Università di Pisa
Dipartimento di Informatica

Technical Report: TR–11–13

Targeting multi cores by structured programming and data flow

M. Aldinucci°, L. Anardu⋆, M. Danelutto⋆, P. Kilpatrick†, M. Torquati⋆

Dept. Computer Science, Univ. Pisa⋆
Dept. Computer Science, Univ. Torino°
Dept. Computer Science, Queen's Univ. Belfast†

ADDRESS: Largo B. Pontecorvo 3, 56127 Pisa, Italy. TEL: +39 050 2212700 FAX: +39 050 2212726

Targeting multi cores by structured programming and data flow

M. Aldinucci°, L. Anardu⋆, M. Danelutto⋆, P. Kilpatrick†, M. Torquati⋆

Dept. Computer Science, Univ. Pisa⋆
Dept. Computer Science, Univ. Torino°
Dept. Computer Science, Queen's Univ. Belfast†

Abstract

Data flow techniques have been around since the early '70s when they were used in compilers for sequential languages. Shortly after their introduction they were also considered as a possible model for parallel computing, although the impact here was limited. Recently, however, data flow has been identified as a candidate for efficient implementation of various programming models on multi-core architectures. In most cases, however, the burden of determining data flow "macro" instructions is left to the programmer, while the compiler/run time system manages only the efficient scheduling of these instructions. We discuss a structured parallel programming approach supporting automatic compilation of programs to macro data flow and we show experimental results demonstrating the feasibility of the approach and the efficiency of the resulting "object" code on different classes of state-of-the-art multi-core architectures. The experimental results use different base mechanisms to implement the macro data flow run time support, from plain pthreads with condition variables to more modern and effective lock- and fence-free parallel frameworks. Experimental results comparing efficiency of the proposed approach with those achieved using other, more classical parallel frameworks are also presented.

Keywords: macro data flow, structured programming, multi-core, shared memory, lock-free queues

1 Introduction

Data flow is a computing model that has been around since the earliest days of computer science research activities [1, 2, 3, 4]. In classical imperative models the instruction to execute is determined by the value of a special register–the program counter–which is normally incremented each time an instruction is executed or updated in the case of branch instructions. By contrast, in data flow the instruction(s) to be executed is (are) identified as those having all their

input data available. Therefore data flow programs are *graphs* of data flow instructions.

Each instruction has a function code (the "program" to be executed on the input data), one or more input *tokens* (the input data to be processed, expressed as data plus a boolean flag stating whether the data is available or not) and one or more *destinations* (expressed as instruction and token identifiers where the computed data will be stored/delivered). A macro data flow program execution proceeds by assigning the input data to the graph input tokens and then executing a cycle where:

1. *fireable* instructions are located in the graph (a fireable instruction is a data flow instruction with all input tokens available, that is, with the boolean flags of all the input tokens set to true).

2. fireable instructions are scheduled for execution on a *functional unit*. Normally, it is assumed that a number of functional units are available, each capable of executing any of the "programs" represented by the data flow instruction code. Ideally, scheduling of a data flow instruction to a functional unit implies that the corresponding input tokens are also delivered to the functional unit.

3. the results of the execution of the fireable instructions are stored in the locations denoted by the instruction destinations. This activity again involves transmission of computed result tokens from the functional units to the data flow graph storage.

The cycle is run until no more fireable instructions exist–*de facto* the program termination.

It is clear that properly programmed data flow graphs represent programs with minimal execution time, as the only dependencies preventing execution of an instruction are those representing *true data dependencies*, which are the only ones that cannot be ignored, suppressed or reduced (without actually changing the algorithm).

The possibility of executing fireable data flow instructions according to arbitrary scheduling on multiple functional clearly presents the possibility of parallel execution of data flow programs *for free*. In fact, in the '80s a number of activities have been undertaken aimed at designing and implementing data flow processors [5, 6, 7, 8]. Those processors were usually build of a *matching unit* associated with the data flow graph storage and responsible for the scheduling of fireable data flow instructions on functional units, and of a number of functional units, computing fireable instructions and returning to the matching unit the data flow instruction output tokens along with their destinations in the data flow graph.

Those architectures turned out to be quite expensive, however, and unable to deliver significant performance improvements over existing imperative architectures. The available technology (at that time we were still far behind VLSI integration as we know it today) required huge investments for the development

and the marketing of these unconventional architectures. In particular, data communications to and from functional units turned out to be far less efficient than those needed to implement the Von Neumann bottleneck communications associated with conventional imperative programming models. Moreover, the difficulty of extracting data flow programs from existing applications or providing high level data flow programming languages suitable for the development of new applications without incurring huge overheads prevented their wide-scale adoption and contributed also to their downfall.

The situation is radically different today. As we shall discuss in Sec. 2, technology advances, in particular those related to the new shared memory multi-core architecture models and to the implementation of thread/lightweight processes, allow the data flow programming model to be reconsidered as an efficient model to support pervasive parallel programming models.

A number of different programming frameworks (e.g.[9, 10, 11]) have already been developed using (aspects of) data flow concepts to support the implementation of compilers and run time systems whose main aim is to keep busy the increasing number of cores–per socket and per system–sharing the same memory hierarchy. Most of these frameworks heavily rely on the programmer's ability to identify data flow instructions, or *independent tasks* in modern parlance, while providing limited support for automatic derivation of data flow graphs (with notable exceptions in the area of stream processing) from some high level, possibly declarative descriptions of algorithms/applications.

In this paper we propose a new dataflow-based methodology targeting shared memory multi- and many-core architectures and we show experimental results demonstrating the feasibility and the efficiency of the proposed approach. The methodology is build on two distinct pillars: i) the efficient implementation of generic parallel *macro data flow* interpreters, where the term macro refers to the possibility to have task-level data flow instructions [12], and ii) the automatic compilation of macro data flow graphs from structured parallel programming environments based on parallel design pattern/algorithmic skeleton concepts.

The remainder of the paper is structured as follows: Sec. 2 discusses how efficient parallel macro data flow interpreters may be implemented on top of modern multi-core architectures. Sec. 3 details how applications may be compiled to macro data flow graphs to feed these parallel macro data flow interpreters. Sec. 4 presents results obtained with three parallel macro data flow interpreter implementations on different target architectures and with different kinds of synthetic and real application kernels. Finally, Sec. 5 discusses related work and outlines the main differences with the proposed approach and Sec. 6 assesses the main features of our approach and discusses future activities.

