY` task, that defines all variables that are used in the program, before any use.
2. Create a fake `EXIT` task, that uses all variables that are defined in the program, after all definitions.
3. For every variable of interest that is accessed in tasks T_s and T_d (with at least one access being a definition), create an Omega Relation in the following way:
   (a) Set the Relation's input variable count to the number of loops that enclose T_s.
   (b) Name the input variables using the induction variables of the loops that enclose T_s.
   (c) Set the output variable count to the number of loops that enclose the task that uses the variable, T_d.
   (d) Name the output variables using the induction variables of the loops that enclose T_d. For all loops that enclose both T_s and T_d, add an apostrophe to the output variable name.
   (e) For each input and each output variable, add constraints equivalent to the bounds of the corresponding loop.
   (f) For every common enclosing loop, add to the Relation a logical conjuntion of constraints to guarantee that T_s executes no later than T_d.
      i. If there is a path from T_s to T_d in the inner most loop, then the iteration vectors must be such that $\mathbf{I_s} \leq \mathbf{I_d}$. E.g., for loops k, m, n:
      $(k < k') \vee$
      $(k = k' \wedge m < m') \vee$
      $(k = k' \wedge m = m' \wedge n \leq n')$
      ii. If there is no path from T_s to T_d in the inner most loop, then the iteration vectors must be such that $\mathbf{I_s} < \mathbf{I_d}$. E.g., for loops k, m, n:
      $(k < k') \vee$
      $(k = k' \wedge m < m') \vee$
      $(k = k' \wedge m = m' \wedge n < n')$
   (g) Create constraints to guarantee that the array indices that appear in the variable occurance at T_s are equal to the corresponding indices that appear in the variable occurance at T_d. For example if the potential edge is from `A[k][m]` of T_s to `A[m'][n]` of T_d, then the conjunction of constraints will be:
   `(k==m' && m==n)`.

After this algorithm is completed, all potential data dependence edges are formulated as Omega Relations. The next step consists of taking into consideration conflicts between edges, to calculate the actual data flow of the problem.

4.4 Finalizing Flow Edges

When T_s defines the variable and T_d uses it, the generated Relation will describe a potential flow edge (also known as true dependence). When both tasks define the variable, the generated Relation will describe a potential output edge. When T_s uses the variable and T_d defines it, the generated Relation will describe a potential anti-dependence edge. To discover the final Relations of the various dependence edges, we need to consider which edges *kill* other edges and subtract the corresponding Relations. For example, in the code of Figure 3, at iteration $\mathbf{I_s} = \{k = 0, m = N-1\}$ task T_b writes into `A[N-1][N-1]`. Since at iteration $\mathbf{I_d} = \{k = N-1\}$ task T_a will read `A[N-1][N-1]` and $\mathbf{I_s} < \mathbf{I_d}$ the aforementioned algorithm will generate a potential flow edge. However, the data that was written by T_b in $\mathbf{I_s}$ will not survive until iteration $\mathbf{I_d}$ since it will be overwritten by a different instance of T_b at iteration $\mathbf{I_s^{'}} = \{k = 1, m = N-1\}$ and then again at $\mathbf{I_s^{''}} = \{k = 2, m = N-1\}$ all the way to $\mathbf{I_s^n} = \{k = N-2, m = N-1\}$.

This process is similar to what traditional compilers do to perform data flow analysis on scalar variables. For scalar variables a compiler needs to know, given a definition, which uses can be

[2] More details on data dependence analysis and iteration vectors can be found in popular compiler textbooks [20].

reached from it (DU-chains) and given a use, which definitions can reach it (UD-chains). However, in the case of our compiler, we are interested in the exact symbolic Relation that reveals what conditions need to hold at the definition and what conditions need to hold at the use for a given edge to exist. Not merely the information that a definition *may*, or *must* reach a use. This symbolic relation will be output directly, to be used by the scheduling engine to compute the predecessors and successors of a given Task Instance, when needed. Figure 5 shows a subset of the flow edges and output edges of the code shown in Figure 3. This subset corresponds to the example discussed above.

