Parallel Computing 40 (2014) 661–680

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Couillard: Parallel programming via coarse-grained Data-flow Compilation

Leandro A.J. Marzulo[a,1], Tiago A.O. Alves[b,*], Felipe M.G. França[b,2], Vítor Santos Costa[c,3]

[a] *Departamento de Informática e Ciência da Computação, Instituto de Matemática e Estatística, IME, Universidade do Estado do Rio de Janeiro, Rua São Francisco Xavier, 524, Pavilhão Reitor João Lyra Filho, 6o Andar, Diretoria, Sala 6019, Bloco B, Rio de Janeiro, Rj 20550-900, Brazil*

[b] *Programa de Engenharia de Sistemas e Computação, COPPE, Universidade Federal do Rio de Janeiro, Cidade Universitária, Centro de Tecnologia, Bloco H, Sala 319, Rio de Janeiro, RJ 21941-972, Brazil*

[c] *CRACS & INESC-Porto LA, Faculdade de Ciências, Universidade do Porto, Rua do Campo Alegre, 1021, 4169-007 Porto, Portugal*

A R T I C L E   I N F O

Article history:
Received 24 November 2011
Received in revised form 5 September 2013
Accepted 9 October 2014
Available online 24 October 2014

Keywords:
Parallel programming
Data-flow
Language
Compiler
Run-time environment

A B S T R A C T

Data-flow is a natural approach to parallelism. However, describing dependencies and control between fine-grained data-flow tasks can be complex and present unwanted overheads. TALM (TALM is an Architecture and Language for Multi-threading) introduces a user-defined coarse-grained parallel data-flow model, where programmers identify code blocks, called super-instructions, to be run in parallel and connect them in a data-flow graph. TALM has been implemented as a hybrid Von Neumann/data-flow execution system: the *Trebuchet*. We have observed that TALM's usefulness largely depends on how programmers specify and connect super-instructions. Thus, we present *Couillard*, a full compiler that creates, based on an annotated C-program, a data-flow graph and C-code corresponding to each super-instruction. We show that our toolchain allows one to benefit from data-flow execution and explore sophisticated parallel programming techniques, with small effort. To evaluate our system we have executed a set of real applications on a large multi-core machine. Comparison with popular parallel programming methods shows competitive speedups, while providing an easier parallel programing approach. More specifically, for an application that follows the wavefront method, running with big inputs, *Trebuchet* achieved up to 4.7% speedup over Intel® TBB novel flow-graph approach and up to 44% over OpenMP.

1. Introduction

Data-flow programming provides a natural approach to parallelism, where instructions execute as soon as their input operands are available [1–4]. Actually in dynamic data-flow, we may even have independent instructions from multiple iterations on a loop running simultaneously, as parts of the loop may run faster than others and reach next iterations. Therefore

* Corresponding author. Fax: +55 21 2562 8676.
E-mail addresses: leandro@ime.uerj.br (L.A.J. Marzulo), tiagoaoa@cos.ufrj.br (T.A.O. Alves), felipe@cos.ufrj.br (F.M.G. França), vsc@dcc.fc.up.pt (V.S. Costa).
[1] Principal corresponding author. Fax: +55 21 2334 0144.
[2] Fax: +55 21 2562 8676.
[3] Fax: +351 220 402 950.

http://dx.doi.org/10.1016/j.parco.2014.10.002
0167-8191/© 2014 Elsevier B.V. All rights reserved.

it is complex to describe control in data-flow, since instructions must only proceed to execution when operands from the same iteration match. However, this difficulty is compensated by the amount of parallelism exploited.

TALM (TALM is an Architecture and Language for Multi-threading) [5–7] is an execution model designed to exploit the advantages of data-flow in multithread programming. A program in TALM is comprised of code blocks called *super-instructions* and simple instructions connected in a graph according to their data dependencies (i.e. a data-flow graph). To parallelize a program using the TALM model, the programmer marks portions of code that are to become super-instructions and describe their dependencies. With this approach, parallelism comes naturally from data-flow execution.

The major advantage of TALM is that it provides a coarse-grained parallel model that can take advantage of data-flow. It is also a very flexible model, as the main data-flow instructions are available, thus allowing full compilation of control in a data-flow fashion. This gives the programmer the latitude to choose from coarser to more fine-grained execution strategies. This approach contrasts with previous work in data-flow programming [1,2,4], which often aim at hiding data-flow execution from the programmer.

A first implementation of TALM, the *Trebuchet* system, has been developed as a hybrid Von Neumann/data-flow execution system for thread-based architectures in shared memory platforms. *Trebuchet* emulates a data-flow machine that supports both simple instructions and super-instructions. Super-instructions are compiled as separate functions that are called by the runtime environment, while regular instructions are interpreted upon execution. Although *Trebuchet* needs to emulate data-flow instructions, experience showed most running time is within our super-instructions. Initial results show the parallel engine to be competitive with state-of-the-art parallel applications using OpenMP, both in terms of base performance, and in terms of speedups [6,5,7]. On the other hand, parallelism for simple SPMD (Single-Program Multiple-Data) applications can be explored quite well with tools such as OpenMP. The main benefits exploited by TALM become apparent when experimenting with applications that require more complex techniques, such as software pipelining, wavefront [8], or speculative execution.

The usefulness of TALM clearly depends on how the programmer can specify and connect super-instructions together, including the complex task of describing control using data-flow instructions. We therefore introduce *THLL* (TALM High-Level Language), an extension of C-language that allows programmers do define and connect super-instruction definitions. We also present *Couillard*, a `C`-compiler designed to compile *THLL* applications into a data-flow graph, including the description of program control using dynamic data-flow. *Couillard* is designed to insulate the programmer from the details of data-flow programming. By requiring the programmer to just annotate the code with the super-instruction definitions and their dependencies, *Couillard* greatly simplifies the task of parallelizing applications with TALM.

This work makes the following contributions:

- Definition of THLL (TALM High-Level Language), as an extension of ANSI C.
- Implementation of the *Couillard* Compiler, which generates data-flow graphs and super-instruction code for TALM.
- Extension of TALM Assembler with macros that allow placement definition along with the assembly code.
- Addition of naive placement generation on Couillard, which is enough for most regular applications.
- Creation of a placement algorithm based on list scheduling, to be used on more complex applications.
- Addition of work-stealing mechanism to the *Trebuchet* runtime environment, based on the ABP algorithm [9]. The proposed mechanism only steals super-instructions, since simple instructions perform a very small amount of work and stealing them would not be profitable. Moreover, programmer can mark (in their code) which super-instructions can be stolen.
- Performance evaluation of *Couillard*/*Trebuchet* using four applications: a ray-tracer, a DNA global sequence alignment application that follows the Needleman–Wunsch algorithm, Blackscholes and Ferret (the last two from the PARSEC benchmark suite [10]).

We demonstrate that *Trebuchet* and *Couillard* allows one to explore complex parallel programing techniques, such as non-linear software pipelines and hiding I/O latency. Comparison with popular parallel programming models, such as Pthreads [11], OpenMP [12] and Intel Thread Building Blocks [13] shows that our approach is not just competitive with state-of-the-art technology, but that in fact can achieve better speedups by allowing one to easily exploit a sophisticated design space for parallel programs.

The paper is organized as follows. In Section 2 we briefly review TALM architecture and its implementation, the *Trebuchet*. In Section 3 we describe THLL (TALM High-Level Language) and Couillard Compiler implementation. In Section 4 we discuss our placement mechanism and work-stealing technique. In Section 5 we present performance results on the two PARSEC benchmarks. In Section 6 we discuss some related works. Last, in Section 7 we present our conclusions and discuss future work.

2. TALM and Trebuchet

TALM [5–7] allows application developers to take advantage of the possibilities available in the data-flow model in current Von Neumann architectures, in order to explore TLP in a more flexible way. TALM Instruction Set Architecture sees applications in the form of a data-flow graph, where independent instructions can be run in parallel if there are available computation resources.

A main contribution of TALM is that it enables programmers to introduce user-defined instructions, the so called *super-instructions*. TALM assumes a contract with the programmer whether she or he guarantees that execution of the super-instruction can start if all inputs are available, and where she or he guarantees to make output arguments available as soon as possible, but not sooner. Otherwise, TALM has no information on the semantics of individual super-instructions, and indeed imposes no restrictions. Thus, a programmer can use shared memory in super-instructions without having to inform TALM. Although this requires extra care from the programmer, the advantage is that TALM allows easy porting of imperative programs and easily allows program refinement.

TALM has been implemented for multi-cores as a hybrid Von Neumann/Data-flow execution system: the *Trebuchet* [5–7]. *Trebuchet* is in fact a data-flow virtual machine that has a set of data-flow processing elements (PEs) connected in a virtual network. Each PE is associated with a thread at the host (Von Neumann) machine. When a program is executed on *Trebuchet*, instructions are loaded into the individual PEs and fired according to the Data-flow model. Independent instructions will run in parallel if they are mapped to different PEs and there are available cores at the host machine to run those PEs' threads simultaneously.

Trebuchet is a Posix-threads based implementation of TALM. It loads super-instructions as a dynamically linked library. At run-time, execution of super-instructions is fired by the virtual machine, according to the data-flow model, but their interpretation comes down to a function call resulting in the direct execution of the related block.

Trebuchet may either rely solely on static scheduling of instructions among PEs or may also use work-stealing as a tool against imbalance. The work-stealing algorithm employed by *Trebuchet* is based on the ABP algorithm [9], the main difference being that the algorithm developed for *Trebuchet* provides a FIFO double-ended queue (deque) instead of a LIFO one, as is the case for the ABP algorithm. The FIFO order is chosen so that older instructions have execution priority, which is desirable for the applications we target at this moment. This work-stealing mechanism is discussed in more details on Section 4.

Fig. 1 shows the work-flow that describes the necessary steps to parallelize a sequential program and execute on *Trebuchet*. The steps are the following:

1. **Blocks Definition**: From the original (sequential) C code, programmers define blocks that will form super-instructions. Profiling tools may be used in helping to determine which portions of code are interesting candidates for parallelization and to define those instruction blocks. Super-instructions are marked using THLL annotations. THLL (TALM High-Level language) is an extension of the C-Language introduced in this paper (Section 3).

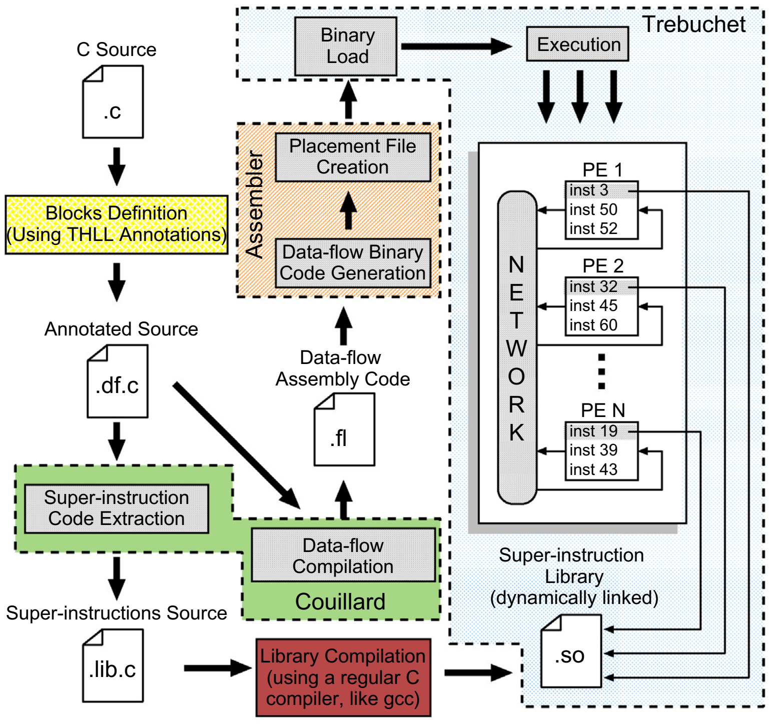


Fig. 1. Work-flow to follow when writing parallel applications with *Trebuchet*. Original (sequential) code is annotated with THLL and compiled with Couillard to generate an assembly representation of the application data-flow graph and super-instruction C-code. The graph is assembled, placement is defined and super-instruction code is compiled with a regular C-compiler (such as gcc) as a dynamically linked library. The graph binary, placement file, and dynamic library are passed to *Trebuchet* Runtime and execution follows the data-flow firing rule. Simple instructions must be interpreted while super-instructions execution is resumed to a simple function call to the shared library.