2 Macro data flow targeting multi-cores

Modern multi-core architectures are characterized by the following key features: i) the availability of an increasing number of independent cores sharing part of the levels of memory hierarchy; ii) different kinds of hardware support for multi-

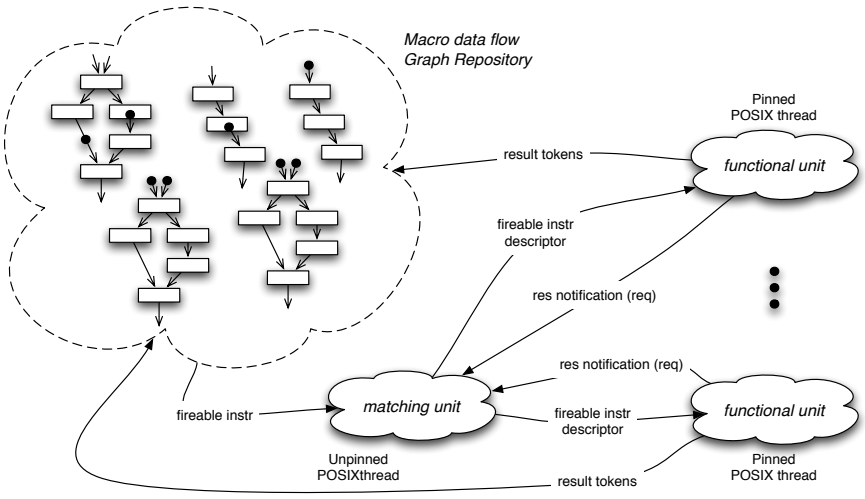

Figure 1: Architectural design of the parallel macro data flow interpreter

threading and/or lightweight multi-tasking; and iii) different kinds of hardware support for automatic cache coherency.

These features match quite well the typical needs of a macro data flow interpreter. First, data movement between logical matching unit and functional units may exploit the shared memory hierarchy: input and output token *pointers* may be moved around rather than actual, possibly large, data segments. Second, the availability of increasingly many efficient multi-threading facilities may be exploited to implement, in software rather than in hardware, efficient matching and functional units with an increased degree of flexibility and customizability of the interpreter. Third, the memory hierarchy can be exploited to realize token circulation without incurring significant implementation complexity and run-time overheads thanks to cache-coherent shared data (token) accesses, as these accesses are natively regulated by the correct implementation of the macro data flow interpreter computing the macro data graph (program). Last but not least, macro data flow graphs of parallel applications may provide a number of simultaneously fireable data flow instructions suitable for feeding a large number of software functional units implemented on the available cores.

Following these principles, we designed a parallel macro data flow interpreter targeting multi-cores as follows:

- Macro data flow instructions are represented as tuples

  $$\langle g_{id}, i_{id}, f_{id}, T^*, D^* \rangle$$

  where g_{id} is a graph identifier, i_{id} is the instruction identifier, f_{id} is the function identifier, T^* is the set of input tokens and D^* is the set of output

destinations. In turn, input tokens are represented as

$$\langle pointer,\ boolean_presence_bit \rangle$$

pairs, and output tokens as

$$\langle graph_id,\ instruction_id,\ input_token_number \rangle$$

tuples.

- A global *instruction pool* hosting macro data flow graphs is defined. Each time an input data set is available, a copy of the application macro data flow graph with a fresh graph id is inserted into the pool, with input data placed in the appropriate input tokens by the input manager thread.

- A *matching unit* thread is started. It accepts notifications of new tokens available from either the input manager thread (input data) or by the interpreter threads (output tokens from computed macro data flow instructions–see below). It checks whether the destination instruction of the tokens has completed its input token sets, and if so, inserts a descriptor of these new fireable instructions in a fireable instruction queue.

- A number of *interpreter* threads are started, usually as many as there are available cores, each pinned to one of the cores. Each interpreter thread fetches a fireable instruction from the fireable instruction queue, executes it, stores the output tokens in the appropriate destinations and notifies the matching unit thread with the id(s) of the updated instruction(s).

The general schema of this parallel macro data flow interpreter is outlined in Fig. 1.

This parallel macro data flow interpreter adheres to a quite standard master-workers schema, which is quite typical for this kind of interpreter (see e.g. [13]). However:

- Communications between matching unit/thread and functional units/threads are much more efficient, as data is not actually moved: only pointers are moved, resulting in smaller size memory copies.

- The notification mechanism of new tokens available avoids continuous scanning of the instruction pool by the matching thread.

- The adoption of a *macro* data flow model succeeds in mitigating the effects of communications and synchronizations between matching unit and functional unit threads, as the amount of work to be performed to execute a fireable macro data flow instruction is large, significantly different from the amount of work considered in the '80s when simple arithmetic operations were considered as "programs" to be executed when computing a data flow instruction.

- Similarly, the adoption of a *macro* data flow model allows efficient use of the cache hierarchy, as long instruction computations succeed in making good use of local caches (i.e., long computations successfully amortize the time spent to load data into local caches, which is not the case when simple, very short computations are performed). This positive effect may be further enhanced by collapsing entire data flow subgraphs into a single macro data flow instruction in a completely automated and autonomic way upon observing that the grouped instructions have too fine a grain to be efficiently executed as separate instructions.

- Exploitation of the memory subsystem may be enhanced by adopting suitable affinity scheduling policies, such that instructions using "huge" input tokens are possibly scheduled to places where those tokens had been previously consumed/produced, in the hope that copies still exist in local caches, suitably updated through hardware cache coherence protocols.

We implemented three different versions of this interpreter, differing mainly in the communication mechanisms used, namely i) shared memory data structures, protected with Pthread condition variables and mutexes, ii) Unix pipes and iii) FastFlow lock-free communication mechanisms [14]. All three versions use pthreads to implement the matching unit and the functional units. Following extensive experimentation, we observed that the three versions demonstrate almost identical behaviour and performances on typical use cases on the architectures used for the experiments, and we concluded that the performance figures obtained are due mainly to the model chosen rather than to the mechanisms used to implement the distributed interpreter.

The scheduling policy used to fetch fireable instructions from the pool may be programmed in the three cases. The results shown here all use a simple FIFO policy to deliver fireable instructions to functional units. A macro data flow program execution ends when no more fireable instructions exist and the policy chosen to implement the fireable instruction "queue" actually does not impact the completion time of the application, while it still impacts the service time in the case of streaming applications, that is, where the application is computed on a stream of input tasks. In streaming applications, a mechanism is implemented to guarantee the ordered delivery of results onto the output stream. Tokens with special "output" destination are routed to an output manager thread that in this case reorders results according to their graph id, in such a way that the input/output order is preserved.

3 Compiling to macro data flow graphs

The efficient implementation of the parallel macro data flow interpreter contributes to the efficient execution of the applications, but there still remains the problem of how to produce suitable macro data flow graphs from an application's high level specification.

Within the project, we chose to adopt two distinct and complementary approaches:

- *A structured parallel programming approach*, deriving instructions from application code written according to a parallel design pattern model. The application is expressed as a composition of parallel patterns chosen from a set of pre-defined patterns provided to the programmer by means of an algorithmic skeleton library. The application code is automatically compiled to a macro data flow graph in this case.