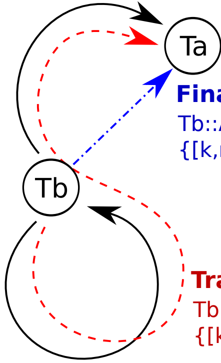


Figure 5: Example Data-Flow

As explained, part of the potential flow edge is masked (or, killed) by the output edge. However, the constraints of the output edge are more restrictive than the constraints of the flow edge. Thus, part of the flow edge "survives" the output edge and reaches the use at task T_a. This means that there are iteration vectors **I** and **J** such that values written by task T_b in **I** are read by task T_a in **J**. To calculate the constraints under which the flow edge reaches T_a, we compute the *transitive edge* (shown as a dashed, red arrow), by composing the output edge with the potential flow edge and then subtract the transitive edge from the potential flow edge. This results in the final flow edge (shown as a dash-dotted, blue arrow). In the general case, we calculate the actual constraints of each flow edge by applying algorithm 1 to all potential flow edges.

4.5 Relations to Data Exchanges

As stated before, the goal of our compiler is to analyze a serial code and generate symbolic expressions that capture the data flow of the code. These expressions will be stored in the JDF that describes the input code and will be used by the run-time to assess what messages need to be exchanged between the tasks. The expressions should be such that the run-time will be able to evaluate them for each task instantiation $T_i(\ldots)$ independently of the task's place in the DAG. Also, the evaluation of each expression should cost constant time (i.e., it should not depend on the size of the DAG). The result of evaluating each symbolic expression will be another task

return $FinalizeFlowEdges(I_G)$
Input: I_G Input graph, where each node corresponds to a task as it appears in the serial code, and edges are all flow dependence or output dependence edges of the serial code.
Result: Modifies I_G to remove killed (impossible) flow dependence edges, or reduce their definition space to its minimum.
begin
  foreach *flow edge* $E_f \in I_G$ **do**
    foreach *output edge* $E_o \in I_G : E_o.src = E_f.src$ **do**
      if $\exists E'_f : E'_f.src = E_o.src \wedge E'_f.dst = E_f.dst$ **then**
        $E_t \leftarrow E'_f \circ E_o$
        $E_f \leftarrow E_f - E_t$

Algorithm 1: $FinalizeFlowEdges(I_G)$

instantiation $T_j(\ldots)$, to which data must be send, or from which data must be received. Therefore, the only parameters allowed in a symbolic expression are the parameters of the execution space of $T_i(\ldots)$, and globals used in the input code (constants). So, if T_i is enclosed by loops with induction variables "k" and "m" (i.e., $T_i(k,m)$), only the globals and variables "k" and "m" can appear in T_i's data edges symbolic expressions.

In the example shown in Figure 5, there is a flow edge from `A[m][m]` in task T_b to `A[k][k]` in task T_a. The finalized Omega Relation that describes this edge is:

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

However, when the run-time is processing an instance $T_b(k,m)$ of task T_b, the only parameters available will be "k" and "m", so the Omega Relation must be expressed in terms of only these two parameters. Conversely, the information associated with task T_a, with respect to this edge, is the inverse of the Omega Relation for T_b, and it must be expressed in terms of the parameter "k" which is the only parameter in the execution space of task instances $T_a(k)$.

The process of converting an Omega Relation to the appropriate form so that it can be stored in the JDF and used by the run-time to assess outgoing and incoming messages is described in the following sections.

4.5.1 Outgoing Messages

In the general case, to produce the information 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. Solving the equality constraints for a destination parameter is a recursive process that terminates when the resulting expression contains nothing other than parameters of the source task, symbolic constants and numeric constants, or the logical constants "TRUE", or "FALSE". As an example, the Relation:

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

for the flow edge of Figure 5, would be stored in the JDF of task T_b as:

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

Therefore, when the run-time is processing task instance $T_b(7,8)$ for example, it can compute in $O(1)$ time that it needs to send tile `A[8][8]` to task instance $T_a(8)$. 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 $T_a(k)$ to `A[k][k]` of $T_b(k, m)$ which is described by the Relation:

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

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.

