Introducing Parallelism by Using REPARA C++11 Attributes

Conference Paper · February 2016

DOI: 10.1109/PDP.2016.115

CITATIONS
16

READS
203

5 authors, including:

Marco Danelutto
Università di Pisa
319 PUBLICATIONS **3,589** CITATIONS
SEE PROFILE

José Daniel García
University Carlos III de Madrid
92 PUBLICATIONS **514** CITATIONS
SEE PROFILE

Rafael Sotomayor
University Carlos III de Madrid
6 PUBLICATIONS **50** CITATIONS
SEE PROFILE

Massimo Torquati
Università di Pisa
138 PUBLICATIONS **1,400** CITATIONS
SEE PROFILE

Introducing Parallelism by using REPARA C++11 Attributes

M. Danelutto*, J. Daniel Garcia†, Luis Miguel Sanchez†, Rafael Sotomayor†, M. Torquati*
*Dept. of Computer Science, Univ. of Pisa
email: {marcod,torquati}@di.unipi.it
†Computer Science and Engineering Dep., Universidad Carlos III de Madrid
{josedaniel.garcia,luismiguel.sanchez,rafael.sotomayor}@uc3m.es

***Abstract*—Patterns provide a mechanism to express parallelism at a high level of abstraction and to make easier the transformation of existing legacy applications to target parallel frameworks. That also opens a path for writing new parallel applications. In this paper we introduce the REPARA approach for expressing parallel patterns and transforming the source code to parallelism frameworks. We take advantage of C++11 attributes as a mechanism to introduce annotations and enrich semantic information on valid source code. We also present a methodology for performing transformation of source code that allows to target multiple parallel programming models. Another contribution is a rule based mechanism to transform annotated code to those specific programming models. The REPARA approach requires programmer intervention only to perform initial code annotation while providing speedups that are comparable to those obtained by manual parallelization.**

Keywords: C++11 attributes, annotations, refactoring, multi-core, parallel programming, parallel patterns.

I. Introduction

As parallel architectures have become pervasive, parallel programming is becoming more necessary for software development. It spans from data center servers and workstations to mobile computing and embedded systems. With the advent of multi/many-core platforms, programmers are forced to exploit parallelism in their applications to improve the overall performance. One possible approach is to rewrite/redesign applications from scratch, but this is typically not acceptable because it is too costly in terms of development costs and time-to-solution.

Due to these parallel programming issues, many efforts on parallel computing research have been spent to provide high-level programming interfaces to exploit easily the available parallelism in multi-cores.

A possible approach is to make use of program annotations to extend the syntax. C++11 offers such opportunity through the standard language mechanism of *attributes* [5] which can be used to express parallelism in a sequential code without any modification. C++11 attributes allow to attach annotations to any syntactic element of a C++ program, integrating such annotations into the abstract syntax tree of the program.

In the context of the REPARA project [8], C++ attributes are used to define parallel regions of a code in an application and their parallel behaviour. By using the REPARA C++ attributes, the programmer clearly separates *what* is computed from *how* the computation is executed. This important design principle (*separation of concerns*), allows the programmer to easily and transparently introduce parallelism in a sequential C++ code.

The main contributions of this paper can be summarized as follows:

- A methodology to transform existing sequential C++ applications to multiple programming models.
- The definition of an attribute system for C++, used to introduce parallelism in sequential programs with no or minor changes to existing code base.
- The definition of a rule based system to transform the sequential annotated code into parallel code, capable to support multiple target programming models.

The rest of this paper is organized as follows: In Section II, related work is summarized. Section III presents the proposed methodology. Section IV defines attributes used to introduce parallelism. Section V describes a set of transformation rules for annotated source code. In Section VI, we show experimental results. Finally, in Section VII we summarize conclusions and outline some future work.

II. Related Work

REPARA C++ parallel interface makes use of C++11 generalized attributes approach which has some advantages over the standard parallel interfaces used for HPC programming. In terms of expressing parallelism, OpenMP [2], is mostly specialized to parallelize loops, and it easily introduces data and task parallel programming into sequential programs. Compared to OpenMP, which is pragma based, REPARA attributes provide additional flexibility as attributes can be attached to syntactic program elements while pragmas need to appear in separate source code lines. Moreover, higher level patterns, such as pipeline, are not easily expressed with OpenMP directives. Other options, such as TBB [4] or FastFlow [3] frameworks give more flexibility than OpenMP, but the sequential equivalence is not preserved due to the necessary changes to the code. Other simple options, such as Cilk [1], are language extension of C/C++ for multithreaded parallel computing. However, being extensions of the language, they are not directly supported by the C/C++ standard.