- *A reverse engineering approach*, deriving instructions from a high level imperative description of the application, consisting of control flow statements–typically loops–calling library functions such as those provided by classical BLAS library implementations.

Both approaches are discussed in the following Sections.

3.1 Parallel pattern based languages

Applications are expressed as compositions of well-known parallel programming patterns specialized by sequential portions of code specifying the application "business logic". This approach builds on the huge range of results from the algorithmic skeleton community and from the more recent parallel design pattern community [15].

We provide the programmer with a set of patterns, including classical patterns such as pipelines, expressing staged computations, farms, expressing embarrassingly parallel, master worker style computations, map and reduce, expressing data parallel computation. The pattern set is completely modular. Each pattern has parameters specifying the orchestrated computations. As an example, the pipeline pattern has parameters to specify the component stages. Such parameters may recursively be other patterns or wrappings of sequential code chunks.

In this case, a parallel computation may be expressed as a pattern nesting such as

$$pipe(seq(f), farm(seq(g)), seq(h))$$

representing a streaming computation (because of the outer pipeline pattern) with three stages computing a sequential code wrapping as the first stage (f), passing partial results to an embarrassingly parallel second stage with sequential workers ($farm(seq(g))$) which eventually delivers its results to a third stage computing the sequential code wrapping h.

Currently, such an application is represented by code that:

- declares three sequential wrapper objects transforming the code computing f, g, and h into nestable patterns;

- declares a farm object, using the g wrapper as worker code parameter;

- declares a pipeline object and adds the f pattern, this farm pattern and the h pattern as stages; and
- associates an input stream and an output stream to the pipeline skeleton, and calls a `compute` method on the pipeline skeleton to start computation via the parallel interpreter.

The overall result is similar to what happens in other skeleton frameworks [11, 16].

Algorithmic skeleton approaches, and those based on parallel design patterns, have often been criticized for their lack of flexibility in expressing complex or non-standard parallel computation patterns. If the parallel pattern for the application at hand cannot be expressed with a composition of the existing patterns, it usually cannot be implemented in the parallel structured programming framework. This is because the structured programming frameworks do not provide APIs to access the implementation detail of the patterns available to the application programmer. In turn, this is due to the desire to prevent the inexpert application programmer from impairing the efficient implementation and optimizations of the natively provided patterns. Due to our macro data flow based implementation, however, we can adopt a more flexible approach such as that discussed in [17]:

- We provide a suitable API to access internal macro data flow instructions and graph implementations.
- The application programmer may express the pattern he/she has in mind as a macro data flow graph, provided the graph has only a single input and a single output token, that is, there is a single instruction with one input token and no other instructions directing results to this token, and a single instruction with a single output token directed to the special "output" destination.
- Finally, suitable API calls exist to name this graph as a parallel pattern, in such a way that it can be used where any other, predefined pattern may be used.

Programmers needing a pattern not natively provided by the library may thus introduce new patterns into the system by providing their macro data flow compiled code. It is worth pointing out that such new patterns may have parameters modelling nested computations. In this case, the instructions directing tokens to the nested graphs may simply invoke API macros directing tokens to the input arcs of the nested pattern macro data flow graph or getting results from the output arcs of the nested pattern graph.

Although this is not a completely new technique (macro data flow implementation of skeleton frameworks was introduced in the late 90's [18] and adopted in different frameworks targeting COW/NOW architectures [19, 20], and expandability has already been proposed through macro data flow in [17]), to the best of our knowledge, this is the first attempt to migrate these techniques

```
FOR k = 0..TILES-1
  FOR n = 0..k-1
    A[k][k] := CHERK(A[k][n], A[k][k])
  A[k][k] := CPOTF2(A[k][k])
  FOR m = k+1..TILES-1
    FOR n = 0..k-1
      A[m][k] := CGEMM(A[k][n], A[m][n], A[m][k])
    A[m][k] := CTRSM(A[k][k], A[m][k])
```

Figure 2: Pseudo code for Left-looking block Cholesky factorization algorithm for complex matrix

```
seq vecMat in(float v[N], m[N][N])
           out(float r[N])
$C++{
   for(int i=0; i<N; i++)
     for(int k=0; k<N; k++)
       r[i]+=a[k]*b[k][i];
   }
}C++$
end seq

map in(float A[N][N], B[N][N])
    out(float C[N][N])
  vecMat in(A[*i][], B[][]) out(C[*i][])
end map
```

Figure 3: Skeleton code for MM

onto multi-cores. Incidentally, the adoption of a structured parallel programming approach based on patterns has been advocated as a means to program multi/many-core machines and to inform a new generation of parallel programmers in the well-know Berkeley report [21].

3.2 "Well-formed" numerical code

In a number of cases, programmers wish to implement in parallel numerical code whose numerical algorithm is known and expressed in pseudo code with loops and calls to library functions. We define *well-formed code* as code containing only loops and function calls. This notion captures a large number of significant numerical kernels used in a wide range of algorithms. Fig. 2 shows an example of such well-formed pseudo code for block Cholesky matrix factorization, hosting calls to standard BLAS and LAPACK functions.

To compile such code to macro data flow graphs we apply the following algorithm:

- We assume each BLAS function call is represented by a macro data flow instruction.

- Each macro data flow instruction has a number of input tokens equal to the number of arguments of the corresponding BLAS function call and a number of destinations equal to the number of results of the BLAS call (respectively one, two or three and one, in this simple case).

- we substitute the calls in the pseudo code with calls to the macro data flow instruction generation code API from our library, and we run the program in such a way that the macro data flow graph is generated.

The program generating the macro data flow graph looks like that in Fig. 4. Currently this code is hand written but we are implementing a compiler accepting as input pseudo code such as that in Fig. 2 and automatically generating the code needed to compile the pseudo code to macro data flow code.

Portions of macro data flow code derived using this automatic procedure may be wrapped in such a way that they become "standard" patterns exploiting the pattern set expandability procedure discussed in Sec. 3.1. As a result, our pattern based programming framework will eventually allow application programmers to use both standard parallel patterns (e.g. pipelines, farms and maps) and ad hoc patterns encapsulating well-known numerical kernels.

4 Experiments

To validate our approach we ran a number of experiments using synthetic applications and standard application benchmarks on various target architectures. In particular:

- The standard applications (kernels) used were in part derived from pattern-based, structured parallel code (e.g. matrix multiplication, with integer, float or complex elements, compiled from a stream parallel map pattern such as that shown in Fig. 3 in P3L syntax [22]) and in part compiled through our prototype macro data flow compiler processing well-formed pseudo code such as that in Fig. 2 (block Cholesky factorization).