4.5.2 Incoming Messages

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

4.6 Finalizing Anti-dependence Edges

When task T_s uses a variable that task T_d defines, and there is an execution path from T_s to T_d, then an anti-dependence edge exists from T_s to T_d. The algorithm outlined in Section 4.3 will detect and record all the anti-dependence edges as Omega Relations, just as it will do with the flow and output edges.

While anti-dependence edges have no meaning for a serial code, they are very important for maintaining the semantics of the serial code when executing tasks in parallel. Namely, if there is anti-dependence edge from T_s to T_d the run-time must ensure that T_d executes only after T_s has finished (or executes on a node with different address space than T_s and therefore the two tasks do not interfere with the memory of one another). Therefore, anti-dependence edges will not generate data exchanges between tasks, but they will generate synchronization control messages.

Finalizing an anti-dependence edge, E_a, uses a very different approach than the finalization of flow edges. Namely, output dependencies play no role, but flow dependencies and all anti-dependencies other than E_a do. Intuitively, we can understand the finalization methodology by observing the following facts:

1. If there is a potential anti-dependence edge from T_s to T_d and there is also a flow edge between the same tasks, then T_d will have to wait for T_s to complete anyway, since it is waiting for data, so there is no need for additional synchronization.
2. If there are two potential anti-dependence edges from T_s to T_d, a single synchronization message is necessary.
3. If there is an edge from T_s to T_m and an edge from T_m to T_d, then T_d will have to wait for T_s due to transitivity. Therefore, transitive edges can also render anti-dependences unnecessary.
4. If there is an edge E' (flow, anti, or transitive) that partially overlaps with the potential anti-dependence edge E_a we are trying to finalize, then we subtract the Relation of E' from the Relation of E_a. We consider that two edges partially overlap, if the conjunctions of constraints in the Relations of the edges define partially overlapping polyhedra.

Function $RemoveAntiDependence(I_G)$
Input: I_G, Input graph.
Result: Modifies I_G to remove redundant anti-dependence edges, or minimize their definition space.
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 $(p_1, \ldots, p_n)$ be the parameters of the task that correspond to N_0
    - `/* Initiate` $Cycle(N_0)$ `with an empty (tautologic) Relation to self. */`
    - $Cycle(N_0) \leftarrow \{[p_1, \ldots, p_n] \rightarrow [p_1, \ldots, p_n]\}$
  - **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
      - $Cycle(N_0) \leftarrow Cycle(N_0) \bigcup T$
  - Remove all edges from any node to itself
  - $A \leftarrow FindTransitiveEdge(Source(E_a), \emptyset, \emptyset)$
  - `/* May remove` E_a `if empty */`
  - Change E_a to $(E_a - A)$ in I_G

Algorithm 2: $RemoveAntiDependence(I_G)$

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
- `/* This algorithm uses the variables of */`
- `/* the` $RemoveAntiDependence$ `Function */`
- `/* in Algorithm 2. It operates on` G`. */`
- Mark N_c as visited
- **if** $N_c \neq Source(E_a)$ **then**
  - $T \leftarrow Cycle(N_c) \circ T$
- **foreach** *Edge* $N_c \xrightarrow[R_i]{} N_i$ *s.t.* N_i *is not visited* **do**
  - $T \leftarrow R_i \circ T$
  - $A \leftarrow FindTransitiveEdge(N_i, T, L)$
- **if** $N_c = Sink(E_a)$ **then**
  - **return** $A \bigcup T$
- **else**
  - **return** A

Algorithm 3: $FindTransitiveEdge(N_c, T, L)$

```
for(k = 0; k < NT; k++)
  Task( POTRF, A[k][k], INOUT )

  for(m = k+1; m < NT; m++)
    Task( TRSM, A[k][k], IN,
                A[m][k], INOUT )

  for(n = k+1; n < NT; n++)
    Task( SYRK, A[n][k], IN,
                A[k][k], INOUT )
    for(m = n+1; m < NT; m++)
      Task( GEMM, A[m][k], IN,
                  A[n][k], IN,
                  A[m][n], INOUT )
```