2. **Super-Instruction Code Extraction**: According to program annotations, each super-instruction is transformed into a function that will collect input operands from *Trebuchet*, execute, and return output operands. This is automatically done by the Couillard Compiler (explained at Section 3).
3. **Data-flow Compilation**: The annotated code is compiled with Couillard to generate an assembly code representation of the data-flow graph that connects all instructions. The assembly code may have both super-instructions and simple (fine-grained) instructions. TALM provides all the standard data and control instructions that one would expect in a dynamic data-flow machine.
4. **Data-flow Binary Code Generation**: TALM Assembler converts assembly code into a binary representation that can be read by Trebuchet Runtime.
5. **Placement File Creation**: Instruction placement has a big impact on performance. It determines which instructions are assigned to each processing element and, therefore, which instructions will be allowed to run in parallel. TALM assembly language was improved with the inclusion of macros to allow manual definition of instruction placement. Couillard generates assembly code using those macros to specify a naive placement (described in Section 4) that works for most regular applications. For applications with more complex parallelization patterns, programmers can rely on our placement algorithm (also described in Section 4) or, in the worse scenario, define their own customized placement file.
6. **Library Compilation**: The super-instruction function file is compiled, using a regular C-compiler (such as gcc) into a dynamic library, which will be available to the abstract machine interpreter.
7. **Binary Load**: The Trebuchet Runtime loads all instructions and super-instructions from the binary code into its processing elements, according to the placement file.
8. **Execution**: Program execution on *Trebuchet* is guided by the data-flow firing rule, i.e., instructions are executed as soon as their input operands become available. Execution of simple instructions requires full interpretation, whereas super-instructions are directly executed on the host machine, by simply calling the super-instruction function in the shared library.

Notice that in previous work [5–7], Blocks Definition, Super-Instruction Code Extraction, Data-flow Compilation, and instruction placement had to be manually performed, since THLL, Couillard and our Placement mechanism were not yet introduced. Those tools are essential to improve programming productivity and avoid coding errors.

3. THLL and Couillard

The data-flow model exposes ILP (instruction-level parallelism) by taking advantage of how data is exchanged between processing elements. In this vein, programming in TALM is about identifying parallel tasks and how data is consumed and produced by them. Since in *Trebuchet*, virtual PEs are mapped to threads at the host (multicore) machine, ILP is transformed into TLP (Thread-Level Parallelism). The initial *Trebuchet* implementation provided an execution environment for multicores, plus an assembler and loader [5–7]. It was up to the programmer to code super-instructions in the library, to write TALM assembly code connecting the different instructions together, specifying control through data-flow instructions, and defining a placement file to map each instruction (or super-instruction) to a PE. It is evident that this task is not always trivial and is often prone to error.

In this work we propose *THLL* (TALM High-Level Language) and *Couillard*, a language and compiler for data-flow style execution. With *THLL*, the programmer annotates blocks of code that are going to become super-instructions, and further annotates the program variables that correspond to their inputs and outputs. *Couillard* then produces the `C`-code corresponding to each super-instruction to be compiled as a shared object to the target architecture and loaded by *Trebuchet*. Moreover, *Couillard* generates TALM assembly code to connect all super-instructions according to the user's specification. This assembly code represents the actual data-flow graph of the program. Control constructs such as loops and if-then-else statements that are not within super-instruction will also be compiled to TALM assembly code. This assembly code will then be converted to binary and used by *Trebuchet* to guide execution, following the data-flow rules. *Couillard* also generates a naive placement that is enough for most applications that follow regular parallelism patterns.

3.1. THLL

THLL, TALM High-Level Language, is a based on ANSI C and focus on providing high-level annotations to describe super-instructions and their input and output operands. In this section we discuss THLL annotations and provide some examples on how to use THLL to parallelize applications that fit into popular patterns (fork-join with reduction and pipelines) as well as applications that require more advanced parallelization techniques, such overlapping of I/O and computation.

3.1.1. Blocks and super-instructions

The annotation pair `#BEGINBLOCK` and `#ENDBLOCK` is used to mark blocks of code that will *not* be compiled to data-flow. Those blocks usually contain include files, auxiliary function definitions, and global variables declarations, to be used by super-instruction code in the dynamic library.

Super-instruction annotation is performed according to the following syntax:

```
treb_super < single|parallel> [input(<input_list>)]
                              [output(<output_list>)]
#BEGINSUPER
   //super-instruction body
#ENDSUPER
```

Super-instructions declared as `single` will always have only one instance in the data-flow graph, while instructions declared as `parallel` may have multiple instances that can run in parallel, depending on the placement and availability of resources at the host machine. In the example of Fig. 4 (described in more details in Section 3.1.4), there are `single` super-instructions at the beginning and end of the computation. In contrast, the inner code corresponds to `parallel` super-instructions.

3.1.2. Variables

After choosing the type of super-instruction (`single` or `parallel`) programmers should define their inputs and/or outputs. If super-instruction A has x as an output and super-instruction B, declared after A, has x as input, this means that there is a path between A and B in the data-flow graph, since A produces x and B consumes it. More specifically, if B is declared right after A this path will be a direct edge $A \rightarrow B$. Moreover, all variables used as inputs or outputs of super-instructions must be previously declared to guarantee that data will be exchanged correctly between instructions (without loss due to wrong type castings). Also, output variables used on parallel super-instructions must be declared as follows:

`treb_parout <type> <identifier>;`

The *Storage Classifier* `treb_parout` is used because parallel super-instructions, in general, have multiple instances, Therefore, output variables of parallel super-instructions will also have multiple instances, one for each instance of the related parallel super-instruction.

When using a `treb_parout` variable as input to another super-instruction it is necessary to specify the instance that is being referenced. To do so, *THLL* provides the following syntax:

```
<identifier>::< NUMBER |
                     * |
                 mytid |
       (mytid + NUMBER) |
      (mytid − NUMBER) |
               lasttid >
```

Consider a variable named x. The notation $x :: 0$ refers to instance 0 of variable x, while $x :: *$ refers to all instances of this variable (this provides an useful abstraction when a super-instruction can receive inputs from a number of sources, as in a reduction operation). Also, it is often convenient to refer to the instance of the current (parallel) super-instruction. If x is used as input to another parallel super-instruction, we can select x through the expression $x :: mytid$. To illustrate this situation, in the example of Fig. 4, each instance k ($0 \leqslant k \leqslant 1$, since there are 2 instances of each parallel super-instruction) of `Proc-2A` receives as input $p1Data :: k$, produced by `Proc-1`. Expressions with $+$ and $-$ operators are also allowed with *mytid*. For example, if a parallel super-instruction A produces operand x and another parallel super-instructions B uses specifies $x :: (mytid - 1)$ as input, it means that for a task i, B_i will receive x from $A_{(i-1)}$. Last, the reserved word *lasttid* refers to the last instance of a parallel super-instruction and can be used to specify inputs to parallel and single super-instructions.

For the cases where there are dependencies between instances of the same parallel super-instructions we can specify input variables using the following construct:

```
local.<identifier>::<(mytid + NUMBER) |
                    (mytid − NUMBER)>
```

For example, if we state that a parallel super-instruction S produces operand o and receives $local.o :: (mytid - 2)$, it means that S_i (instance i of S) depends on $S_{(i-2)}$. Moreover, it means that S_0 and S_1 do not have local dependencies. We can also specify operands that will be sent only to those independent instances of S. We use the following syntax:

```
starter.<identifier>::< NUMBER |
                             * |
                         mytid |
              (mytid + NUMBER) |
              (mytid - NUMBER) |
                       lasttid >
```

In the former example if we also define *starter.c* as an input of *S*, only S_0 and S_1 will receive this operand. A practical example of use of this constructs is to serialize distributed I/O operation to hide I/O latency, explained in Section 3.1.4.

The rationale to describe parallel code in super-instructions is simple. The developer first divides the code in blocks that can be run in parallel. Initialization and termination blocks will most often be `single`, whereas most of the parallel work will be in `parallel` blocks. The programmer next specifies how the blocks communicate. If the communication is purely control-based the programmer should further add an extra variables to specify these connections (a common technique in parallel programming). Note that the programmer still has to prevent data races between blocks unless speculative execution is used (which is not yet supported by the *Couillard* compiler).

3.1.3. Auxiliary functions

We have added special functions to the *Trebuchet* Runtime environment to help programmers identify instances of parallel tasks, to obtain information about the number of available processing cores in the host machine, to measure execution time and to manipulate command line arguments passed to the application.

The functions, `treb_get_tid()` and `treb_get_n_tasks()`, have been added to *Trebuchet* virtual machine and they can be called inside super-instructions code. The former returns the *task id* of that super-instruction's instance, while the latter returns the *number of tasks*. Those functions can be used to identify the portion of work to be done by each instance. Moreover, function `treb_get_n_procs()` is used to retrieve the number of available cores in the system. The number of cores to be used by *Trebuchet* threads can be limited by setting the environment variable `NUM_CORES` to the desired number before execution.

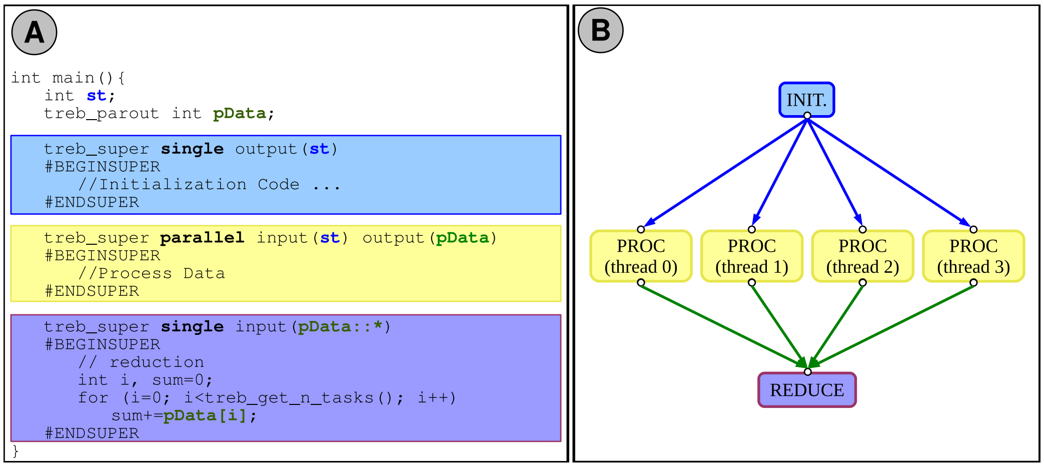
In our system, applications are executed in the *Trebuchet* virtual machine. Therefore, command line argument variables cannot be declared within the application's code. They need to be passed through *Trebuchet*'s command line. Thus, *Trebuchet* stores a vector of command line arguments. The number of arguments passed to application can be accessed using the `treb_get_n_args()` function, while each argument can be retrieved using the `treb_get_arg(number)` function.

When parallelizing applications, programmers often need to instrument their code to measure execution time of some portions of the application. This can be done using the `treb_get_time(RESOLUTION)` function. Timer resolution is an integer passed as parameter. Constants `TIME_s`, `TIME_ms` and `TIME_us` can be used to specify resolution in seconds, milliseconds and microseconds, respectively. The function returns a double value containing the time passed since the Epoch (it uses `clock_gettime()` function). To measure the execution time of a piece of code, programmers should call `treb_get_time()` at the beginning and at the end of that portion and subtract the values.