Parallelism in sequential code may be introduced directly at language level using specific constructs. This approach is used in several frameworks such as Scala [7] or X10 [9],

offering various degrees of support for parallel programming. The level of support from languages themselves can vary from simple threading primitives as in C++ to extensive abilities for distributed computing as in X10. Although these languages are gaining interest rapidly, their adoption in the embedded and HPC communities has been very low so far, probably due to performance concerns and the dominance of Fortran/C/C++ code. In the case of C++, many code bases are highly complex and with a large size, making the parallelization of existing code an extremely difficult task.

A common approach for multi-core platforms, is to incrementally introduce parallelism in the sequential code by using patterns having well-known parallel semantics [6]. This requires effort to identify both sources of parallelism and the adequate patterns. It is therefore tedious and error-prone.

The REPARA approach tries to overcome these limitations by introducing patterns such as *pipeline*, *task-farm*, *map* and *map-reduce* in a more controlled way. Moreover, by limiting the possible nesting of patterns, the parallel structure is simpler thus reducing the opportunity to introduce errors.

III. Methodology

The REPARA methodology designed to introduce parallelism in C++ code is structured into steps, each taking into account different aspects of the parallelisation of the code and implementing a clear *separation of concerns*. A *code annotation* phase identifies the code regions, or "kernels", subject to parallelisation. Then a *source-to-source* transformation phase deals with the refactoring of the identified parallel kernels into suitable run-time calls, according to the Hardware&Software (HW&SW) architecture(s) targeted. Eventually, a target-specific compilation phase generates the actual "object" (executable) code. The first phase only deals with "qualitative" aspects of parallelism exploitation. It is currently the responsibility of the application programmer to identify kernels but eventually will be implemented through proper rule-based rewriting tools. The second and third phases are completely automatic and specific to the HW&SW architecture targeted. The methodology steps are sketched in Fig. 1 -Left and they are further detailed as follows:

1) The first phase starts with a sequential program in which the user detects those parts of the code which can be annotated using C++ *attributes*, namely the *rpr::kernel* attribute. REPARA attributes are described in Sec. IV, here we just say that a *rpr::kernel* attribute can be used to annotate loops, function calls, single or compound statements.
2) In the second phase, the annotated code is passed to the *Source-to-Source Transformation Engine*. From the annotated source code, an *Abstract Intermediate Representation* (AIR) is generated. Then, the engine uses the AIR and a set of rules (described in Sec. V), specific for each *parallel programming model* for determining whether the corresponding code can be transformed into a *Parallel Programming Model Specific Code* (PPMSC)[1]. The PPMSC is the generated parallel code that is functionally equivalent to the original sequential code extended with parallel kernels execution according to the attribute parameters and to the selected programming model (e.g. Intel TBB, OpenMP, FastFlow, Cilk, etc.).
3) The third phase includes the target compilation phase using a standard C++ compiler and all low-level dependencies needed to run the code. The run-time used should provide coordination and all the mechanisms needed to support the deployment, scheduling and synchronization of kernels on the target platform(s).

[1]REPARA imposes some restrictions on the parallelizable source code when targeting specific hardware

IV. Expressing parallelism through attributes

REPARA C++ attributes can be used to define parallel regions in a code base, their behaviour (e.g. `rpr::map`) and their parameters (e.g. `rpr::in`). We refer to these parallel regions as *kernels*. An attribute is attached to a syntactic entity (e.g. a statement, loop, or definition), as defined by the standard C++ grammar. In general, an annotation precedes the syntactic element it is annotating to and does not require any preprocessing (a key difference with *pragma* based solutions). Table 1-Right summarizes the subset of the REPARA C++ attributes considered in this paper.

In this section, we introduce the REPARA attributes through three different usage scenarios: a parallel for-loop, a task parallel computation and a pipeline pattern.

A. Loop parallelism

Listing 1 shows a basic map compitations using REPARA C++ attributes. The attribute `rpr::kernel` annotates the subsequent single or compound statement expressing the programmers intent of marking it as a kernel region. Kernel nesting is not considered in the REPARA model, therefore when two or more kernels are nested, inner annotations are ignored.