- The compiled macro data flow code was run on various state-of-the-art multi-core architectures, including Intel and AMD multi-cores. In particular we had available two machines: a dual quad core with Intel Xeon E5520 Nehalem and a quad 12 core AMD Opteron 6174 Magny-Cours. Both platforms run the Linux x86_64 operating system and all software was compiled into 64-bit executables using the GNU C compiler with the -O3 compilation flag. For the remainder of the paper, we refer to these two architectures as Nehalem and Magny-Cours, respectively.

We report results on three different kinds of experiment aimed at i) validating the general impact of our data flow framework, ii) evaluating the efficiency of our framework when processing streams of data parallel tasks and iii) comparing the efficiency of our framework w.r.t. state-of-the-art data parallel frameworks on single item (i.e. non-stream parallel) data parallel computations.

```
for(k=0;k<=(tiles-1);k++) {
 for(n=0;n<=(k-1); n++) {
   // A[k][k] := CHERK(A[k][n], A[k][k])
   VAR * rv1 = newVar('a',k,n);
   VAR * rv2 = newVar('a',k,k);
   SET * rs = newSet(rv1);
   addToSet(rs,rv2);
   SET * ls = newSet( newVar('a',k,k) );
   ASSIGN * a =
     newAssign(instrNo++,ls,"cherk",rs);
   pgm = addToProgram(pgm, a);
   // END A[k][k] := ...
 }
 // A[k][k] := CPOTF2(A[k][k])
 SET * l = newSet(newVar('a',k,k));
 SET * r = newSet(newVar('a',k,k));
 ASSIGN * a =
    newAssign(instrNo++,l,"cpotf2",r);
 pgm = addToProgram(pgm, a);
 // END A[k][k] := ...
 for(m=(k+1); m<=(tiles-1); m++) {
   for(n=0; n<=(k-1); n++) {
     // A[m][k] := CGEMM(A[k][n], A[m][n], A[m][k])
     SET * l = newSet(newVar('a',m,k));
     SET * r = newSet(newVar('a',k,n));
     addToSet(r, newVar('a',m,n));
     addToSet(r, newVar('a',m,k));
     ASSIGN * a =
       newAssign(instrNo++,l,"cgemm",r);
     pgm = addToProgram(pgm, a);
     // END A[m][k] := ...
   }
   // A[m][k] := CTRSM(A[k][k], A[m][k])
   SET * ll = newSet(newVar('a',m,k));
   SET * rr = newSet(newVar('a',k,k));
   addToSet(rr,newVar('a',m,k));
   ASSIGN * aa =
     newAssign(instrNo++,ll,"ctrsm",rr);
   pgm = addToProgram(pgm, aa);
   // END  A[m][k] := ...
 }
}
```

Figure 4: Code generating macro data flow graph from block Cholesky pseudo code of Fig. 2

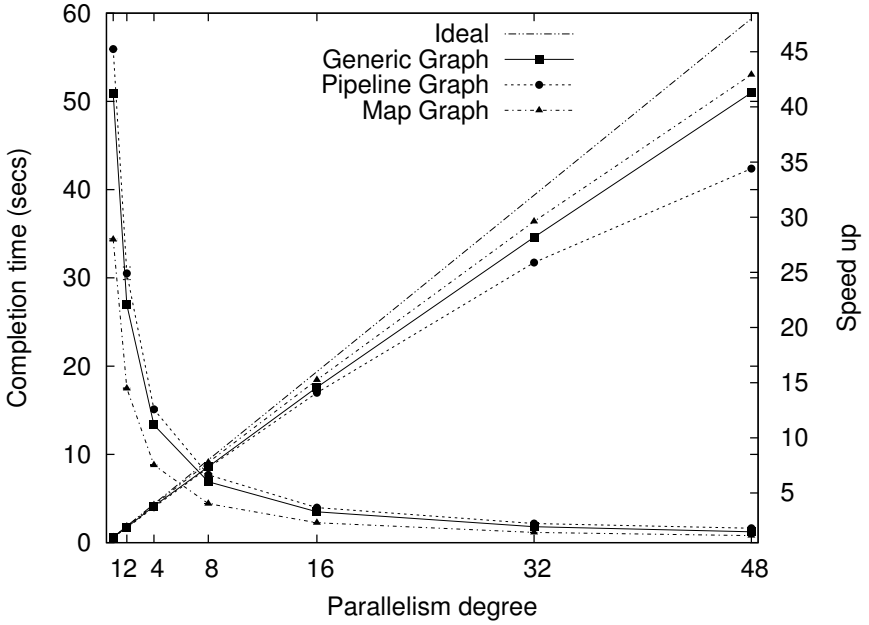

Figure 5: Scalability of synthetic applications: comparison between results achieved by typical macro data flow graphs (on Magny-Cours, Unix pipe based interpreter).

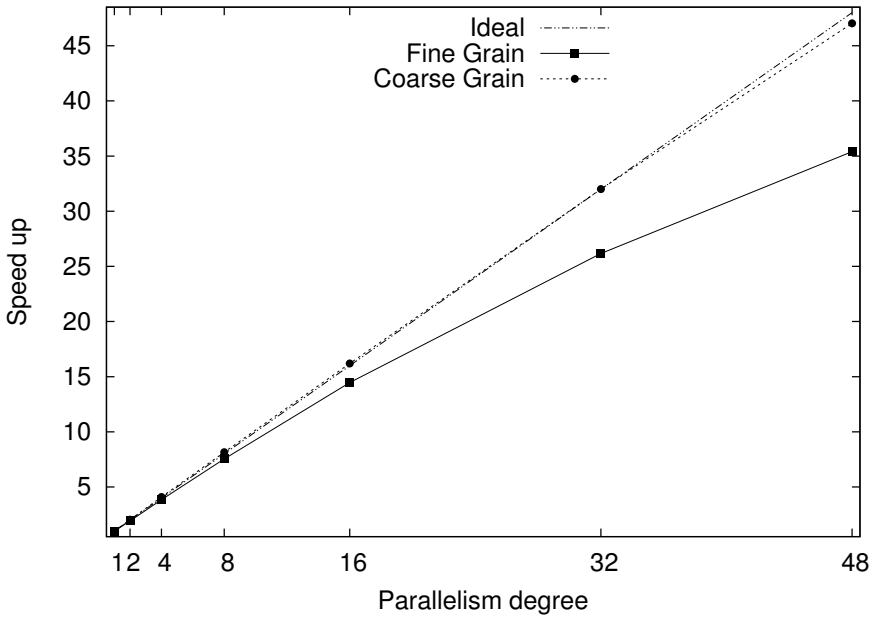

Figure 6: Effect of macro data flow instruction average computation weight: in the "coarse grain" application the average weight is ten times that of the fine grain application (on Magny-Cours, Unix pipe based interpreter).