3.1.4. Illustrative examples

Fig. 2 shows how to use *THLL* to parallelize an application that follows a fork-join and reduction pattern. There are 3 super-instructions:

1. A `single` super-instruction that contains some initialization code (such as reading data from a file) and produces a token `st` that will release the execution of the processing phase.
2. A `parallel` super-instruction that process data in parallel (each instance processes a chunk of data) and produces an instance of `pData`.
3. A `single` super-instruction that receives all instances of `pData` (`pData::*`) and performs a reduction operation (the summation of all instances). Inside this super-instruction `pData` is seen as a vector with *N* elements (where *N* is the number of parallel tasks).



```
int main(){
   int st;
   treb_parout int pData;

   treb_super single output(st)
   #BEGINSUPER
      //Initialization Code ...
   #ENDSUPER

   treb_super parallel input(st) output(pData)
   #BEGINSUPER
      //Process Data
   #ENDSUPER

   treb_super single input(pData::*)
   #BEGINSUPER
      // reduction
      int i, sum=0;
      for (i=0; i<treb_get_n_tasks(); i++)
         sum+=pData[i];
   #ENDSUPER
}
```

Fig. 2. Example of fork-join and reduction with *THLL*. Variables and Super-instructions are color-coded with their corresponding edges and nodes. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

Fig. 3 provides an example on how to use *THLL* to hide I/O latency in an application. In this example chunks of data are read from a file, processed and result is written to another file. Pane *A* shows the different steps to be performed by super-instructions (inner code not shown): (i) initialization of variables and FILE pointers, (ii) reading, (iii) processing, (iv) writing and (v) closing of files. Pane *B* shows the associated data-flow graph, generated by *Couillard*. One can notice that reading and writing stages are described as parallel super-instructions, but since there are local inputs, they will be executed serially (although spread among different PEs). This construct allows the execution of each processing task to start as soon as the corresponding read operation has finished. It also allows writing the results of each processing task T_i without having to wait for tasks T_x, where $x < i$, to finish.

Fig. 4 provides an example of how to use *THLL* to describe a non-linear parallel pipeline. The example is a skeleton code of an application that reads a file containing a bag of tasks to be processed and writes the results to another file. The processing phase can be divided into 3 stages (`Proc-1`, `Proc-2` and `Proc-3`). Stage `Proc-2`, was separated into two different tasks (`Proc-2A` and `Proc-2B`), that are executed conditionally. Fig. 4 (pane *A*) shows TALM annotations, while the corresponding data-flow graph for 2 threads, generated by the *Couillard* compiler, is shown in Fig. 4 (pane *B*).

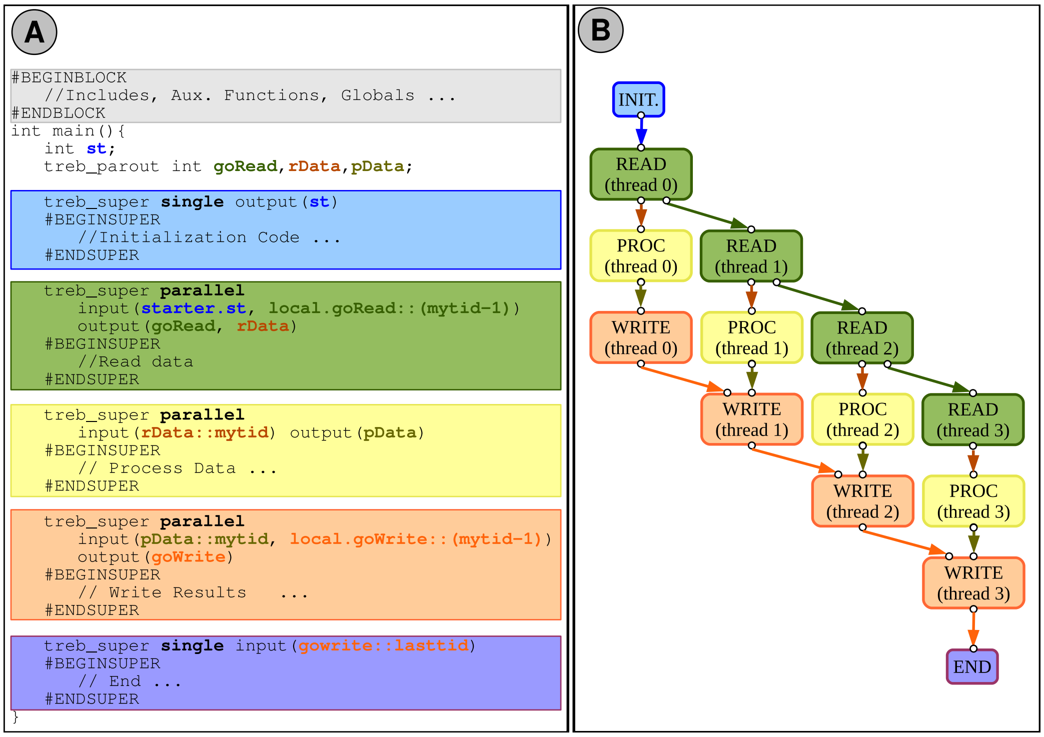
3.2. Couillard

Couillard is a data-flow compiler that converts *THLL* into TALM assembly code and generates super-instructions C-code (to be compiled into a dynamically linked library) and a graph representation of the program, using Graphviz notation [14]. *Couillard* also generates a naive placement for parallel super-instructions. Next, we describe *Couillard* front-end and back-end.

3.2.1. Front-end

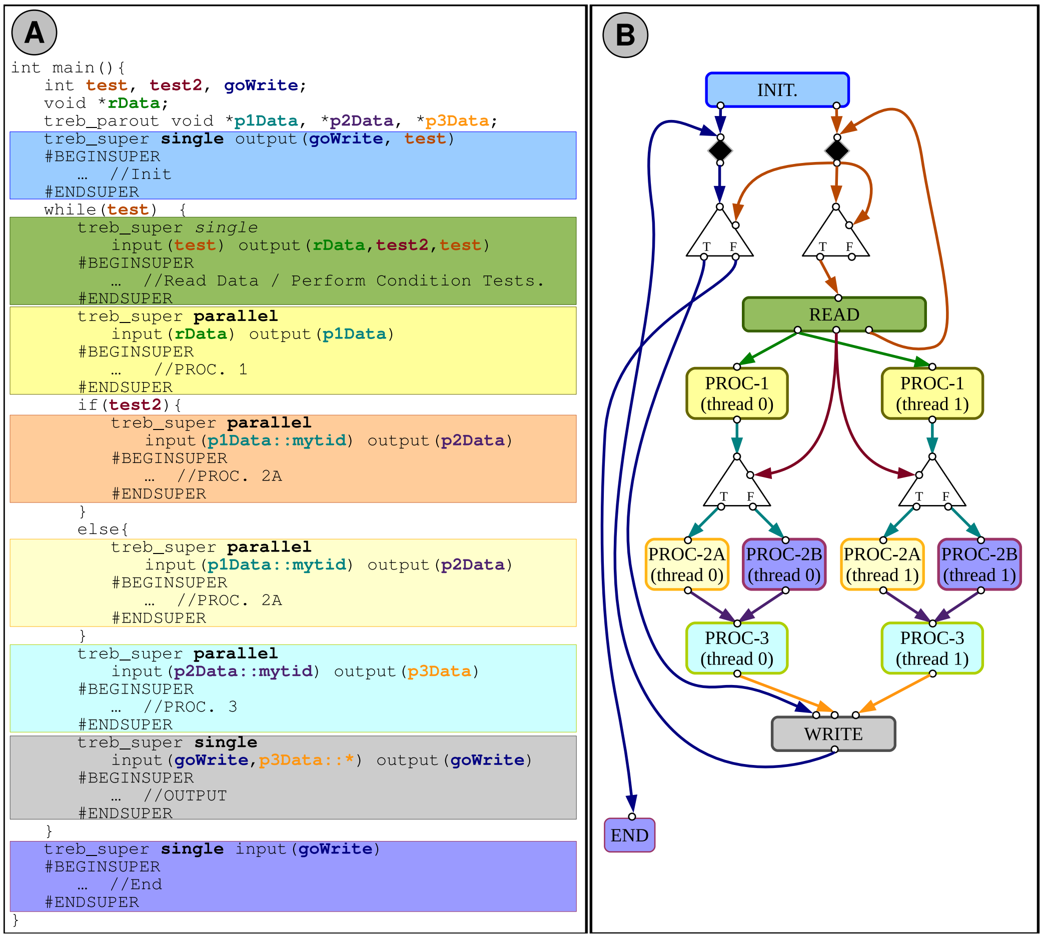
Couillard front-end uses PLY (Python Lex-Yacc) [15]. We assume that super-instructions take most of the running time of an application, as regular instructions are mostly used to describe the data and control relations between super-instructions. Since super-instruction code will be compiled using a regular `C`-compiler and regular instructions tend to be simple, *Couillard* does not need to support the full ANSI-`C` grammar.

Couillard, therefore adopts a subset of the ANSI-C grammar extended to support data-flow directives relative to super-instructions and their dependencies. We have also changed the syntax of variable declaration and access, which is necessary to parallelize super-instructions. The compiler front-end produces an AST (Auxiliary Syntax Tree) that will be processed to generate a data-flow graph representation.



```
#BEGINBLOCK
   //Includes, Aux. Functions, Globals ...
#ENDBLOCK
int main(){
   int st;
   treb_parout int goRead,rData,pData;

   treb_super single output(st)
   #BEGINSUPER
      //Initialization Code ...
   #ENDSUPER

   treb_super parallel
      input(starter.st, local.goRead::(mytid-1))
      output(goRead, rData)
   #BEGINSUPER
      //Read data
   #ENDSUPER

   treb_super parallel
      input(rData::mytid) output(pData)
   #BEGINSUPER
      // Process Data ...
   #ENDSUPER

   treb_super parallel
      input(pData::mytid, local.goWrite::(mytid-1))
      output(goWrite)
   #BEGINSUPER
      // Write Results   ...
   #ENDSUPER

   treb_super single input(gowrite::lasttid)
   #BEGINSUPER
      // End ...
   #ENDSUPER
}
```

Fig. 3. Example of how to hide I/O latency with TALM. Variables and Super-instructions are color-coded with their corresponding edges and nodes. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)



```
int main(){
   int test, test2, goWrite;
   void *rData;
   treb_parout void *p1Data, *p2Data, *p3Data;
   treb_super single output(goWrite, test)
   #BEGINSUPER
      … //Init
   #ENDSUPER
   while(test)  {
      treb_super single
         input(test) output(rData,test2,test)
      #BEGINSUPER
         … //Read Data / Perform Condition Tests.
      #ENDSUPER
      treb_super parallel
         input(rData) output(p1Data)
      #BEGINSUPER
         …  //PROC. 1
      #ENDSUPER
      if(test2){
         treb_super parallel
            input(p1Data::mytid) output(p2Data)
         #BEGINSUPER
            … //PROC. 2A
         #ENDSUPER
      }
      else{
         treb_super parallel
            input(p1Data::mytid) output(p2Data)
         #BEGINSUPER
            … //PROC. 2A
         #ENDSUPER
      }
      treb_super parallel
         input(p2Data::mytid) output(p3Data)
      #BEGINSUPER
         … //PROC. 3
      #ENDSUPER
      treb_super single
         input(goWrite,p3Data::*) output(goWrite)
      #BEGINSUPER
         … //OUTPUT
      #ENDSUPER
   }
   treb_super single input(goWrite)
   #BEGINSUPER
      … //End
   #ENDSUPER
}
```

Fig. 4. Example of non-linear parallel pipeline with TALM. Stages `PROC-2A` and `PROC-2B` are executed conditionally. Variables in the code are colored to match corresponding edges in the graph. Super-instruction nodes an their declaration are also color-coded. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

3.2.2. Back-end

After generating an Abstract Syntax Tree (AST) of a program, *Couillard* produces its corresponding data-flow graph. From this graph, it generates three output files:

1. A `.dot` file describing the graph in the Graphviz [14] notation. This file will be used to create an image of that graph, using the Graphviz toolchain. Although a Graphviz graph is not needed by *Trebuchet*, it may be useful for academic purposes or to provide a more intelligible look of the produced graph to the programmers that want to perform manual adjustments to their applications.
2. A `.fl` file describing the graph using TALM's ISA [6,7]. This file will be the input to *Trebuchet*'s Assembler, producing the `.flb` binary file that will be loaded into *Trebuchet*'s Virtual Machine. *Couillard* uses the `placeinpe` macro (described in Section 4.1) to propose a naive placement for `parallel` super-instructions.
3. A `.lib.c` file describing the super-instructions as functions, in C-code, to be compiled as a dynamically linked library, using any regular C-compiler. All inputs and outputs variables described with *Couillard* syntax are automatically declared and initialized within the generated function. Notice also that the super-instruction body does not need to parsed by *Couillard*. It is just treated as the value of a super-instruction node at the AST representation. This allowed us to focus only on the instructions necessary to connect super-instructions in a coarse-grained data-flow graph.