Additional attributes may be applied to a kernel region to refine intentions and to provide additional information. For example, `rpr::map` or `rpr::farm` can be used to express the expected parallel pattern transformation. In absence of additional information, `rpr::map` is applied by default to a kernel loop. The *map* pattern is used for split-execute-merge computations where multiple instances of the same code block is applied to different elements of a dataset (*data parallelism*).

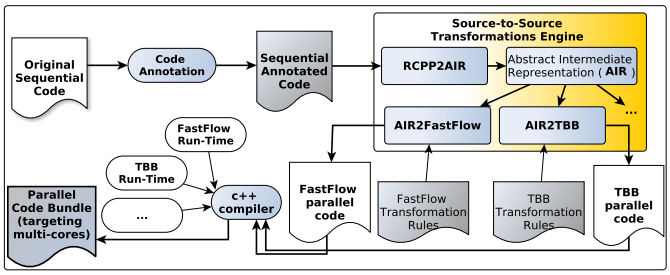
Additionally, the `rpr::in` and `rpr::out` attributes are used to identify input and output parameters of the kernel, respectively. Input/output sets do not need to be disjoint, allowing a parameter to be both input and output when needed.

Listing 1: Map computation with `REPARA` attributes.

```
[[rpr::kernel,rpr::map,rpr::in(C,CN,b),rpr::out(C)]]
for(long i=0;i<mblocks;++i)
 for(long j=0;j<nblocks;++j) {
  double *Cij=&C[b*(i*CN+j)];
  zerosBlock(b,Cij,CN);
 }
```

Listing 2: Asynchronous computation with `REPARA` attributes.

```
for(long i=0; i<mblocks; ++i){
 for(long j=0; j<nblocks; ++j) {
  [[rpr::kernel,rpr::async,rpr::in(C,CN,b),
```



REPARA C++ attr.	Description
`rpr::kernel`	Defines parallel region.
`rpr::pipeline`	Pipeline pattern.
`rpr::async`	Asynchronous kernel.
`rpr::sync`	Synchronizes asynchronous kernels.
`rpr::farm`	Task-farm pattern.
`rpr::map`	Map pattern.
`rpr::in`	Input parameters.
`rpr::out`	Output parameters.

Fig. 1: Left:) REPARA attributes compilation steps for multi-core targets. Right:) Subset of the *REPARA* C++ attributes.

```
        rpr::out(C)]]{
    double *Cij = &C[b*(i*CN+j)];
    zerosBlock(b, Cij,CN);
  }}}
[[rpr::sync]]; // empty statement
```

B. Task parallelism

In task parallelism several kernels are executed in parallel with a master program execution flow. Listing 2 shows an asynchronous computation through task parallelism for independent iterations. In contrast with default kernel annotation, we use `rpr::async` to signal an asynchronous kernel where the master flow of execution does not wait for kernel termination, but goes on in the execution until a synchronization point is reached. It may be either a non-asynchronous kernel or a statement annotated with the attribute `rpr::sync`.

The asynchronous kernels do not have any mechanism to ensure memory consistency. Thus, the programmer is responsible for keeping consistency when using this mechanism for data independent tasks.

C. Pipeline

The pipeline pattern combines several stages that need to be executed in a particular order and where data flows through those stages. Each stage performs some processing on the input data and provides the output to the next stage (if present). The `rpr::pipeline` attribute identifies a block with several stages as a *pipeline* pattern. The stages are identified by means of the `rpr::kernel` attribute and the data flowing through the stages by the `rpr::stream` attribute.

Listing 3 presents an image filtering pipeline use case.

Listing 3: Pipeline computation in an image processing application.

```
[[rpr::pipeline,rpr::stream(path,img)]]
while( fit.getNext(path) ) {
  [[rpr::kernel,rpr::in(path,img),rpr::out(img)]]
  ReadImage(filepath, img);
  [[rpr::kernel,rpr::in(img),rpr::out(img)]]
  ApplyFilter1(img);
  [[rpr::kernel,rpr::in(img),rpr::out(img)]]
  ApplyFilter2(img);
  [[rpr::kernel,rpr::in(path, img)]]
  WriteImage(filepath, img);
}
```

Usually, the data stream is created at the beginning of the first stage of the pipeline. Produced data will be defined as output parameter in the first kernel of the pipeline. The data stream will be deleted at the end of the last stage of the pipeline, including total or partial data stream.

The semantic of streaming input and output attributes ensures that inputs are buffered until the next activation of the producer kernel, and outputs are buffered until the next activation of the consumer kernel. The pipeline uses the loop termination condition to determine the end of the stream. In case of nested loops, the pipeline will use the combination of the loop conditions defined below the pipeline attribute definition.