Validating the general impact Scalability of our macro data flow interpreter was measured using synthetic applications and the results obtained are outlined in Figs. 5 and 6. The synthetic applications in this case were compiled from i) a generic macro data flow graph application hosting instructions with varying numbers of input tokens compiled from well-formed pseudo code, ii) a skeleton application using a pipeline with a large number of stages and iii) a skeleton application using a map skeleton (such as that in Fig. 3). The time spent computing the single macro data flow instructions in the three cases is in the range of milliseconds and the applications process streams of input data to produce streams of output results with the input streams having hundreds of items. Fig. 5 shows the different performances resulting from the different kinds of graphs used. The pipeline compilation compiles to a linear chain of macro data flow instructions. The map compilation produces a graph with a single split and a single merge macro data flow instruction plus a large number of macro data flow instructions fired by the execution of the split instruction and directing results to the merge instruction. The generic graph application, instead, compiles to more irregular macro data flow instruction sub-graphs. Fig. 6 demonstrates how different weights in the macro data flow instruction computations determine efficiency.

We also compared performance figures obtained with those achieved when executing the same benchmarks with standard multi-core programming techniques. We compared the performance of matrix multiplication (naive algorithm) using our framework with that achieved using OpenMP. Fig. 7 shows the results. Our map pattern implementation in macro data flow performs better than the OpenMP implementation of the same algorithm (parallelization on the external `for i` loop of the three used to implement the naive matrix multiplication algorithm, with no particular optimizations/pragma clauses) on the Nehalem dual quad core. With careful use of the `parallel for` options–e.g. those affecting scheduling–and with larger core numbers (NUMA architectural characterization gets stronger, in this case), OpenMP performance is closer to that of the macro data flow structured framework and actually outperforms our framework when only a few of the available cores are used (see Fig. 8). It is worth pointing out, however, that we are comparing code automatically derived from parallel patterns with hand-optimized OpenMP code.

Efficiency when processing streams of data parallel tasks Ideally, when a stream of many tasks has to be computed and both the service time and the total completion time need to be optimized, the best solution is to use a farm paradigm. On the other hand, when the input stream length is small or very small (less than the available parallelism) the farm paradigm is not always able to produce the best performance. In this latter case, we need to parallelize also the single task of the stream hence producing a mixed stream and data-parallel computation. We ran experiments aimed at determining if our macro data flow framework is able to approach farm performance when the stream length is long enough, and if it is able to produce improved performance when the input

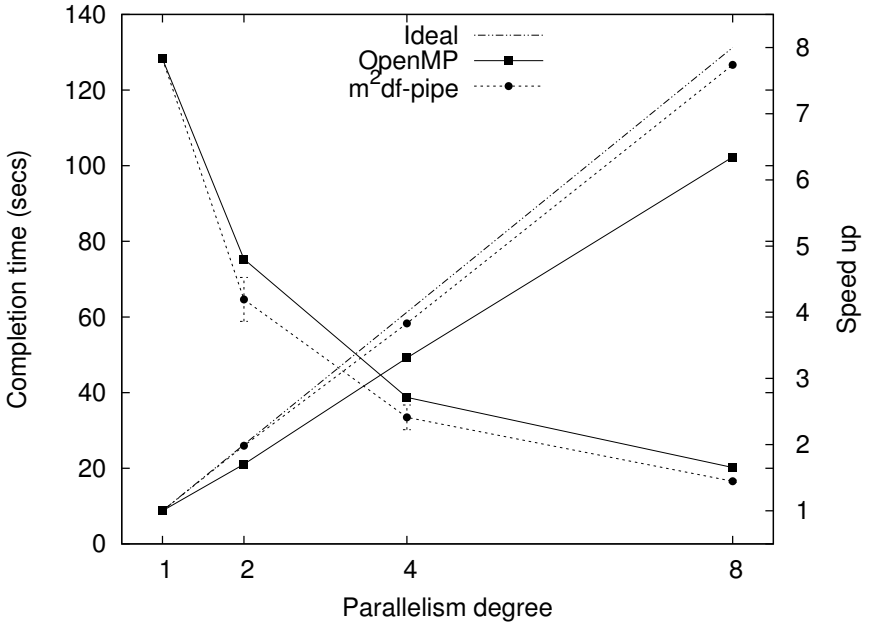

Figure 7: Comparing with OpenMP (MM, Intel Nehalem)

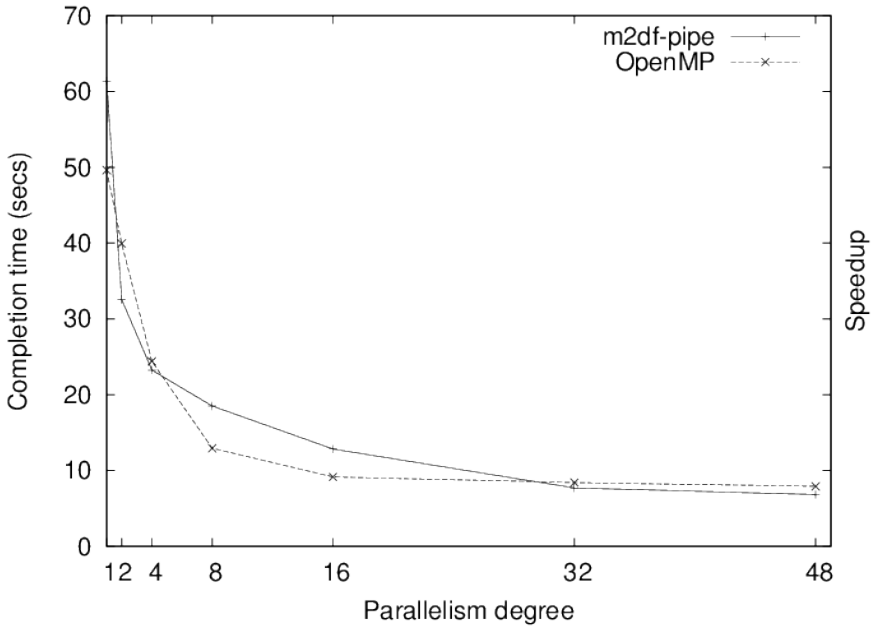

Figure 8: Comparing with OpenMP (MM, AMD Magny-Cours)