4. Instruction placement and work-stealing

Instruction placement has a strong correlation with performance. A good placement mechanism must consider two orthogonal goals: to exploit parallelism and avoid communication costs. On one hand independent instructions should be placed on different PEs so they can run in parallel. On the other hand, if two independent instructions *A* and *B* perform a small amount of work and communicate with instruction C very often, A, B and C should be placed on the same PE, which serializes execution, but avoids communication overhead.

Static placement algorithms define an initial instruction mapping. However, for irregular applications, static placement is not enough, since PEs' load will often become unbalanced due to intrinsic characteristics of input data. In this section we discuss our solutions to the placement problem.

4.1. Placement assembly macros

For programmers who wish to tune their applications by writing the dataflow-graph directly in assembly, we provide a macro that allows them to define instruction placement and assembly code in the same file. The `placeinpe` macro is also used by Couillard to define a naive placement as described in Section 4.2. The `placeinpe` macro has the following syntax:

```
placeinpe(<PE#>, <"STATIC"|"DYNAMIC">)
```

This macro can be used multiple times in the assembly code to redefine placement generation for instructions that follow. `STATIC` keyword indicates that all instruction that follow must be mapped to the same PE (indicated by `PE#`). `DYNAMIC` keyword indicates that each time a parallel super-instruction (or instruction) is found, each of its instances will be mapped to different PEs, starting by `PE#`.

4.2. Couillard naive placement

For applications that are balanced and follow a simple data parallelism pattern a naive placement algorithm should usually be enough. Regular applications that follow the fork/join pattern, for example, only need to have their `parallel` super-instructions equally distributed among PEs to achieve maximum parallelism.

This strategy is the one adopted by *Couillard*. It basically generates assembly code that uses `palceinpe (0,"DYNAMIC")` for all parallel super-instructions. For applications in which simple data parallelism does not suffice to obtain good speedups, a more complex static placement algorithm is necessary. Next we present such algorithm.

4.3. Static placement algorithm

Since our model relies on both static and dynamic scheduling of the instructions, we needed to develop strategies for static scheduling (as work-stealing is our dynamic scheduling mechanism). The algorithm we adopted for mapping instructions to PEs in compile time is a variation of previous list scheduling algorithms with dynamic priorities [16–18].

The most significant improvement introduced to the traditional list scheduling techniques in our algorithm is the adoption of the probabilities associated with conditional edges of the graph as a priority level for the instruction mapping. In a similar fashion to the execution models presented in [19], our dataflow graphs have conditional edges, namely edges that carry the output of instructions which are conditionally executed. The execution of such instructions depend on the series of control variables that guide the execution through the different branches of the dataflow graph. Given a point of decision in the graph, to select the path through which data should be carried forward, the compiler inserts a `steer` instruction (explained in [6]), which basically just forwards the input operand received on input port 1 through one of the two output ports, depending on the boolean value received in input port 0.

Traditionally, list scheduling algorithms try to map instructions that have dependencies close to each other. Since an input operand can come from multiple source instructions, due to conditional edges, real dependencies are only known during execution. Because of this uncertainty of the dependencies, our algorithm uses the probabilities associated with the edges (i.e. the probability that a certain edge will forward data) to prioritize dependencies that are more likely to really occur during execution.

In the dataflow graph $D(I, E)$, where I is the set of instructions in the graph and E is the set of edges connecting them, the probabilities of the edges are computed in the first step of our algorithm. Each edge $e \in E$ has a probability $\phi(e)$ associated with it, which is essentially the probability that an operand will be sent through that edge. The probabilities of the edges can be defined as functions of the boolean variables that control the `steer` instructions in the graph.

In order to obtain these probabilities, first it is necessary to estimate the probability that each boolean variable in the graph is true. This must be done in the same way as branch prediction hints are used in architectures that adopt them. Profiling executions of the dataflow graph may be done to come up with an estimation of the percentage of times that the variable is *True*. Based on this statistical profiling of control variables it becomes possible to define the edge probabilities in a recursive manner. First we shall introduce the notation we use:

- (a_p, b_q) represents an edge between the output port p of instruction a and the input port q of instruction b. The input/output may be omitted if the context does not require their specification.
- $bool(e)$ is the control variable forwarded through edge e.
- $B(e) = P(bool(e) = True)$ is the probability of $bool(e)$ being *True*, which is estimated via profiling of executions of the dataflow graph. Conversely, $\neg B(e) = (1 - B(e))$.
- $inports(i)$ the set of input ports of instruction i.
- $outports(i)$ the set of output ports of instruction i.

Now, given a dataflow graph $D(I, E)$ where I is the set of instructions and E the set of edges connecting them, we can define the probability $\phi(e)$ of each edge $e = (a, b) \in E$ as the following:

- 1, if a is a root (i.e. there is no $i \in I$ such that $(i, a) \in E$).
- $\sum\{\phi(e') \cdot B(e') | e' = (\cdot, a_0) \in E\}$, if a is a `steer` and e comes from output port 0 of a (i.e. the *True* port).
- $\sum\{\phi(e') \cdot \neg B(e') | e' = (\cdot, a_0) \in E\}$, if a is a `steer` and e comes from output port 1 of a (i.e. the *False* port).
- $\sum\{\phi(e') | e' = (\cdot, a_0) \in E\}$, otherwise.

It is also necessary to define the rest of the parameters we use in our algorithm. P is the set of processing elements available. In our current implementation, all PEs are equal so the average execution time of instructions do not vary depending on which PE they are executed. Hence, we just express the average execution time of instruction i as t_i.

The average time it takes to send the data corresponding to one operand from one PE to another is also considered homogeneous, since the current implementation uses shared memory. In case the data produced by one instruction is bigger then the value field of the operand format, the producer can just send the shared memory address of the data produced. This average communication time between PE p and PE q is represented as c_{qp} and C is the matrix that comprises all $c_{qp}, \{p, q\} \subseteq P$, where P is the set of processing elements available. The values of c_{pq} may come from profiling executions or from instrumentation of the Trebuchet. However, in our experiments we consider all c_{pp} (i.e. the cost of sending operands between two instructions mapped to the same processing element) as 0, since our focus is to enforce that instructions with strong dependencies are mapped to the same PE to preserve locality.

In Fig. 5 we present our algorithm for static scheduling of the instructions of a dataflow graph. It is important to notice that the list scheduling approach only works with acyclic graphs, so as a preparation to run the algorithm we remove the edges in the graph that return operands to the loop (i.e. the edges that go from the instructions in the loop body into `inctag` instructions, to send operands to the next loop iteration). In the initialization of the algorithm we set the priorities of each instruction. The priority of a instruction i is the summation of the average execution times of all instructions on the longest path from i the instruction to another instruction that is a leaf in the input dataflow graph.

4.4. Work-stealing

In some programs, the average execution time of the super-instructions is just unpredictable and may vary a lot depending on the inputs. These cases can cause computational load imbalance among PEs since the good performance of the placements produced by the static scheduling algorithm depends on the predictability of execution times. To address this matter we also implemented a work-stealing mechanism based on the algorithm proposed in [9]. To reduce the overheads due to synchronization and improve the load balance even further, it is possible for the programmer to specify which super-instructions can be stolen. Typically the programmer would want stolen the super-instructions whose execution times cannot be predicted and have significant impact on the overall program performance.

5. Experiments and results

Our goal is to obtain good performance in real applications and evaluate TALM for complex parallel programming. We studied how our model performs on four applications:

1. **Blackscholes**: Option pricing with Black–Scholes Partial Differential Equation (PDE) (belongs to the PARSEC benchmark suite [10]). A performance comparison between OpeMP, TBB, Pthreads and *THLL/Couillard/Trebuchet* was made.
2. **Ferret**: Content similarity search server (also from the PARSEC benchmark suite [10]). A performance comparison between TBB, Pthreads and *THLL/Couillard/Trebuchet* was made.
3. **Raytracing**: 3D image rendering. A performance comparison between OpeMP and *THLL/Couillard/Trebuchet* was made.
4. **Needleman–Wunsch**: Determines optimal global alignment between two DNA sequences. This application uses the wavefront method [8]. A performance comparison between OpeMP, TBB (Intel® Threading Building Blocks) and *THLL/Couillard/Trebuchet* was made.

The experiments were executed 10 times for each scenario in order to remove discrepancies in the execution time. We used as parallel platform a machine with four AMD Six-Core Opteron™8425 HE (2100 MHz) chips (24 cores) and 128 GB of DDR-2 667 MHz (32×4 GB) RAM dual channel, running GNU/Linux (kernel 2.6.34.7-66 64 bits). The machine was running in multi-user mode, but no other users were in the machine.

We started our study with a regular application: Blackscholes. It calculates the prices for a portfolio of European options analytically with the Black–Scholes partial differential equation (PDE). There is no closed-form expression for the Black–Scholes equation, and as such it must be computed numerically. The application reads a file containing the portfolio. Black–Scholes partial differential equation for each option in the portfolio can be calculated independently. The application was parallelized with multiple instances of the processing thread that were responsible for a group of options. Results were then written sequentially to an output file. The PARSEC suite already comes with 3 parallel versions of the Blackscholes

```
Input: Dataflow graph $D(I,E)$, the set of processing elements $P$, the inter-
  processor communication costs matrix $C$ and the average execution time
  $t_i$ of each instruction.
Output: A mapping of $I \to P$
  Calculate the priorities of all instructions and the probability $\phi(e)$ associ-
  ated with each edge $e \in E$
  Initialize the ready list with all $i \in I$ such that $\{j|(j,i) \in E\} = \emptyset$
  Initialize all $makespan(p)$, for $p \in P$ with 0
  while $ready \neq \emptyset$ do
      Remove the instruction $i$ with the highest priority from $ready$.
      MAP($i$)
      Mark all edges $(i, \cdot)$ going out of $i$
      Add to ready the instructions that have become ready after this (i.e.
  all input edges have been marked).
  end while
  procedure MAP($i$)
      $finish(i) \leftarrow \infty$
      for all $p \in P$ do
          Pick edge $e = (j,i) \in E$ that maximizes $(finish(j) + c_{qp}) \cdot \phi(e)$,
  where $j$ was mapped to processor $q$
          $aux \leftarrow max(finish(j) + c_{qp}, makespan(p)) + t_i$
          if $aux < finish(i)$ then
              $finish(i) \leftarrow aux$
              $chosenproc \leftarrow p$
          end if
      end for
      map i to $chosenproc$
      $makespan(p) \leftarrow finish(i)$
  end procedure
```

Fig. 5. Algorithm for scheduling data flow graphs.

benchmark: OpenMP, Pthreads and TBB. We have produced a *THLL* version of Blacksholes, following the same patterns present in the PARSEC versions to exploit parallelism. However, we observed that we could hide I/O latency and increase memory locality if we had multiple instances of the input and output threads. Thus, we also implemented Blackscholes according to the example shown at Section 3.1.4, Fig. 3.

Fig. 6 shows the results obtained for the Blackscholes benchmark. Using *THLL* and *Trebuchet* Runtime, it is possible to obtain good performance (comparable to Pthreads implementations) in a simple fashion. However, the flexibility of the language enables the programmer to achieve even greater results, with little effort, employing more complex techniques of parallelization. For this applications, standard deviation was irrelevant and, therefore, not included in the chart.

The second benchmark we considered is an irregular application called *Ferret*. This application is based on the Ferret toolkit which is used for content-based similarity search. It was developed at Princeton University, and represents emerging next-generation search engines for non-text document data types. Ferret is parallelized using the pipeline model and only a Pthreads version is provided with PARSEC. However, we had access to a TBB version of Ferret [20] which is also used in this experiment.