V. Transformation Rules

In this section, a subset of the rules used to introduce parallel patterns for the REPARA attributes presented in Sec. IV are described. The transformation rule phase of the REPARA workflow is based on static analysis techniques. The goal is to identify the transformation opportunities offered by the set of rules associated with the selected run-time.

Rules are defined as *rewriting rules* detailing right and left hand side through "pseudo code" by using the FastFlow run-time[2] syntax. A more complete set of transformation examples for the FastFlow run-time can be found in the REPARA website [8].

```
[[rpr::kernel,rpr::map(nw),rpr::in(x0),rpr::out(x1)
    ]]
for(idx:{start,stop,step}) {body{x0,x1};}
```

```
ff::ParallelFor pf(nw);
pf=parallel_for({start,stop,step},
[&](long idx){body{x0,x1};});
```

TABLE I: For-indep rule

The $\boldsymbol{R_{for-indep}}$ rule I introduces parallelism for a kernel loop with independent iterations:
The optional nw parameter is the maximum parallelism degree used to execute the loop in parallel. If multiple loop kernels are present in a block of code, a single FastFlow ParallelFor instance is created, as an optimization.The $\boldsymbol{R_{pipe-loop-stream}}$ rule II introduces pipe parallelism with stream items generated through a loop:

The first stage of the pipeline encapsulates the loop condition and generates the stream of tasks after the execution of the first kernel. The last stage deallocates tasks and keeps alive the run-time returning GO_ON. To identify the *end-of-stream*, a spacial task (EOS) is explicitly generated by the first stage. The EOS task starts the pipeline termination phase.

[2]http://mc-fastflow.sourceforge.net/

```
[[rpr::pipeline,rpr::stream([x_0,x_1 ... x_M,x_N]+ ]]
loop(cond) {
 [[rpr::kernel, rpr::in(x_0) [,rpr::out(x_1)] ]]
 Kern_1;
 ...
 [[rpr::kernel,rpr::in(x_M [,rpr::out(x_N)] ]]
 Kern_N;
}
```

```
ff_Pipe<> pipe;
struct TaskT { foreach i in (1,N) decltype(xi) xi;
     };
pipe.add_stage([&](TaskT *in) {
 foreach i in (1,n) auto &x_i=(*in).x_i;
 if(i==0) { //first stage
  loop(cond){
   Kern_1; TaskT *out=make_task({x_0[,...,x_i]});
   ff_send_out(out);
  }
  return(EOS);//produces the end-of-stream
 } else Kern_i;
 if (i==n) {delete_task(in);return GO_ON};
 return in;
}); pipe.run_and_wait_end(); //starts pipeline
```

TABLE II: Pipe-loop-stream rule

```
[[rpr::pipeline,rpr:stream([x_0$,x_1 ... x_M,x_N]+
    ]]
loop(cond) {
 ...
 [[rpr::kernel,rpr::farm(nw),rpr::in(x_i),rpr::out(
     x_(i+1)) ]]
  Kern_i
  ...
}
```

```
ff_Pipe<> pipe;
struct TaskT { foreach i in (1,n) decltype(x_i) x_i;
     };
...
auto F_(x_i)=[&](TaskT *in) {
 foreach j in (i 1,i) auto &x_j=(*in).x_j;
 Kern_i;
 return in;
};
...
ff_Farm<TaskT> farm(F_(x_i),nw);//defines a farm
pipe.add_stage(farm);//adds the farm stage
...
pipe.run_and_wait_end();
```

TABLE III: Pipe-farm rule

```
[[rpr::pipeline,rpr:stream([x_0,x_1 ... x_M,x_n]+ ]]
loop(cond) {
 ...
 [[rpr::kernel,rpr::map(nw),rpr::in(x_i),rpr::out(x_
     (i+1))] ]]
  for(idx:{start,stop,step})
   {body{x_i,x_(i+1)};}
 ...
}
```

```
ff_Pipe<> pipe;
struct TaskT { foreach i in (1,n) decltype(x_i) x_i;
     };
...
struct map_{x_i}_t:ff_Map<TaskT>{ // def the map
 map_{x_i}_t(size_t nw):ff_Map<TaskT>(nw){}
 TaskT* svc(TaskT* in) {
  auto &x_i=(*in).x_i; auto &x_(i+1)=(*in).x_(i+1);
  parallel_for(start,stop,step,
   [&](long idx) {body{x_i,x_(i+1)};});
  return in;
 };
} map_(x_i);
pipe.add_stage(map_(x_i)); ...
```