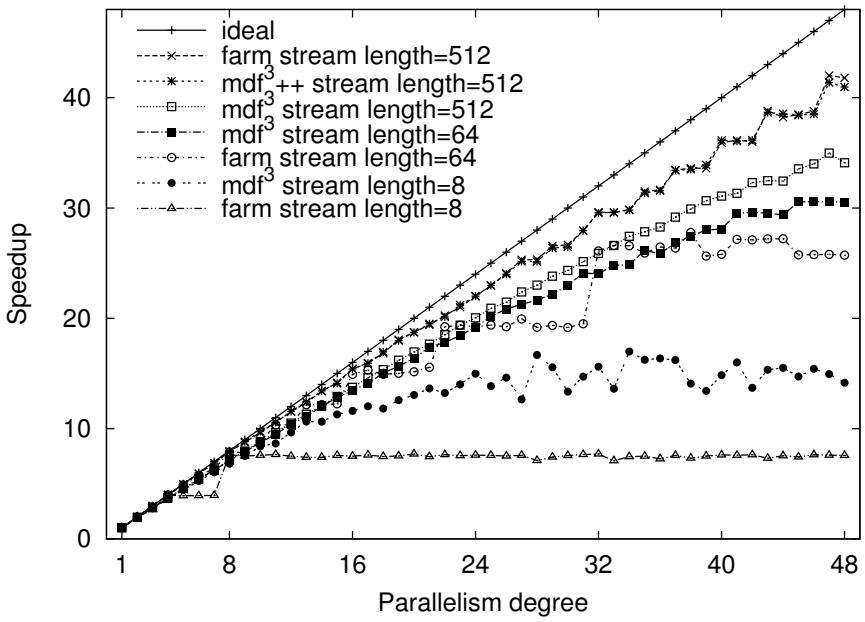

Figure 9: Comparing farm and macro data flow paradigms for different stream lengths (AMD Magny-Cours).

stream length is small. For this purpose, we tested the block Cholesky algorithm on a stream of relatively small complex input matrices of size 1024x1024 using 3 different stream lengths: 512, 64 and 8 items. These experiments were run using the mdf^3 implementation of the interpreter build on top of FastFlow and a "pure FastFlow" stream parallel implementation using farm with sequential workers. The block size was tuned and eventually set to 64x64, thus resulting in 816 data flow instructions per matrix (16 CPOTF2, 120 CTRSM, 560 CGEMM and 120 CHERK) with a computational grain which spans the range 150–220 microseconds per instruction. We used Intel's MKL 10.3 for the computation of the instruction on the single block.

Figure 9 shows the speedup obtained. The farm implementation outperforms the mdf^3 version for a stream of 512 tasks. This is due mainly to the inferior cache exploitation by the mdf^3, which schedules single instructions operating on small blocks toward the pool of executors without taking into account–for the time being, anyway–any cache affinity. In contrast, the mdf^3 implementation outperforms the farm paradigm for smaller stream lengths where the greater number of tasks produced by the macro data flow version is able to feed all executors for more time.

However, a simple optimization may be introduced that makes the macro data flow framework also competitive on long streams. It has already been demonstrated that structured parallel programs may be automatically trans-

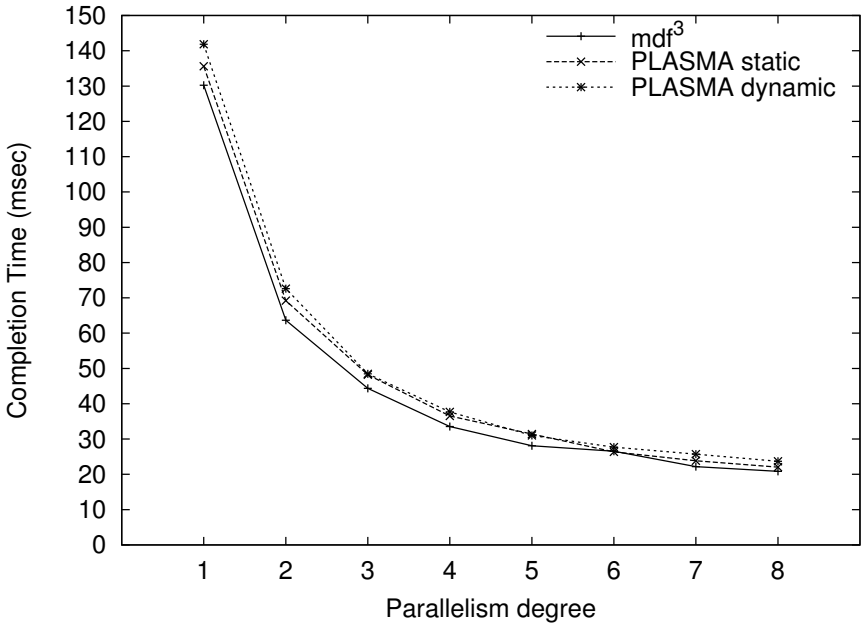

Figure 10: mdf^3 vs PLASMA library. Cholesky factorization for a single 1024x1024 complex matrix (Intel Nehalem).

formed into "normal form" presenting better performance and efficiency w.r.t. the original program [23]. By applying a similar concept, we have introduced the possibility to "group" entire macro data flow subgraphs into a (logically) single instruction. By applying this optimization to the computation of each single matrix in the stream, we obtained the speedup labelled as mdf^3++ in Fig. 9, which is basically the same speedup as the pure farm implementation (the differences are almost indistinguishable). This proves that the macro data flow framework is able to obtain similar or better performance than the farm paradigm for any length of input stream.

Efficiency when processing data parallel tasks Finally, to validate the implementation of the data flow interpreter when fine-grained data-parallel computations are considered, we tested the mdf^3 implementation of the block Cholesky algorithm operating on a single complex input matrix against the PLASMA 2.3.1 library (Parallel Linear algebra for Scalable Multi-core Architectures) [24] version of the same algorithm. The PLASMA library, developed at University of Tennessee, was specifically designed and optimized to target shared cache multi-core platforms.

For the PLASMA version we tested two different scheduling policies: the static pipeline scheduling policy (PLASMA_STATIC_SCHEDULING) and the fully dynamic scheduling policy (PLASMA_DYNAMIC_SCHEDULING). In the

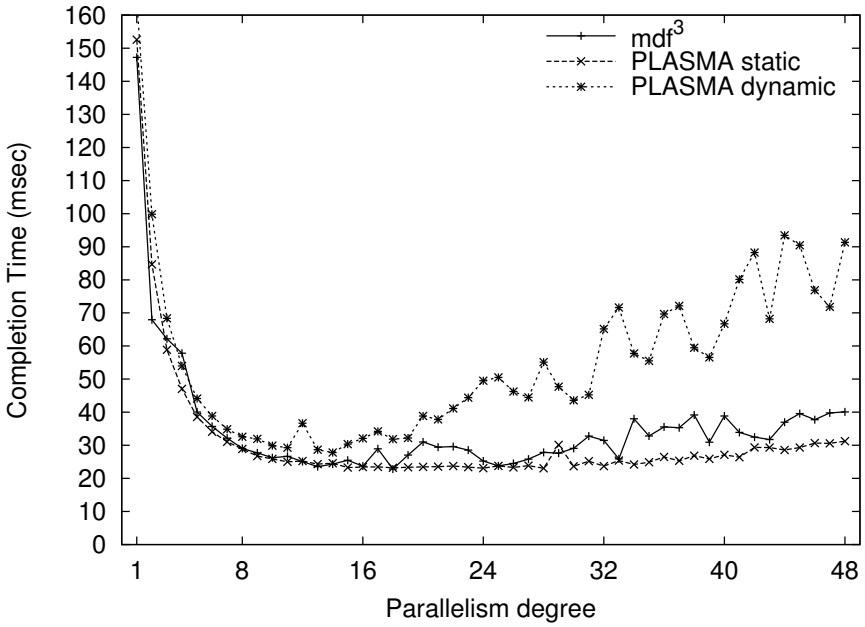

Figure 11: mdf^3 vs PLASMA library. Cholesky factorization for a single 1024x1024 complex matrix (AMD Magny-Cours).