First, we observed that the task size in Ferret was quite small, and would result in high interpretation overheads by the virtual machine, specially when using a large number of cores, where the communication costs become more apparent. Therefore, we adapted the application to process blocks of five images per task, instead of one.

After that we instrumented the application to measure the time it took to read, process and write each block and verified that about 96.7% of execution time lies within the processing stage, while about 3.2% is on the reading stage and the writing stage is irrelevant. Moreover, the processing and reading stages presented significantly high standard deviations (35% and 17%, respectively), suggesting that a load balancing mechanism would help us achieve performance gains.

Finally, for *Trebuchet* the read stage contributes to load imbalance since it blocks the PE thread that is executing the reading super-instruction and no other instruction is executed in that PE during this time. The same can happen with TBB, since it uses worker threads and a read operation could block one of the worker thread. This does not happen with pthreads because it has exclusive threads for reading and writing and it implements the processing phase in 4 stages (as opposed to one in TBB and *Trebuchet)*. Thus, one can say that Pthreads relies on oversubscription to balance the load.

We implemented two parallel versions of ferret, both using pipeline patterns. The first one (basic skeleton depicted in Fig. 7) described I/O stages as `single` super-instructions and processing stages as `parallel` (we call that version *Trebuchet Single I/O*). The second version (basic skeleton depicted at Fig. 8) described I/O stages as `parallel` super-instructions with local dependencies to preserve execution order while allowing distribution to explore memory locality and balance I/O blocking time.

For the experiments we had to modify the Pthreads version to use schedule affinity to prevent threads from using more cores than allowed in each scenario (since Pthreads uses over-subscription).

Results presented in Fig. 9 show that the TBB version presented lower performance when compared to Pthreads. For fewer cores, TBB performed better, but it did not scale up. *Trebuchet* (both Single and Multiple I/O with work-stealing off) presented even lower performance. However, one should remember that Ferret is a very unbalanced application and Pthreads relies on oversubscription while TBB relies on work-stealing.

When we turned work-stealing (WS) on for *Trebuchet Single I/O* we improved performance a bit, but not enough to reach TBB or Pthreads. Next, we marked the processing stage super-instruction as the only one that could be stolen, which put us above TBB and right beside Pthreads. *Trebuchet Multiple I/O* did not improve with WS because there are reading stages in all PEs that could cause execution blocking. Therefore, for *Trebuchet Multiple I/O* version we relied on oversubscription and did the same for TBB. Executing 3 times more threads than cores allowed Trebuchet (and TBB) to outperform Pthreads. Notice that standard deviation was higher when we used work-stealing with *Trebcuhet*. This happened because collisions when accessing the task deque varied between executions.

This experiments showed that our selective work-stealing mechanism can provide better speedups than TBB and that, if programmers want to tune their applications and rely on oversubscription, good performance can also be achieved with little effort.

The next application is a Raytracing of sphere objects. This application processes a 800 $\times$ 600 pixels scene with about four thousand spheres. It sends six rays per pixel and recursion depth is set to 50, meaning that a primary ray can bounce up to 50 times. There are two nested loops to process lines and columns of the image. Basic parallelization (adopted by OpenMP and *Trebuchet* versions) follows the fork-join pattern (similar to the one shown on Fig. 2) to parallelize the inner loop, with means there is a barrier on each iteration of the outer loop.

We improved our *Trebuchet* version to hide I/O latency by breaking the loop in two parts: a processing stage (formed by a `parallel` super-instruction) and a writing stage (formed by a `single` super-instruction). Loop control was detached from loop body to result in parallelism explosion, allowing concurrent execution of multiple iterations of the processing stage while the writing stage preserved program order (it had a dependency with the previous iteration).

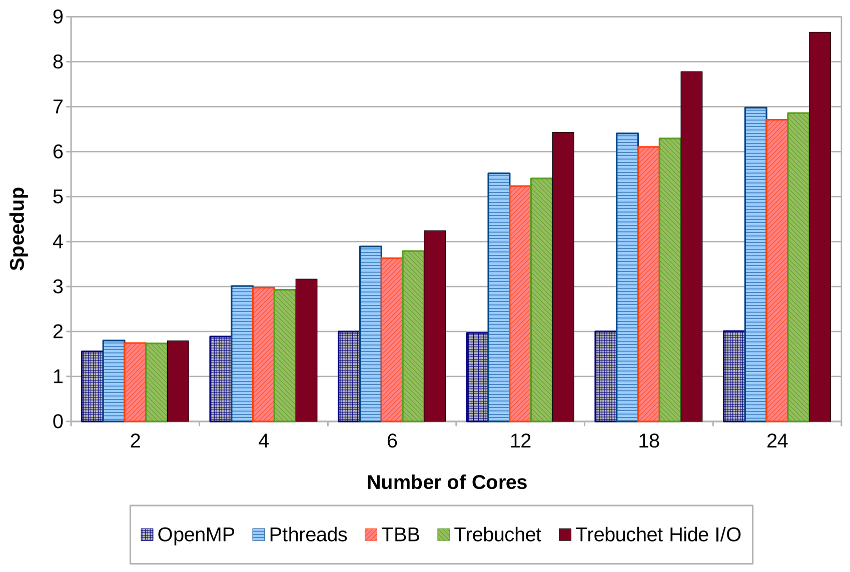
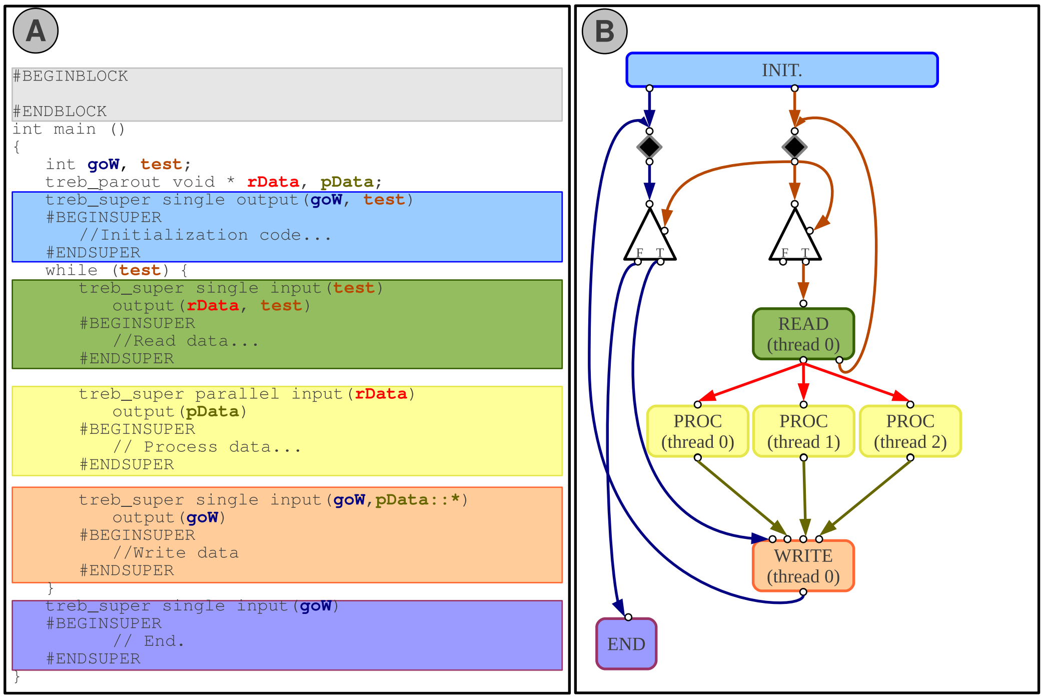


Fig. 6. Blackscholes results.



```
#BEGINBLOCK

#ENDBLOCK
int main ()
{
    int goW, test;
    treb_parout void * rData, pData;
    treb_super single output(goW, test)
    #BEGINSUPER
        //Initialization code...
    #ENDSUPER
    while (test) {
        treb_super single input(test)
            output(rData, test)
        #BEGINSUPER
            //Read data...
        #ENDSUPER

        treb_super parallel input(rData)
            output(pData)
        #BEGINSUPER
            // Process data...
        #ENDSUPER

        treb_super single input(goW,pData::*)
            output(goW)
        #BEGINSUPER
            //Write data
        #ENDSUPER
    }
    treb_super single input(goW)
    #BEGINSUPER
        // End.
    #ENDSUPER
}
```

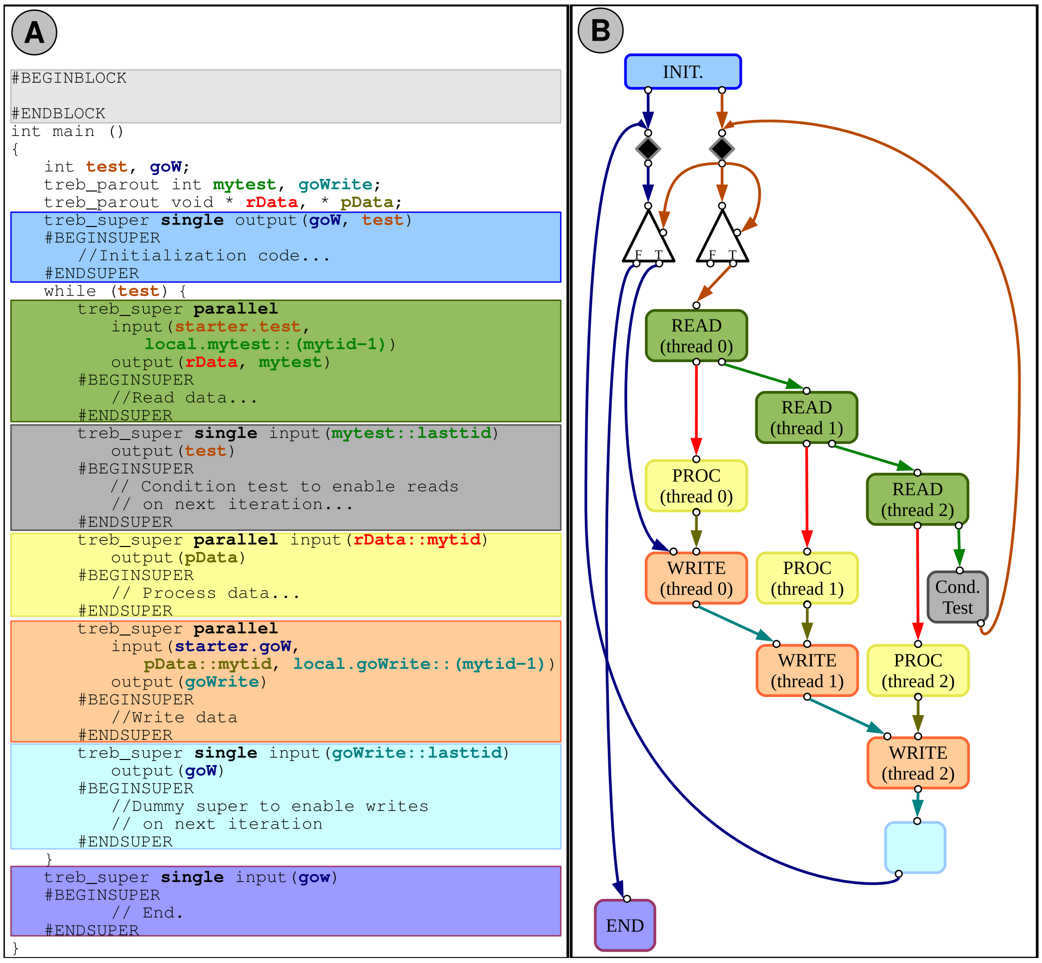
Fig. 7. Ferret Skeleton – *Trebuchet Single I/O* version.

Fig. 10 shows that *Trebcuhet* performs better even when it implements the application using the same restrictions as OpenMP. Performance is improved when hiding I/O techniques are added and more gains are achieved with work-stealing, since imbalance arises form high recursion depth. We obtained almost linear speedups (21.65 for 24 cores), while OpenMP provided speedups up to 16. For this applications, standard deviation was irrelevant and, therefore, not included in the chart.