(a) Cholesky Factorization

```
for(k = 0; k < NT; k++)
  Task( GETRF, A[k][k], INOUT,
               T[k][k], OUT)
  for(m = k+1; m < NT; m++)
    Task( TSTRF, A[k][k], INOUT,
                 A[m][k], INOUT,
                 T[m][k], INOUT )
  for(n = k+1; n < NT; n++)
    Task( GESSM, A[k][k], IN,
                 T[k][k], IN,
                 A[k][n], INOUT )
    for(m = n+1; m < NT; m++)
      Task( SSSSM, A[m][k], IN,
                   T[m][k], IN,
                   A[k][n], INOUT,
                   A[m][n], INOUT )
```

(b) LU Factorization

```
for(k = 0; k < NT; k++)
  Task( GEQRT, A[k][k], INOUT,
               T[k][k], OUT)
  for(m = k+1; m < NT; m++)
    Task( TSQRT, A[k][k], INOUT,
                 A[m][k], INOUT,
                 T[m][k], INOUT )
  for(n = k+1; n < NT; n++)
    Task( ORMQR, A[k][k], IN,
                 T[k][k], IN,
                 A[k][n], INOUT )
    for(m = n+1; m < NT; m++)
      Task( SSMQR, A[m][k], IN,
                   T[m][k], IN,
                   A[k][n], INOUT,
                   A[m][n], INOUT )
```

(c) QR Factorization

Figure 6: Sequential codes used as input to the compiler

In the remaining of this section, we present a formal algorithm that uses the aforementioned observations to systematically reduce the amount of synchronization messages needed due to anti-dependence edges.

The algorithm assumes that there is a graph I_G where each node corresponds to a task as it appears in the serial code, and edges are all flow or anti-dependence edges of the serial code. Note that this graph exists only during the compilation stage and has no relation to the DAG that connects the task instances during execution. Each edge of the input graph I_G is tagged with the Omega Relation that defines under what conditions the edge holds, and the parameter space of the involved tasks. Algorithm 2 presents the main function to remove the anti-dependence edges that give redundant information. It operates mainly on a copy of the input graph, to compute how each of the anti-dependence edges can be reduced. To do so, it uses the function defined in Algorithm 3 that computes the transitive Relation that encompasses all dependence cases already present in the graph, without taking into account the edge that the first algorithm is trying to reduce. In the following algorithms, the term "$Cycle(N_i) \leftarrow \cdots$" denotes that the information right of the arrow is stored in node N_i's tag "$Cycle$" and the term "$\cdots \leftarrow Cycle(N_i)$" denotes that the information stored in that tag is used. Also $Source(E_i)$ and $Sink(E_i)$ denote the node that edge E_i starts from or leads to respectively.

5. 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. For the dense linear algebra operations found in the PLASMA library, the execution time of the compiler tool is in the order of 100ms, on hardware commonly found on average personal computers. The slowest execution of the compiler that we have observer so far was 220ms when analyzing *pzgerbb*, an operation that invokes eight different kernels and has a maximum loop nesting level of three.

The performance that applications can achieve by using our system has been extensively studied in previous work, that we will cite in the final version of this paper. 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 we use to describe inter-task dependence edges. The ability of our compiler to go beyond dependence testing by generating these algebraic expressions enables the end-result, i.e., high performance execution on distributed memory platforms.