static pipeline version the work is partitioned in one dimension by block-rows and jobs are assigned across all steps of the factorization in a pipeline fashion. In order to satisfy dependencies a global progress table is maintained, with threads performing busy-waiting if the data dependencies on a given block are not satisfied [25]. The dynamic version implements fully dependency-driven/data-driven dynamic scheduling as in the macro data flow version. Single tasks, corresponding to one LAPACK or BLAS operation on a single block, are scheduled as their dependencies become satisfied and subsequently input data becomes available [26].

In Fig. 10 and Fig. 11 are sketched the completion times obtained on the Nehalem and Magny-Cours platforms, respectively. As can be seen the mdf^3 implementation is comparable with (and on the Nehalem platform better than) the highly optimized static pipeline version of the PLASMA library. PLASMA's dynamic scheduling version is always slower than mdf^3 and this is probably due to the higher scheduling overhead. As expected, the PLASMA static pipeline version is much more stable when the number of cores increases. In fact, starting from a parallelism degree of 16, there is no performance gain due to lack of available parallelism for the matrix size tested, and thus the greater overhead of the dynamic scheduling policies of the mdf^3 and of the PLASMA dynamic versions result in performance degradation.

5 Related work

As we propose to combine macro data flow with structured parallel programming to provide a programming framework targeting multi-core architectures, we address several different areas when considering related work.

A number of programming environments use data flow concepts in the implementation of different parallel programming models. For example, different systems provide the programmer with the possibility to define/identify *tasks* that are subsequently scheduled for parallel execution according to different execution models and scheduling policies. Cilk provides the application programmer with the possibility to *spawn* computations (C function/procedure calls) and to wait asynchronously for their termination [27]. The execution model for tasks is based on the scheduling of the resulting DAG representing tasks and task dependencies. The supporting run time library uses job-stealing to guarantee efficiency and appears similar to the macro data flow interpreter we have implemented. However, application programmer responsibilities are much heavier than those resulting from the use of high level patterns, as the programmer must be completely aware of the management of the dependencies of the tasks through suitable coding of control flow in the annotated C program representing the Cilk source. OpenMP tasks [28] and some recent extensions of the OpenMP task model [10] also provide tasks, identified in C or Fortran code through annotations, which are scheduled for execution in a fairly similar manner to that used for our fireable macro data flow instructions. StarPU [29] supports task graphs in essence similar to macro data flow graphs and executes these graphs

on heterogeneous architectures (multi-core + GPU) with interesting results. However the task of identifying tasks in the code, defining dependencies and managing tasks is explicitly and completely the responsibility of the application programmer. X10 [30] provides more sophisticated concurrency control statements than those strictly needed to set up a task parallel computation, but again requires quite a deep understanding of concurrency/parallelism on the part of the application programmer.

As far as structured programming models are concerned, there are several parallel programming environments that provide a framework similar to ours. Among them notable systems are Muesli [11], written in C++ and running on top of MPI with the possibility to exploit OpenMP, Skandium [31] written in Java and targeting multi-core architectures, and SkeTo [32], written in C and targeting the MPI virtual machine, supporting only data parallel computations. However, none of these frameworks provides the possibility to extend the skeleton/pattern set supported, nor do they provide any API to the internal implementation engine. In fact, they are all based on implementation template technology [33] rather than on macro data flow. This notwithstanding, the programming framework exposed to the application programmer is very close to that which we provide.

6 Conclusions

We discussed a programming framework providing the application programmer with the possibility to use either pre-defined parallel patterns or application or domain specific new patterns programmed as parametric macro data flow graphs. The high level programming abstractions are compiled to macro data flow and the macro data flow code is eventually executed on multi-cores through a parallel interpreter. The approach combines results from structured parallel programming and multi-core programming and reuses in part experiences from the community investigating task parallelism issues on multi-cores. Experimental results on state-of-the-art multi-core architectures equipped with different versions of parallel macro data flow interpreters demonstrate the feasibility and the efficiency of the approach.

We are currently investigating several improvements and optimizations for our framework. We are looking at the possibility of removing the logical matching unit thread bottleneck. In particular, we are evaluating a hierarchical, parallel implementation of the macro data flow repository with an associated matching unit implemented in FastFlow. Preliminary results demonstrate the feasibility of this approach along with the possibility of implementing different policies to ensure load balancing through macro data flow instruction stealing.

We are also looking at the extensive data flow literature to see whether data flow graph optimization and rewriting results may be reused in our framework to improve grain and locality in the graphs generated from the high level programming abstractions presented to the application programmer.

References

[1] G. Kahn, "The semantics of a simple language for parallel programming," in *Information processing*, J. L. Rosenfeld, Ed. Stockholm, Sweden: North Holland, Amsterdam, Aug 1974, pp. 471–475.

[2] J. B. Dennis and D. P. Misunas, "A preliminary architecture for a basic data-flow processor," in *Proceedings of the 2nd annual symposium on Computer architecture*, ser. ISCA '75. New York, NY, USA: ACM, 1975, pp. 126–132. [Online]. Available: http://doi.acm.org/10.1145/642089.642111

[3] E. Lee and T. Parks, "Dataflow process networks," *Proc. of the IEEE*, vol. 83, no. 5, pp. 773–801, May 1995.

[4] W. M. Johnston, J. R. P. Hanna, and R. J. Millar, "Advances in dataflow programming languages," *ACM Comput. Surv.*, vol. 36, pp. 1–34, March 2004. [Online]. Available: http://doi.acm.org/10.1145/1013208.1013209

[5] I. Watson and J. Gurd, "A practical data flow computer," *Computer*, vol. 15, pp. 51–57, February 1982. [Online]. Available: http://portal.acm.org/citation.cfm?id=1318726.1319194

[6] Y. Yamaguchi, S. Sakai, K. Hiraki, and Y. Kodama, "An architectural design of a highly parallel dataflow machine," in *IFIP Congress*, 1989, pp. 1155–1160.