Our last experiment focused on comparing *Trebuchet* with the latest features of TBB, the flow-graph constructs, which allows programmers to build a graph made of computation nodes that exchange messages, which is very similar to our approach. The main difference is that TBB is not a virtual data-flow machine, which suggests it might not deliver the same flexibility or performance for applications that would benefit the most from the data-flow model.

The Needleman–Wunsch (NW) algorithm [21] performs global optimal alignment (no heuristics) of two DNA (or protein) sequences. To do so, it computes a score matrix with size of N*M (N and M being the length of each sequence). Each element of the matrix depends on its left, upper and upper-left neighbors. For this reason, this algorithm is commonly parallelized using the wavefront method [8]. The application reads 2 DNA sequences from files, performs the alignment using the NW algorithm and prints the optimal alignment and score. We parallelized this applications by dividing the score matrix in blocks to increase granularity and to maintain memory locality. We have built 4 versions of NW:

1. **OpenMP**: Two loops were used to sweep all blocks on the matrix. The outer loop traverses the blocked matrix diagonally, while the inner loop computes (in parallel) blocks of each diagonal. There are some drawbacks in this approach. First, programmers need to transform diagonal coordinates (d, p) coordinates (d being the diagonal number and p being the position of that block in diagonal d) into regular (i, j) matrix coordinates. Second, the first diagonals have fewer blocks and will expose little parallelism. Third, for each diagonal there will be an OpenMP barrier. Last, although there is some parallelism between blocks that are not on the same diagonal, the classic wavefront method does not exploit it.
2. **TBB**: This version followed the wavefront template for TBB, described in [22]. TBB flow-graph creates edges and nodes dynamically (during execution), which might impact performance. *Trebuchet* allows dynamic dataflow to execute multiple instances of instructions. However, static versions of those instructions are created and placed in compile time to distribute computation and only use work-stealing (or other dynamic scheduling techniques) when it is really necessary.
3. **Trebuchet**: This version (shown in Fig. 11) used *THLL* to implement NW dependency pattern. Placement for this version was automatically generated by *Couillard* (naive placement was enough). It used a loop to compute lines of blocks and parallelism explosion to advance the computation of next lines (according to data dependencies).
4. **Trebuchet ASM**: This version (shown in Fig. 12) used TALM assembly to implement NW dependency pattern. The manual version allowed the elimination of loops. For this version we used a placement automatically generated with the algorithm described in Section 4.3. We also created a manually tuned placement to compare and test the efficiency of our placement algorithm.



```
#BEGINBLOCK

#ENDBLOCK
int main ()
{
   int test, goW;
   treb_parout int mytest, goWrite;
   treb_parout void * rData, * pData;
   treb_super single output(goW, test)
   #BEGINSUPER
      //Initialization code...
   #ENDSUPER
   while (test) {
      treb_super parallel
         input(starter.test,
            local.mytest::(mytid-1))
         output(rData, mytest)
      #BEGINSUPER
         //Read data...
      #ENDSUPER
      treb_super single input(mytest::lasttid)
         output(test)
      #BEGINSUPER
         // Condition test to enable reads
         // on next iteration...
      #ENDSUPER
      treb_super parallel input(rData::mytid)
         output(pData)
      #BEGINSUPER
         // Process data...
      #ENDSUPER
      treb_super parallel
         input(starter.goW,
            pData::mytid, local.goWrite::(mytid-1))
         output(goWrite)
      #BEGINSUPER
         //Write data
      #ENDSUPER
      treb_super single input(goWrite::lasttid)
         output(goW)
      #BEGINSUPER
         //Dummy super to enable writes
         // on next iteration
      #ENDSUPER
   }
   treb_super single input(gow)
   #BEGINSUPER
      // End.
   #ENDSUPER
}
```

Fig. 8. Ferret Skeleton – *Trebuchet Multiple I/O* version.

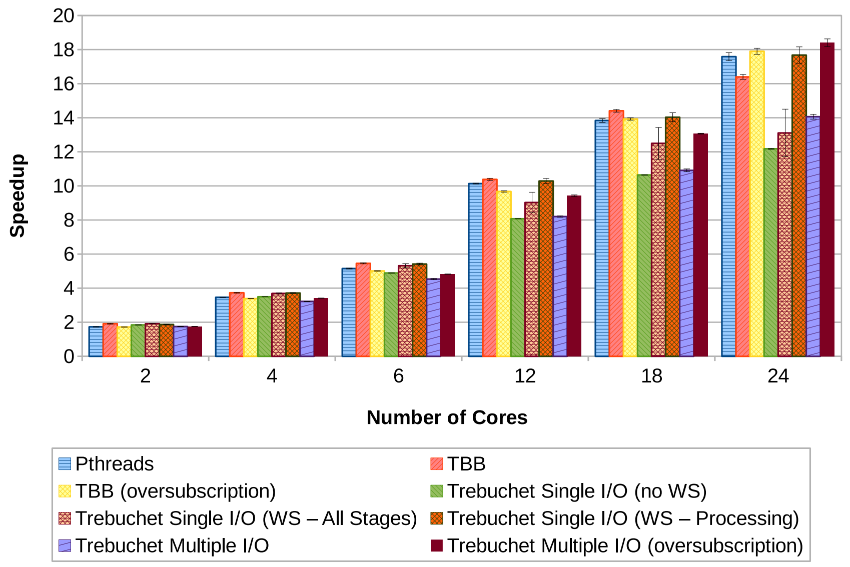


Fig. 9. Ferret results.

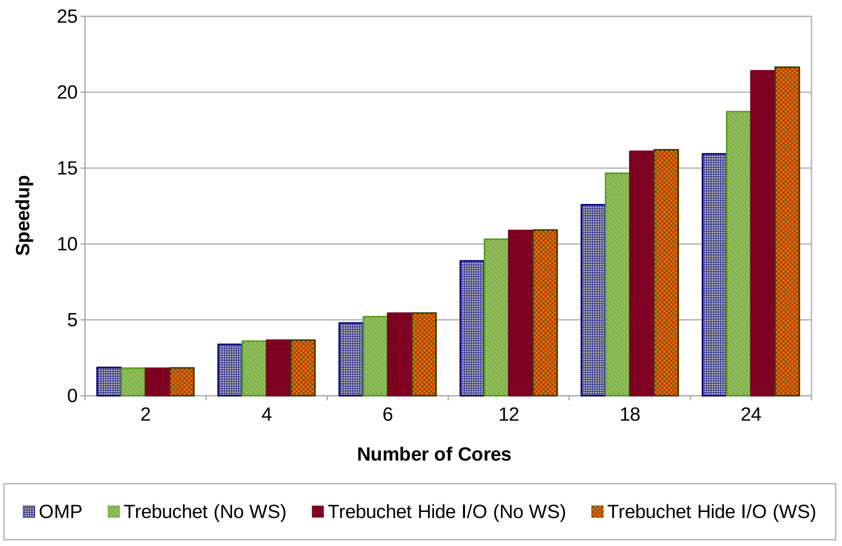
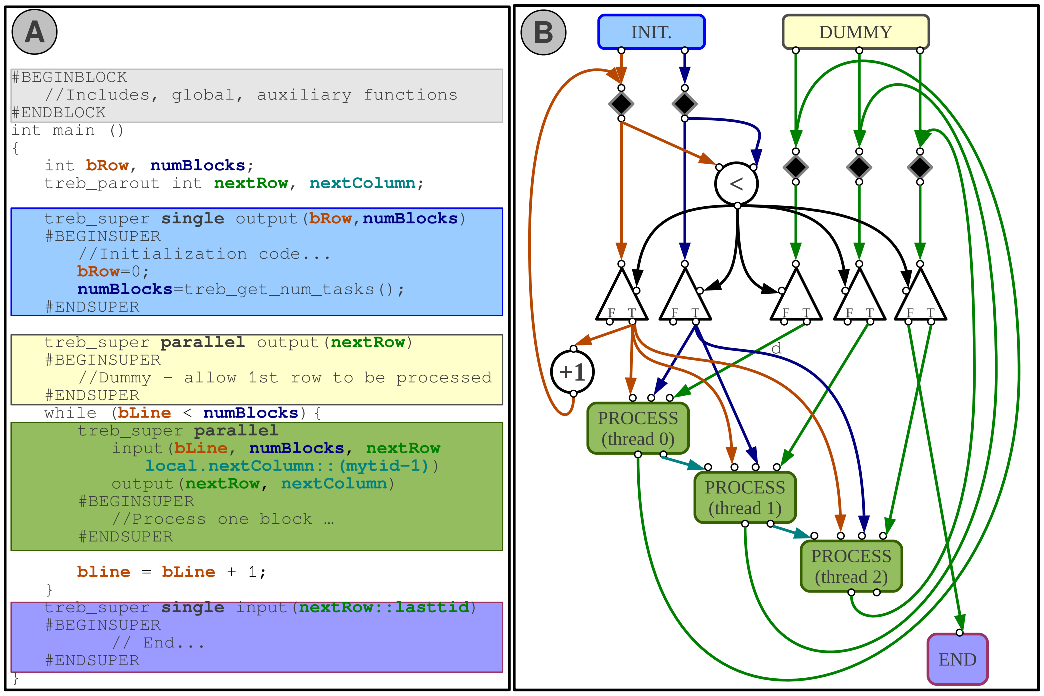


Fig. 10. Raytracing results.

We executed the experiments for `19 k x 19 k` up to `170 k x 170 k` matrices to evaluate the effect of granularity in performance. Block size was tunned for each implementation individually, to perform a fair comparison (TBB worked better with smaller blocks, while OpenMP and *Trebuchet* worked better with bigger ones). Each scenario was executed 10 times and standard deviation was irrelevant.

Fig. 13 shows that *Trebuchet* and TBB outperformed OpenMP in all experiments, which proves that data-flow approach can extract more parallelism from applications, specially for applications that follow more complex patterns. OpenMP version showed good performance only for medium matrices but did not scale well to bigger problem sizes. The *Trebuchet* version that used *THLL* provided similar speedups to the manual version, except for small matrices, where the extra loop instructions with small task grain made interpretation overheads more evident. We could also observe that automatic placement was very efficient, compared to our manual solution.



```
#BEGINBLOCK
   //Includes, global, auxiliary functions
#ENDBLOCK
int main ()
{
   int bRow, numBlocks;
   treb_parout int nextRow, nextColumn;

   treb_super single output(bRow,numBlocks)
   #BEGINSUPER
      //Initialization code...
      bRow=0;
      numBlocks=treb_get_num_tasks();
   #ENDSUPER

   treb_super parallel output(nextRow)
   #BEGINSUPER
      //Dummy - allow 1st row to be processed
   #ENDSUPER
   while (bLine < numBlocks){
      treb_super parallel
         input(bLine, numBlocks, nextRow
            local.nextColumn::(mytid-1))
         output(nextRow, nextColumn)
      #BEGINSUPER
         //Process one block …
      #ENDSUPER

      bline = bLine + 1;
   }
   treb_super single input(nextRow::lasttid)
   #BEGINSUPER
         // End...
   #ENDSUPER
}
```