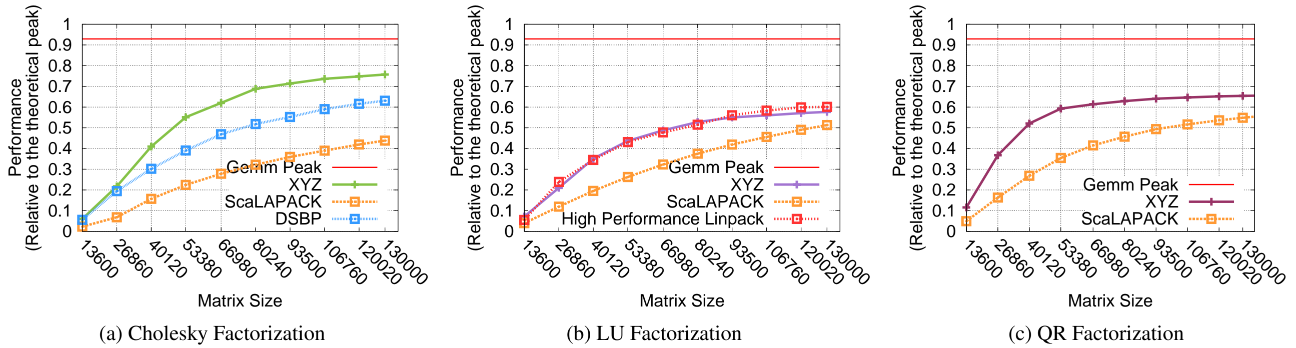
The experiments we present have been conducted on the Griffon platform, which is one of the clusters of Grid'5000 [9]. We used 81 dual socket Intel Xeon L5420 quad core processors at 2.5GHz to gather 648 cores. Each node has 16GB of memory, and is interconnected to the others by a 20Gbs Infiniband network. Linux 2.6.24 (Debian Sid) is deployed on these nodes.

The benchmark consists of three popular dense matrix factorization: Cholesky, LU and QR. The Cholesky factorization solves the problem $Ax = b$, where A is symmetric and positive definite. It computes the real lower triangular matrix with positive diagonal elements L such that $A = LL^T$. The QR factorization offers a numerically stable way of solving full rank underdetermined, overdetermined, and regular square linear systems of equation. It computes Q and R such that $A = QR$, Q is a real orthogonal matrix, and R is a real upper triangular matrix. The LU factorization with partial row pivoting of a real matrix A has the form $A = PLU$ where L is a real unit lower triangular matrix, U is a real upper triangular matrix, and P is an optional permutation matrix.

All these three operations are implemented in the ScaLAPACK numerical library [6]. Moreover, the Cholesky factorization has been implemented in a more optimized way in the DSBP software [18], using static scheduling of tasks, and a data distribution more efficient, and the LU factorization with partial pivoting is also solved by the well known High Performance Linpack benchmark (HPL, [14]), used to measure the performance of supercomputers.

For our comparison, we implemented these operations within our system by using the compiler presented in this paper to generate the JDF symbolic representation from the simple sequential algorithms that are given in Figure 6. The exact form of the kernels is as they appear in the PLASMA library, but in the figure we present a simplified form to increase clarity and reduce the space. As mentioned in Section 2.1, 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 kernels implementations for ScaLAPACK, HPL, DSBP, and our engine were used on each run.

Figure 7 presents the performance measured using our system (labeled as *xyz*) and ScaLAPACK, and when applicable DSBP and HPL, as function of the problem size. All data is normalized to the



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

Figure 7: Performance comparison on the Griffon platform (on 648 cores)

theoretical floating point operation peak of the machine. 648 cores participated to the distributed run, and the data was distributed according to a 9×9 2D block-cyclic grid. Block/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 perform 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.

Moreover, we performed experiments on the Kraken system of the University of Tennesse and National Institute for Computational Science (NICS). In these experiments we compared against libSCI, Cray's specifically tuned numerical library for this machine. We used up to 3,072 cores and our system performed similarly to libSCI within a 10% interval.

On the different machines, our compiler coupled with our runtime significantly outperformed standard algorithms, and competed closely, sometimes favorably, with state-of-the-art optimized versions of similar algorithms, without any further tuning process involved when porting the code between wildly different platforms.