TABLE IV: Pipe-map rule

Within a pipeline, it is possible to have kernels annotated as *rpr::farm* (i.e. kernel replication). The user may set both the number of replicas of the kernel (by default is the number of cores available on the machine) and the task ordering policy (the default policy is *unordered*). The $\boldsymbol{R_{pipe-farm}}$ rule III handles farm stages in a pipeline:

In case the pipeline has a single kernel and it is annotated as a *rpr::farm*, then it can be optimized by generating a single FASTFLOW task-farm and by placing the loop condition directly in the FASTFLOW task-farm scheduler.

The $\boldsymbol{R_{pipe-map}}$ rule IV handles map stages in a pipeline:

In the FASTFLOW framework, a map stage in a pipeline is just an optimized stage wrapping a ParallelFor pattern. If the pipeline has a single kernel and it is annotated as a *rpr::map*, then it a single (optimized) ParallelFor computation is generated inside a sequential loop (pipeline's loop).

Eventually, the parallel execution of independent tasks may be introduced using the $\boldsymbol{R_{bag-of-tasks}}$ rule V:
In the FASTFLOW framework, execution of asynchronous tasks is handled by the *taskf* pattern, which executes functions or C++11 lambdas in parallel on a multi-core platform using a dynamic scheduling policy. In order to use the *taskf* pattern, REPARA asynchronous kernels are wrapped in C++11 lambdas.

```
[[rpr::kernel,rpr::in(...),rpr::out(...),rpr::async
    ]] K_1;
 ...
[[rpr::kernel,rpr::in(...),rpr::out(...),rpr::async
    ]] K_N;
[[rpr::sync]] <statement>;
```

```
auto F_{K_1} = [&](){K_1;}; //lambda wrapper
...
auto F_{K_N} = [&](){K_N;}; //lambda wrapper
ff_taskf tf;
tf.run();
tf.AddTask(F_{K_1});
...
tf.AddTask(F_{K_N});
tf.wait(); //tasks barrier
<statement>;
```

TABLE V: Bag of tasks rule

VI. EXPERIMENTS

In this section, we validate a subset of REPARA attributes by showing the results obtained running an images filtering applications. It is important to remark that the test was not optimized for obtaining the best possible parallel performance, instead the purpose is to illustrate the methodology and the results that can be obtained. The low-level run-time framework used is the latest version of FASTFLOW[3].

The target architecture for the experiments is a dual-socket NUMA Intel multi-core Xeon E5-2695 Ivy Bridge micro-architecture running at 2.40GHz featuring 24 cores (12+12) each with 2-way Hyperthreading. Each core has 32KB private L1, 256KB private L2 and 30MB shared L3. The operating system is Linux 2.6.32 x86_64 shipped with CentOS 6.5. We compiled our tests using GNU gcc 4.8.2 with optimization flag –O3.

[3]http://mc-fastflow.sourceforge.net/

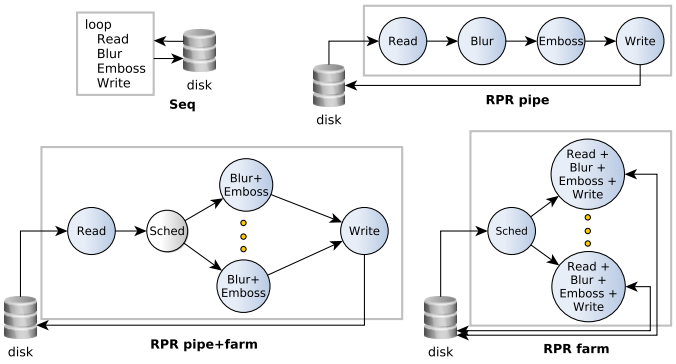


Fig. 2: Implementation schemas tested.