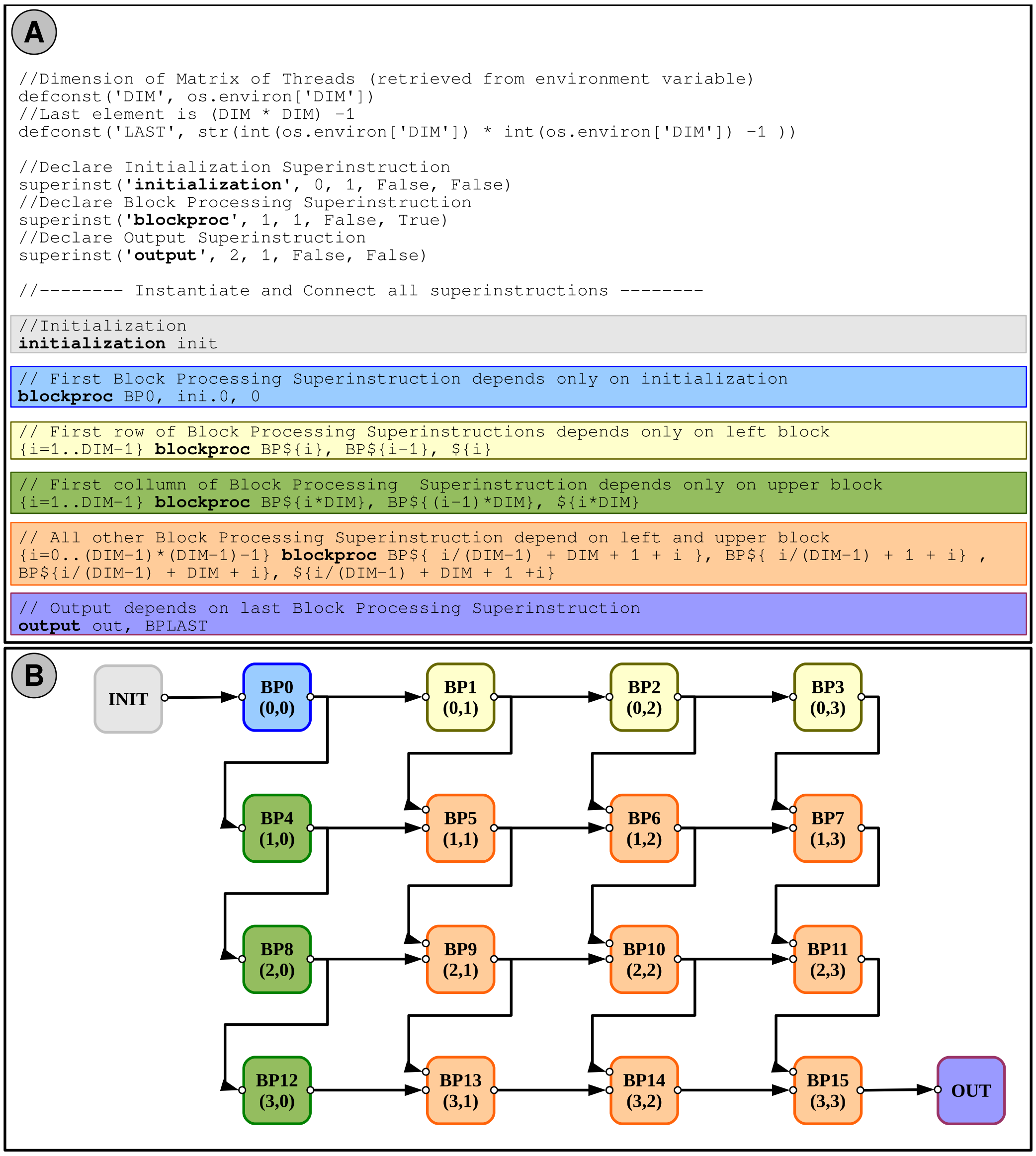
Fig. 11. Needleman–Wunsch *THLL* skeleton code.

Trebuchet and TBB performance were very similar when it comes to bigger problem sizes and for 170 k × 170 k matrix, *Trebuchet* scales better. This indicates that our interpretation overheads are still high, when compared to TBB dynamic flow-graph creation overheads, but we could scale better for bigger problems. We could not test execution for matrices bigger than 170 k × 170 k because we did not have enough memory (170 k × 170 k matrix used around 108 GB of memory, not counting other auxiliary structures).

Those results are very promising, since they show that *Trebuchet* presents better results than state-of-the art solutions available at the market while keeping ease of programming.

6. Related work

Data-flow is an long standing idea in the parallel computing community, with a vast amount of work on both pure and hybrid architectures [4,23,24]. Data-flow techniques are widely used in areas such as internal computer design and stream processing. Swanson's WaveScalar Architecture [4] was an important influence in our work, as it was a Data-flow architecture but also showed that it is possible to respect sequential semantics in the data-flow model, and therefore run programs written in imperative languages, such as C and C++. The key idea in WaveScalar is to decouple the execution model from the memory interface, so that the memory requests are issued according to the program order. To do so, WaveScalar relied on



```
//Dimension of Matrix of Threads (retrieved from environment variable)
defconst('DIM', os.environ['DIM'])
//Last element is (DIM * DIM) -1
defconst('LAST', str(int(os.environ['DIM']) * int(os.environ['DIM']) -1 ))

//Declare Initialization Superinstruction
superinst('initialization', 0, 1, False, False)
//Declare Block Processing Superinstruction
superinst('blockproc', 1, 1, False, True)
//Declare Output Superinstruction
superinst('output', 2, 1, False, False)

//-------- Instantiate and Connect all superinstructions --------

//Initialization
initialization init

// First Block Processing Superinstruction depends only on initialization
blockproc BP0, ini.0, 0

// First row of Block Processing Superinstructions depends only on left block
{i=1..DIM-1} blockproc BP${i}, BP${i-1}, ${i}

// First collumn of Block Processing  Superinstruction depends only on upper block
{i=1..DIM-1} blockproc BP${i*DIM}, BP${(i-1)*DIM}, ${i*DIM}

// All other Block Processing Superinstruction depend on left and upper block
{i=0..(DIM-1)*(DIM-1)-1} blockproc BP${ i/(DIM-1) + DIM + 1 + i }, BP${ i/(DIM-1) + 1 + i} ,
BP${i/(DIM-1) + DIM + i}, ${i/(DIM-1) + DIM + 1 +i}

// Output depends on last Block Processing Superinstruction
output out, BPLAST
```

Fig. 12. Needleman–Wunsch TALM manual assembly code.

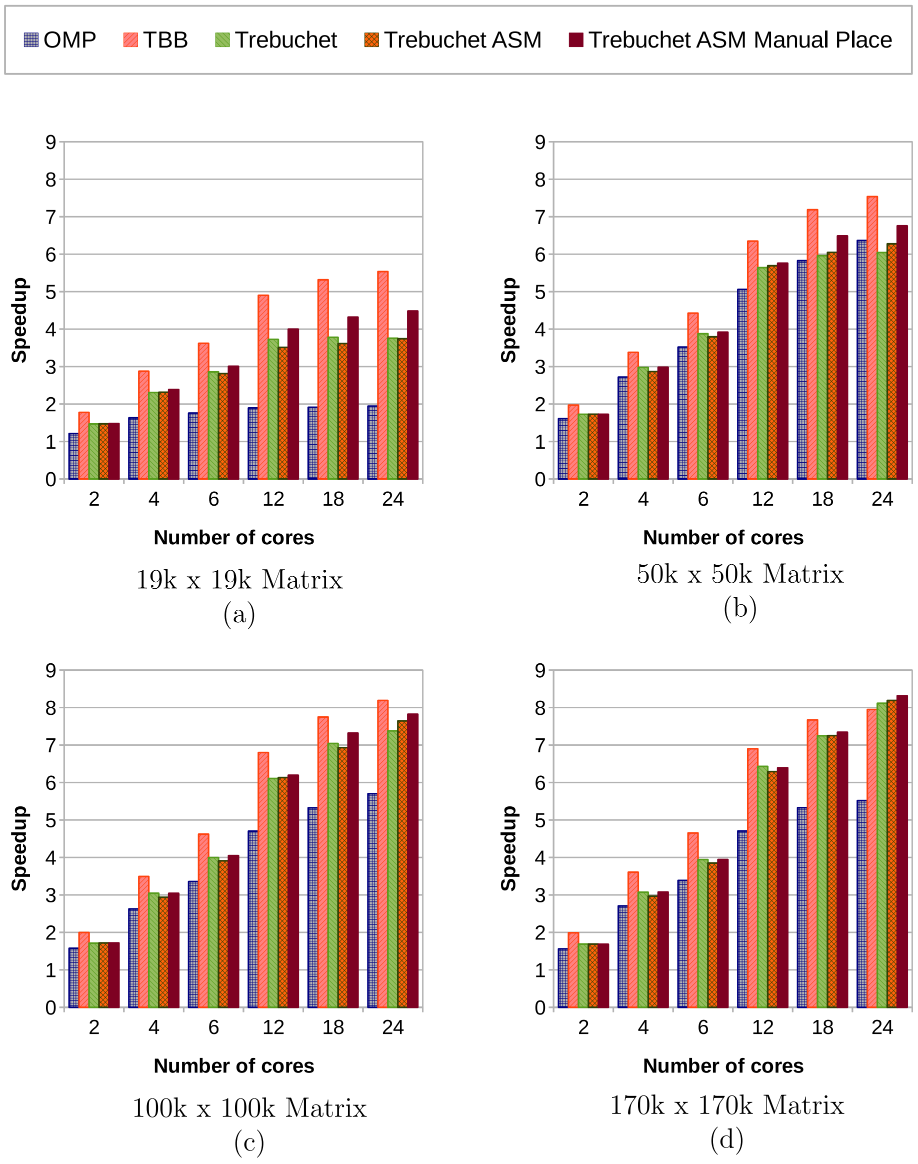


Fig. 13. Needleman–Wunsch results.

compiler to process memory access instructions to guarantee the program semantics. However, the WaveScalar approach requires a full data-flow hardware, that has not been achieved in practice.

Threading Building Blocks (TBB) [13] is a C++ library designed to provide an abstract layer to help programmers develop multi-threaded code. TBB enables the programmer to specify parallel tasks, which leads to a more high-level programming than implementing directly the code for threads. Another feature of TBB is the use of templates to instantiate mechanisms such as pipelines. The templates, however, have limitations. For instance, only *linear* pipelines can be described using the pipeline template. Support for data-flow execution style was implemented at TBB 4.0, the so called flow graph. Programmers can create and connect nodes that perform some computation and exchange messages.

Another project that relies on code augmentation for parallelization is DDMCPP [25]. DDMCPP is a preprocessor for the Data Driven Multithreading model [26], which, like TALM, is based on dynamic data-flow.

HMPP [27] is "an Heterogeneous Multi-core Parallel Programming environment that allows the integration of heterogeneous hardware accelerators in a seamless intrusive manner while preserving the legacy code". It provides a run time environment, a set of compilation directives and a preprocessor, so that the programmer can specify portions of accelerator codes, called codelets, that can run at GPGPU, FPGAs, a remote machine (using MPI) or the CPU itself. Codelets are pure functions, without side-effects. Multiple codelets implemented for different hardware can exist and the runtime environment will chose which codelet to run, according to hardware availability and compile directives previously specified. The runtime environment will also be responsible for the data transfers to/from the hardware components involved in the computation.

The Galois System [28–30] is an ‘‘object-based optimistic parallelization system for irregular applications’’. It comprises: (i) syntactic constructs for packing optimistic parallelism as iteration over ordered and unordered sets, (ii) a runtime system to detect unsafe accesses to shared memory and perform the necessary recovery operations and (iii) assertions about methods in class libraries. Instead of tracking memory addresses accessed by optimistic code, Galois tracks high-level semantics violation on abstract data types. For each method that will perform accesses to shared memory, the programmer needs to describe which methods can be commuted without conflicts (and under which circumstances). Gallois also introduces an alternative method to the commutative checks, since it may be costly [29]. Shared data is partitioned, attributed to the different processing cores and the system monitors if partitions are being ‘‘touched’’ by concurrent threads (which would raise a conflict). Despite the detection method used, the programmer needs to describe, for each method that access shared objects, an inverse method that will be executed in case of rollback. The runtime system is in charge of detecting conflicts, calling inverse methods and commanding re-execution.

SpiceC [31,32] provides similar constructs to the ones provided by THLL. However, those constructs are limited since they only describe control-flow between blocks (data in not exchanged between them). SpiceC requires a private memory space for each block to avoid side effects. This imposes a big restriction and may result in overheads. Moreover, SpiceC is limited to DOALL, DOACROSS, pipelining, and speculative parallelism. Our model is more generic, since it is inspired on dynamic data-flow architectures.

DAGuE [33] (Direct Acyclic Graph Environment) provides a runtime engine and a toolset to build, analyze, and pre-compile a compact representation of a DAG (Directed Acyclic Graph). DAGuE runtime environment implements a distributed multi-level dynamic scheduler, an asynchronous communication engine and a data dependencies engine. It allows programmers to describe parallels applications declaring different types of tasks defining their data dependencies. The user is also responsible for expressing task distribution using the JDF language, and distributing and initializing the original input data accordingly. Finding an efficient scheduling of the tasks, detecting remote dependencies and automatically moving data between distributed resources will be done by DAGuE runtime. *Trebuchet* is more flexible, since implements a full data-flow machine that allows execution of graphs with cycles.