6. Related Work

Data flow analysis has been the target of numerous studies for several decades now and established theory is best covered by compiler textbooks [20, 24, 31]. Symbolic dependence analysis in particular, has also been the subject of several studies [8, 15, 16, 21–23, 28, 30] 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 [7] and SUIF [19]. However, this body of work is very different from the work presented in this paper for several reasons. First, our work does not focus on dependence testing. Our system is not a parallelizing compiler, and as such it is not trying to prove statically whether two statements are data dependent or not, in order to parallelize them. Our compiler derives symbolic expressions that describe the data flow in a parameterized way such that a given task instance can evaluate which other task instances it must exchange data, or synchronize with, due to data dependencies. Second, we primarily 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 [26].

The polyhedral model [3, 5, 17, 27], 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 [10, 11], have emerged that take advantage of it. While this is very related to our work, there are some key differences. First, as mentioned above, we mainly focus on codes that make calls to user specified kernels. As a result, our compiler does not try to identify the granularity of parallelism. This is decided by the user who implements the kernels. Second, unlike the work currently done within the polyhedral model, we do not use the dependence abstractions to drive code transformations, but rather export the symbolic notation that will enable our run-time to make scheduling and message exchange decisions. Finaly, Baskaran et. al [4] performed compiler assisted dynamic scheduling using similar compiler analysis and target applications as we do in this work. The main difference is that 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 adds overhead that grows with the problem size and impedes distributed execution, or at least creates a centralized bottleneck. In our approach, the compiler generates symbolic, algebraic expressions to describe the dependence and control edges between tasks. These expressions can be solved at run-time by each task instance independently, without any regard to the location of the given instance in the DAG and in $O(1)$ time. This enables distributed execution that is not impeded by a centralized entity, or problem dependent serial overheads. This fact also differentiates our approach from alternative approaches [25, 29, 32] that rely on pseudo-execution of the serial loops at run-time to dynamically discover dependencies between kernels.

7. Conclusion

In this paper we presented the compiler front end of the *xyz* system, and discussed how it is integrated into the system's toolchain. We outlined JDF, the internal problem-size independent representation of task systems, which is generated by the compiler and used by the run-time to make all task scheduling and communication decisions. We analyzed how symbolic data dependence analysis is performed on input serial codes consisting of imperfectly nested, affine loops and *if* statements and how Relations produced using the Omega test can be processed to produce a minimal set of edges corresponding to data exchanges and synchronizations.

Finally, we presented experimental results that confirm that annotated serial codes processed by our system can achieve scalability and reach comparable performance, or outperform highly optimized, state of the art, hand tuned, distributed linear algebra codes, such as Scalapack, libSCI and HPL. By preserving high performance on hardware architectures with distinct network and processors characteristics, our system demonstrated that performance portability is achievable under some constraints.

References

[1] Open64 compiler. `http://www.open64.net`.

[2] 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*, volume 180, 2009.

[3] Corinne Ancourt and François Irigoin. Scanning polyhedra with do loops. In *Proceedings of the third ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '91, pages 39–50, New York, NY, USA, 1991. ACM.

[4] Muthu Manikandan Baskaran, Nagavijayalakshmi Vydyanathan, Uday Kumar Reddy Bondhugula, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Compiler-assisted dynamic scheduling for effective parallelization of loop nests on multicore processors. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 219–228, New York, NY, USA, 2009. ACM.

[5] Cedric Bastoul. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 7–16, Washington, DC, USA, 2004. IEEE Computer Society.

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

[7] William Blume, Ramon Doallo, Rudolf Eigenmann, John Grout, Jay Hoeflinger, Thomas Lawrence, Jaejin Lee, David Padua, Yunheung Paek, Bill Pottenger, Lawrence Rauchwerger, and Peng Tu. Parallel programming with polaris. *IEEE Computer*, 29:78–82, December 1996.

[8] William Blume and Rudolf Eigenmann. Demand-driven, symbolic range propagation. In *Proceedings of the 8th International Workshop on Languages and Compilers for Parallel Computing*, LCPC '95, pages 141–160, London, UK, 1996. Springer-Verlag.