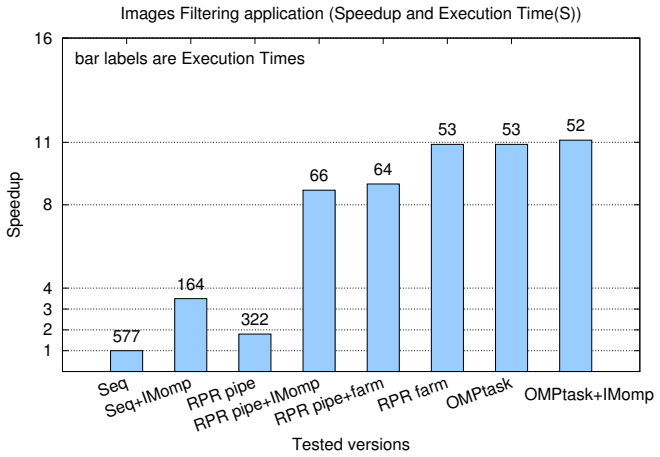


Fig. 3: Performance obtained running the tests on 512 images of different size.

For the test, we used the ImageMagick library[4] (IM hereinafter) version 6.8.9 to manipulate the images and apply the filters, In our tests we considered 512 JPG images of different size (from few KBs up to few MBs). The Blur and the Emboss filters, are applied to each image.

The IM library, internally uses OpenMP to take advantage of multi-core platforms (when the OpenMP support is enabled in IM, we refer to this version as IMomp). The number of OpenMP parallel threads can be controlled by using the environment variable `OMP_NUM_THREADS`.

For this test, we compare the results obtained with three different parallelizations using the REPARA attributes with a corresponding hand-written parallel code parallelized using OpenMP 3.0 pragmas (whenever possible). The first two versions follow the $\boldsymbol{R_{pipe-loop-stream}}$ and $\boldsymbol{R_{pipe-farm}}$ rules, respectively. The third version features a disk r/w optimization of the second version.

The results obtained running all these versions are sketched in Fig. 3. Bars' labels, represent the overall time (i.e. I/O plus computation time).

As we can see, the "RPR pipe" version obtains $1.8\times$ speedup w.r.t. the "Seq" version, whereas the "RPR pipe+IMomp" version obtains $2.6\times$ speedup w.r.t. the "Seq+IMomp" and $8.7\times$ speedup w.r.t. the "Seq" version. The "RPR farm" version, which uses all machine resources, obtains a good speedup, aligned with the hand written "OMPtask" version. Overall, the speedup of this application is good although not optimal (the maximum speedup obtained is 11.1 using all machine cores by the "OMPtask+IMomp" version), and this is mainly due to the very high memory pressure when computing multiple images in parallel.

[4]http://www.imagemagick.org

VII. Conclusions and Future Directions

In this paper we introduced the REPARA approach for expressing parallel patterns and transforming sequential C++ code into parallel code. We take advantage of C++11 attributes as a mechanism to introduce annotations and enrich semantic information on valid source code. We also present a methodology for performing transformation of source code that allows to target multiple parallel programming models by using a rule based mechanism. In the paper we presented a set of rules for generating FastFlow parallel code. The generated code has been compared with OpenMP equivalent parallel code showing that the obtained speedups are comparable to those obtained by manual parallelization.

Currently, the REPARA approach requires programmer intervention for initial code annotation. As future work we intend to extend the approach in order to automatically generate REPARA attributes based code with minimal user assistance. Furthermore, we will extend the REPARA attributes for targeting heterogeneous platforms equipped with GPUs, FPGAs and DSPs.

Acknowledgment This work has been partially funded by the European project REPARA (ICT-609666).

References

[1] Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: an efficient multithreaded runtime system. SIGPLAN Not. 30(8), 207–216 (Aug 1995)
[2] Chapman, B., Jost, G., Pas, R.v.d.: Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation). The MIT Press (2007)
[3] Danelutto, M., Torquati, M.: Structured parallel programming with "core" fastflow. In: Zsók, V., Horváth, Z., Csató, L. (eds.) Central European Functional Programming School, Lecture Notes in Computer Science, vol. 8606, pp. 29–75. Springer International Publishing (2015)
[4] Intel® TBB website (2015), http://threadingbuildingblocks.org
[5] ISO/IEC: Information technology – Programming languages – C++. International Standard ISO/IEC 14882:20111, ISO/IEC, Geneva, Switzerland (Aug 2011)
[6] Mattson, T., Sanders, B., Massingill, B.: Patterns for Parallel Programming. Addison-Wesley Professional, first edn. (2004)
[7] Odersky, M., Spoon, L., Venners, B.: Programming in Scala: A Comprehensive Step-by-Step Guide, 2Nd Edition. Artima Incorporation, USA, 2nd edn. (2011)
[8] REPARA website (2015), http://repara-project.eu/
[9] The X10 Programming Language website (2014), http://x10-lang.org/