7. Conclusions and future work

We have presented the *THLL* (TALM High-Level Language) and the *Couillard* compiler, that generates TALM data-flow assembly code from extended C-language. We also discuss static and dynamic placement issues and present our own solution to those problems. A performance evaluation of state-of-the-art parallel applications showed that using *THLL* to parallelize applications and using the *Trebuchet* Runtime (a TALM implementation for multicores) can present competitive speedups or even outperform handcrafted Pthreads and TBB code, up to 24 processors. Evaluation also shows that we can significantly improve performance by simply experimenting with the connectivity and grain of the building-blocks, applying techniques to hide I/O latency, or using selective work-stealing techniques, supporting our claim that *THLL*,*Couillard* and *Trebuchet* provides a flexible and scalable framework for parallel computing. Moreover, we show that, for bigger problem sizes, *Trebuchet* can scale up better than TBB using its most recent feature: the flow-graph.

Trebuchet’s performance relies on the flexibility and the ability to exploit parallelism of the TALM data-flow model. With *THLL* programmers can write applications only by describing true dependencies between blocks of code and avoid unnecessary synchronization. Popular APIs, such as OpenMP, often insert those barriers between parallel sections of code, resulting in extra overheads as seen in Section 5. TBB, with the flow-graph feature, uses a similar approach to avoid extra synchronization. However, nodes and edges are created and placed in runtime, which adds overheads.

Work on improving *Trebuchet* continues. Flexible scheduling is an important requirement in irregular applications, we thus have been working on improving the work stealing mechanism for *Trebuchet* runtime environment. Moreover, placement has a strong impact on applications performance and scalability. We are therefore studying ways improve automatic placement on *Trebuchet*.

We are also working on refining *Couillard* and on introducing new features to the support library. Extending *Couillard* to allow the use of templates to describe applications that fit well known parallel patterns (some described in this work) and to enable the use of *Trebuchet*’s memory speculation mechanisms [5] are subject of ongoing research. In our experiments, for example, we could see that the manual version of Needleman–Wunsch performed a bit better, since it did not use loops. Wavefront templates could be used to achieve this.

Our compiler and language do not yet support all constructs there are possible in a lower level. For example, some applications need a way to control parallelism explosion to avoid the waste of memory to store operands that will not be consumed for a long time (we have too much concurrency and lack on processing elements to run them). That can be done in assembly language using special `inctag` instruction with customized increment, called `inctagi`, so that each iteration produces a token that will release the *N*th iteration ahead. This is called execution window. A similar effect can be achieved in our high level language but it is not so easy to describe and the generated code does not use inctagi instructions. We need to add this functionality to our annotations.

This work is based in our experience with porting actual applications to the framework. Thus, finding applications that are interesting candidates to be parallelized with *THLL* is constantly within our research goals.

TALM's super-instructions could also be implemented to different hardware, using different languages, as in HMPP [27], as long as there is a way to call them from our virtual machine. Currently, super-instructions are compiled as functions in a dynamically linked library, but a interface to call GPGPU or FPGA accelerators and perform data-transfers could also be created in our environment. This is subject to on-going work.

Since we transform super-instructions into functions and build a data-flow graph connecting them, the code that is executed is not the same that was written by the programmer. For example, if there is an error inside the code of the 2nd super-instruction defined in the original code, when debugging, the programmer will see an error at function super2(). We would need to integrate with GDB or build our own debugger to help programmers more efficiently; to solve this is in our plans.

Acknowledgments

We thank FAPERJ, CAPES and the Euro-Brazilian Windows consortium for the financial support given to the authors of this work.

References

[1] J.R. Gurd, C.C. Kirkham, I. Watson, The Manchester prototype dataflow computer, Commun. ACM 28 (1) (1985) 34–52, http://dx.doi.org/10.1145/2465.2468.
[2] R. Nikhil, Executing a program on the MIT tagged-token dataflow architecture, IEEE Trans. Comput. 39 (3) (1990) 300–318, http://dx.doi.org/10.1109/12.48862.
[3] L.A. Marzulo, F.M. Franca, V.S. Costa, Transactional wavecache: towards speculative and out-of-order dataflow execution of memory operations, in: 2008 20th International Symposium on Computer Architecture and High Performance Computing, 2008, pp. 183–190. http://dx.doi.org/10.1109/SBAC-PAD.2008.29.
[4] S. Swanson, K. Michelson, A. Schwerin, M. Oskin, WaveScalar, in: Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36, IEEE Comput. Soc, 2003, pp. 291–302. http://dx.doi.org/10.1109/MICRO.2003.1253203.
[5] L.A.J. Marzulo, T.A. Alves, F.M.G. Franca, V.S. Costa, TALM: a hybrid execution model with distributed speculation support, in: International Symposium on computer architecture and high performance computing workshops, 2010, pp. 31–36. http://dx.doi.org/10.1109/SBAC-PADW.2010.8.
[6] T.A. Alves, L.A. Marzulo, F.M. Franca, V.S. Costa, Trebuchet: exploring TLP with dataflow virtualisation, Int. J. High Perform. Syst. Archit. 3 (2/3) (2011) 137, http://dx.doi.org/10.1504/IJHPSA.2011.040466.
[7] T.A. Alves, L.A.J. Marzulo, F.M.G. França, V.S. Costa, Trebuchet: Explorando TLP com Virtualizaçáo DataFlow, in: WSCAD-SSC'09, SBC, Sáo Paulo, 2009, pp. 60–67.
[8] J. Anvik, S. MacDonald, D. Szafron, J. Schaeffer, S. Bromling, K. Tan, Generating parallel programs from the wavefront design pattern, in: Proceedings of the 16th International Parallel and Distributed Processing Symposium, IPDPS '02, IEEE Computer Society, Washington, DC, USA, 2002, pp. 165–177. URL <http://dl.acm.org/citation.cfm?id=645610.661717>.
[9] N.S. Arora, R.D. Blumofe, C.G. Plaxton, Thread scheduling for multiprogrammed multiprocessors, Theory Comput. Syst. 34 (2) (2001) 115–144, http://dx.doi.org/10.1007/s002240011004.
[10] C. Bienia, Benchmarking Modern Multiprocessors (Ph.D. thesis), Princeton University, January 2011.
[11] D.R. Butenhof, Programming with POSIX Threads, Addison-Wesley Longman Publishing Co. Inc, Boston, MA, USA, 1997.
[12] L. Dagum, R. Menon, OpenMP: an industry standard API for shared-memory programming, IEEE Comput. Sci. Eng. 5 (1) (1998) 46–55, http://dx.doi.org/10.1109/99.660313.
[13] J. Reinders, Intel threading building blocks: outfitting C++ for multi-core processor parallelism, O'Reilly, 2007.
[14] Graphviz web-site. URL <http://www.graphviz.org>.
[15] D. Beazley, PLY – Python Lex-Yacc. URL <http://www.dabeaz.com/ply/>.
[16] G.C. Sih, E.A. Lee, A Compile-Time Scheduling Heuristic Heterogeneous Processor Architectures, vol. 4, no. 2, 1993, pp. 175–187.
[17] H. Topcuoglu, S. Hariri, I.C. Society, Performance-effective and low-complexity, Computer 13 (3) (2002) 260–274.
[18] W.F. Boyer, G.S. Hura, Non-evolutionary algorithm for scheduling dependent tasks in distributed heterogeneous computing environments, J. Parallel Distrib. Comput. 65 (9) (2005) 1035–1046, http://dx.doi.org/10.1016/j.jpdc.2005.04.017. URL <http://linkinghub.elsevier.com/retrieve/pii/S0743731505000900>.
[19] M. Lombardi, M. Milano, M. Ruggiero, L. Benini, Stochastic allocation and scheduling for conditional task graphs in multi-processor systems-on-chip, J. Schedul. (2010) 315–345, http://dx.doi.org/10.1007/s10951-010-0184-y.
[20] A. Navarro, R. Asenjo, S. Tabik, C. Ca\cscaval, Load balancing using work-stealing for pipeline parallelism in emerging applications, in: Proceedings of the 23rd international conference on Supercomputing, ICS '09, ACM, New York, NY, USA, 2009, pp. 517–518, http://dx.doi.org/10.1145/1542275.1542358.
[21] S.B. Needleman, C.D. Wunsch, A general method applicable to the search for similarities in the amino acid sequence of two proteins, J. Mol. Biol. 48 (3) (1970) 443–453, http://dx.doi.org/10.1016/0022-2836(70)90057-4. <http://www.sciencedirect.com/science/article/pii/0022283670900574>.
[22] Tbb wavefront template. URL <http://software.intel.com/en-us/blogs/2011/09/09/implementing-a-wave-front-computation-using-the-intel-threading-building-blocks-flow-graph>.
[23] K.M. Kavi, R. Giorgi, J. Arul, Scheduled dataflow: execution paradigm, architecture, and performance evaluation, IEEE Trans. Comput. 50 (8) (2001) 834–846, http://dx.doi.org/10.1109/12.947003.
[24] D. Burger, S. Keckler, K. McKinley, M. Dahlin, L. John, C. Lin, C. Moore, J. Burrill, R. McDonald, W. Yoder, Scaling to the end of silicon with EDGE architectures, Computer 37 (7) (2004) 44–55, http://dx.doi.org/10.1109/MC.2004.65.
[25] P. Trancoso, K. Stavrou, P. Evripidou, DDMCPP: The Data-Driven Multithreading C Pre-Processor, in: The 11th Workshop on Interaction between Compilers and Computer Architectures, Citeseer, 2007, p. 32.
[26] C. Kyriacou, Paraskevas Evripodou, P. Trancoso, Data-driven multithreading using conventional microprocessors, IEEE Trans. Parallel Distrib. Syst. 17 (10) (2006) 1176–1188, http://dx.doi.org/10.1109/TPDS.2006.136.
[27] R. Dolbeau, S. Bihan, F. Bodin, HMPP: a hybrid multi-core parallel programming environment, in: First Workshop on General Purpose Processing on Graphics Processing Units, 2007.
[28] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, L.P. Chew, Optimistic parallelism requires abstractions, in: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation – PLDI '07, PLDI '07, ACM Press, New York, New York, USA, 2007, p. 211, http://dx.doi.org/10.1145/1250734.1250759.
[29] M. Kulkarni, K. Pingali, G. Ramanarayanan, B. Walter, K. Bala, L.P. Chew, Optimistic parallelism benefits from data partitioning, in: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XIII, ACM, New York, NY, USA, 2008, pp. 233–243, http://dx.doi.org/10.1145/1346281.1346311.

[30] M. Kulkarni, P. Carribault, K. Pingali, G. Ramanarayanan, B. Walter, K. Bala, L.P. Chew, Scheduling strategies for optimistic parallel execution of irregular programs, in: Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures – SPAA '08, SPAA '08, ACM Press, New York, New York, USA, 2008, p. 217, http://dx.doi.org/10.1145/1378533.1378575.
[31] M. Feng, R. Gupta, Y. Hu, Spicec: scalable parallelism via implicit copying and explicit commit, SIGPLAN Not. 46 (8) (2011) 69–80, http://dx.doi.org/10.1145/2038037.1941564. URL <http://doi.acm.org/10.1145/2038037.1941564>.
[32] M. Feng, R. Gupta, Y. Hu, Spicec: scalable parallelism via implicit copying and explicit commit, in: Proceedings of the 16th ACM symposium on Principles and practice of parallel programming, PPoPP '11, ACM, New York, NY, USA, 2011, pp. 69–80. doi:10.1145/1941553.1941564. URL <http://doi.acm.org/10.1145/1941553.1941564>.
[33] G. Bosilca, A. Bouteiller, A. Danalis, T. Hrault, P. Lemarinier, J. Dongarra, Dague: A generic distributed DAG engine for high performance computing, Parallel Comput. 38 (1–2) (2012) 37–51. URL <http://dblp.uni-trier.de/db/journals/pc/pc38.html#BosilcaBDHLD12>.