[7] K. Arvind and R. S. Nikhil, "Executing a program on the mit tagged-token dataflow architecture," *IEEE Trans. Comput.*, vol. 39, pp. 300–318, March 1990. [Online]. Available: http://dx.doi.org/10.1109/12.48862

[8] Arvind and D. E. Culler, *Dataflow architectures*. Palo Alto, CA, USA: Annual Reviews Inc., 1986, pp. 225–253. [Online]. Available: http://portal.acm.org/citation.cfm?id=17814.17824

[9] R. Newton, F. Schlimbach, M. Hampton, and K. Knobe, "Capturing and composing parallel patterns with Intel CnC," in *Proc. of USENIX Workshop on Hot Topics in Parallelism (HotPar 2010)*, Berkley, CA, USA, Jun. 2010.

[10] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta, "Hierarchical task-based programming with starss," *IJHPCA*, vol. 23, no. 3, pp. 284–299, 2009.

[11] P. Ciechanowicz and H. Kuchen, "Enhancing muesli's data parallel skeletons for multi-core computer architectures," in *HPCC*. IEEE, 2010, pp. 108–113.

[12] Y. Sun and Z. Xie, "Macro-dataflow computational model and its simulation," *Journal of Computer Science and Technology*, vol. 5, pp. 289–295, 1990, 10.1007/BF02945317. [Online]. Available: http://dx.doi.org/10.1007/BF02945317

[13] M. Aldinucci, M. Danelutto, and P. Teti, "An advanced environment supporting structured parallel programming in Java," *Future Generation Computer Systems*, vol. 19, no. 5, pp. 611–626, Jul. 2003. [Online]. Available: http://www.di.unipi.it/~aldinuc/paper_files/2003_lithium_fgcs.pdf

[14] M. Aldinucci, M. Danelutto, M. Meneghin, P. Kilpatrick, and M. Torquati, "Efficient streaming applications on multi-core with FastFlow: the biosequence alignment test-bed," in *Parallel Computing: From Multicores and GPU's to Petascale (Proc. of PARCO 2009, Lyon, France)*, ser. Advances in Parallel Computing, vol. 19. Lyon, France: IOS press, Sep. 2009, pp. 273–280. [Online]. Available: http://www.di.unipi.it/~aldinuc/paper_files/2009_fastflow_parco.pdf

[15] T. Mattson, B. Sanders, and B. Massingill, *Patterns for parallel programming*, 1st ed. Addison-Wesley Professional, 2004.

[16] M. Aldinucci, M. Meneghin, and M. Torquati, "Efficient smith-waterman on multi-core with fastflow," in *Proc. of Intl. Euromicro PDP 2010: Parallel Distributed and network-based Processing*, M. Danelutto, T. Gross, and J. Bourgeois, Eds. Pisa, Italy: IEEE, Feb. 2010. [Online]. Available: http://www.di.unipi.it/~aldinuc/paper_files/2010_fastflow_SW_PDP.pdf

[17] M. Aldinucci, M. Danelutto, and P. Dazzi, "Muskel: an expandable skeleton environment," *Scalable Computing: Practice and Experience*, vol. 8, no. 4, pp. 325–341, Dec. 2007. [Online]. Available: http://www.scpe.org/vols/vol08/no4/SCPE_8_4_01.pdf

[18] M. Danelutto, "Efficient support for skeletons on workstation clusters," *Parallel Processing Letters*, vol. 11, no. 1, pp. 41–56, 2001. [Online]. Available: http://www.di.unipi.it/~marcod

[19] J. Sérot, D. Ginhac, and J. Dérutin, "SKiPPER: A Skeleton-Based Parallel Programming Environment for Real-Time Image Processing Applications," in *PaCT'99 — International Parallel Computing Technologies Conference*, St-Petersburg, September 6–10, 1999. [Online]. Available: http://wwwlasmea.univ-bpclermont.fr/Personnel/Jocelyn.Serot/papers/articles/pact99.ps.gz

[20] M. Danelutto, "QoS in parallel programming through application managers," in *Proc. of Intl. Euromicro PDP: Parallel Distributed and network-based Processing*. Lugano, Switzerland: IEEE, Feb. 2005, pp. 282–289. [Online]. Available: http://www.di.unipi.it/~marcod

[21] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: A view from berkeley," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, Dec 2006. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html

[22] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi, "P3l: A structured high level programming language and its structured support," *Concurrency Practice and Experience*, vol. 7, no. 3, pp. 225–255, May 1995. [Online]. Available: http://www3.interscience.wiley.com/journal/113441483/abstract

[23] M. Aldinucci and M. Danelutto, "Stream parallel skeleton optimization," in *Proc. of PDCS: Intl. Conference on Parallel and Distributed Computing and Systems*, IASTED. Cambridge, Massachusetts, USA: ACTA press, Nov. 1999, pp. 955–962. [Online]. Available: http://www.di.unipi.it/~aldinuc/paper_files/1999_NF_pdcs.pdf

[24] *PLASMA library website*, 2011, http://icl.cs.utk.edu/plasma/.

[25] J. Kurzak, H. Ltaief, J. Dongarra, and R. M. Badia, "Scheduling dense linear algebra operations on multicore processors," *Concurrency and Computation: Practice and Experience*, vol. 22, pp. 15–44, 2010.

[26] J. D. Azzam Haidar Hatem Ltaief, Asim YarKhan, "Analysis of dynamically scheduled tile algorithms for dense linear algebra on multicore architectures," in *First International Workshop on Parallel Software Tools and Tool Infrastructures (PSTI 2010), San Diego*, April, 2010.

[27] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *J. Parallel Distrib. Comput.*, vol. 37, no. 1, pp. 55–69, 1996.

[28] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang, "The design of openmp tasks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 20, pp. 404–418, March 2009. [Online]. Available: http://portal.acm.org/citation.cfm?id=1512157.1512430

[29] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, vol. 23, pp. 187–198, Feb. 2011. [Online]. Available: http://hal.inria.fr/inria-00550877

[30] V. A. Saraswat, "X10: Concurrent programming for modern architectures," in *APLAS*, ser. Lecture Notes in Computer Science, Z. Shao, Ed., vol. 4807. Springer, 2007, p. 1.

[31] M. Leyton and J. M. Piquer, "Skandium: Multi-core programming with algorithmic skeletons," in *PDP*, M. Danelutto, J. Bourgeois, and T. Gross, Eds. IEEE Computer Society, 2010, pp. 289–296.

[32] H. Tanno and H. Iwasaki, "Parallel skeletons for variable-length lists in sketo skeleton library," in *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, ser. Euro-Par '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 666–677. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03869-3_63

[33] S. Pelagatti, *Structured development of parallel programs.* Taylor and Francis, 1999.