[9] Raphaël Bolze, Franck Cappello, Eddy Caron, Michel Daydé, Frédéric Desprez, Emmanuel Jeannot, Yvon Jégou, Stephane Lanteri, Julien Leduc, Noredine Melab, Guillaume Mornet, Raymond Namyst, Pascale Primet, Benjamin Quetier, Olivier Richard, El-Ghazali Talbi, and Iréa Touche. Grid'5000: A large scale and highly reconfigurable experimental grid testbed. *International Journal of High Performance Computing Applications*, 20(4):481–494, 2006.

[10] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *International Conference on Compiler Construction*, CC'08/ETAPS'08, pages 132–146, Berlin, Heidelberg, 2008. Springer-Verlag.

[11] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM.

[12] François Broquedis, Jérôme Clet Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications. In IEEE, editor, *PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, Pisa Italy, 02 2010.

[13] A. Buttari, J. Langou, J. Kurzak, and J. J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Comput. Syst. Appl.*, 35:38–53, 2009.

[14] J. J. Dongarra, P. Luszczek, and A. Petitet. The LINPACK benchmark: Past, present and future. *Concurrency Computat.: Pract. Exper.*, 15(9):803–820, 2003.

[15] Michael P. Gerlek, Eric Stoltz, and Micharl Wolfe. Beyond induction variables: detecting and classifying sequences using a demand-driven ssa form. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 17:85–122, January 1995.

[16] Gina Goff, Ken Kennedy, and Chau-Wen Tseng. Practical dependence testing. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, PLDI '91, pages 15–29, New York, NY, USA, 1991. ACM.

[17] Martin Griebl. Automatic parallelization of loop programs for distributed memory architectures. FMI, University of Passau, Habilitation Thesis, 2004.

[18] Fred G. Gustavson, Lars Karlsson, and Bo Kågström. Distributed SBP cholesky factorization algorithms with near-optimal scheduling. *ACM Trans. Math. Softw.*, 36(2):1–25, 2009.

[19] Mary W. Hall, Jennifer M. Anderson, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, Edouard Bugnion, and Monica S. Lam. Maximizing multiprocessor performance with the suif compiler. *IEEE Computer*, 29:84–89, December 1996.

[20] Ken Kennedy and John R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[21] Konstantinos Kyriakopoulos and Kleanthis Psarris. Data dependence analysis techniques for increased accuracy and extracted parallelism. *Int. J. Parallel Program.*, 32:317–359, August 2004.

[22] Konstantinos Kyriakopoulos and Kleanthis Psarris. Nonlinear Symbolic Analysis for Advanced Program Parallelization. *IEEE Transactions on Parallel and Distributed Systems*, 20:623–640, May 2009.

[23] Dror E. Maydan, John L. Hennessy, and Monica S. Lam. Efficient and exact data dependence analysis. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, PLDI '91, pages 1–14, New York, NY, USA, 1991. ACM.

[24] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[25] J.M. Perez, R.M. Badia, and J. Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *Cluster Computing, 2008 IEEE International Conference on*, pages 142 –151, 29 2008-oct. 1 2008.

[26] William Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 4–13, New York, NY, USA, 1991.

[27] Fabien Quilleré, Sanjay Rajopadhye, and Doran Wilde. Generation of efficient nested loops from polyhedra. *Int. J. Parallel Program.*, 28:469–498, October 2000.

[28] Radu Rugina and Martin Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27:185–235, March 2005.

[29] Fengguang Song, Asim YarKhan, and Jack Dongarra. Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 19:1–19:11, New York, NY, USA, 2009. ACM.

[30] Robert A. van Engelen, J. Birch, Y. Shou, B. Walsh, and Kyle A. Gallivan. A unified framework for nonlinear dependence testing and symbolic analysis. In *Proceedings of the 18th annual international conference on Supercomputing*, ICS '04, pages 106–115, New York, NY, USA, 2004. ACM.

[31] Michael Joseph Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[32] A. YarKhan, J. Kurzak, and J. Dongarra. QUARK users' guide: Queueing and runtime for kernels. Technical Report ICL-UT-11-02, University of Tennessee, Innovative Computing Laboratory, February 2011.