Preserving high-level semantics of parallel programming annotations through the compilation flow of optimizing compilers

Antoniu Pop, Albert Cohen

Preserving high-level semantics of parallel programming languages through the compilation flow of optimizing compilers

Antoniu Pop [1] and Albert Cohen [2]

[1] Centre de Recherche en Informatique, MINES ParisTech, France
[2] INRIA Saclay and LRI, Paris-Sud 11 University, France

Abstract. This paper presents a technique for representing the high level semantics of parallel programming languages in the intermediate representation of optimizing compilers. The semantics of these languages does not fit well in the intermediate representation of classical optimizing compilers, designed for single-threaded applications, and is usually lowered to threaded code with opaque concurrency bindings through source-to-source compilation or a front-end compiler pass. The semantical properties of the high-level parallel language are obfuscated at a very early stage of the compilation flow. This is detrimental to the effectiveness of downstream optimizations. We define the properties we introduce in this representation and prove that they are preserved by existing optimization passes. We characterize the optimizations that are enabled or interfere with this representation and evaluate the impact of the serial optimizations enabled by this technique for concurrent programs, using a prototype implemented in a branch of GCC 4.6. While we focus on the OpenMP language as a running example, we also analyze how our semantical abstraction can serve the unification of the analyses and optimizations for a variety of parallel programming languages.

1 Introduction

Programming applications for multi-core systems increasingly relies on higher-level languages, designed to express concurrency, dependence, synchronization and locality. This information is necessary for efficient and portable parallelization and appears in the form of annotations to conventional programming languages, like pragmas for C or C++. The enhanced semantics of these languages does not fit well in the intermediate representation of classical optimizing compilers, designed for single-threaded applications, and therefore requires either source-to-source compilation to a sequential programming language or a front-end to an existing compiler with an early expansion pass that lowers the language to the sequential intermediate representation. In both cases the loss of the additional information provided in such languages, and the obfuscation of the underlying code, occurs at a very early stage of the compilation flow, forcing a tradeoff between exploiting the available parallelism and classical compiler optimizations. With the ever increasing number of cores, this tradeoff leans towards

concurrency and early expansion, even though it also means losing all hope for optimizing the structure and the granularity of the parallelism, for statically scheduling the computation or for performing global optimizations.

This paper presents a solution whereby the existing intermediate representation is transparently used to represent additional semantics in a way that allows classical analyzes and optimizations to be performed, while also enabling to optimize the expressed parallelism and allowing to check the annotations' validity through static analysis. This solution does not require any adjustment to existing compiler passes. Our work stems from the intuition that early expansion of parallel constructs is a waste of information and results in strong code obfuscation that hampers subsequent attempts at code analysis and optimization. The expansion should therefore be delayed. We present the general concepts, their application to the OpenMP language and the GCC compiler, and an early implementation in GCC 4.5. We show that this approach is both sufficiently flexible to easily integrate new language extensions, which we illustrate on an OpenMP extension for streaming, and generic enough to be compatible with different and domain specific languages, like HMPP.

2 Motivation

The early expansion of user annotations (E.g., OpenMP pragmas) to runtime calls, with the associated code transformations, outlining, opaque marshaling of data and use of function pointers, is a process whereby concurrency is gained, at an early compilation stage, at the cost of the loss of the initial high-level information and obfuscation of the underlying code.

The annotations provide a wealth of precise information[3] about data dependences, control flow, data sharing and synchronization requirements, that can enable more optimizations than just the originally intended parallelization.

The common approach for the compilation of parallel programming annotations is to directly translate them into calls to the runtime system at a very early stage. For example, in the GCC compiler, this happens right after parsing the source code. This means that all the high-level information provided by the programmer is lost and the compiler will have to cope with the resulting code obfuscation and loss of precise information. Our approach is to further abstract the semantics of the user annotations and bring this information into the compiler's intermediate representation using the technique presented in Section 3. The semantical information is preserved, and when possible used or even refined, until the end of the code optimization passes, where it is finally translated to the intended runtime calls in a late expansion pass.

Let us consider the example on Figure 1 where a simple *omp parallel for* loop with a static schedule is expanded. Despite the fact that we chose one of the least disruptive expansions[4], the resulting code does not look quite as appealing

[3] We obviously assume correctness.

[4] If for example the schedule of the loop had been chosen to be dynamic, the resulting expanded code would be much harder to analyze.

```
int main () {
  int *a = ... ;

#pragma omp parallel for shared (a) \
          schedule (static)
    for (i = 0; i < N; ++i)
      {
        a[i] = foo (...);
      }

  for (j = 0; j < N; ++j)
    ... = a[j]
}
```

```
void main_omp_fn_0 (struct omp_data_s * omp_data_i) {
  n_th = omp_get_num_threads();
  th_id = omp_get_thread_num();
  // compute lower and upper bounds from n_th and th_id

  for (i = lower; i < upper; ++i) {
    omp_data_i->a[i] = foo (...);
  }
}

int main () {
  int *a = ... ;

  omp_data_o.a = a;
  GOMP_parallel_start (main_omp_fn_0, &omp_data_o, 0);
  main_omp_fn_0 (&omp_data_o);
  GOMP_parallel_end ();
  a = omp_data_o.a;

  for (j = 0; j < N; ++j)
    ... = a[j]
}
```

Fig. 1. The early expansion of a simple OpenMP example (left) results in information loss and code obfuscation (right).

for most analysis and optimization passes. If the original loop could have been unrolled or vectorized, it is now very unlikely it would still be.

To make matters worse, the resulting code is not only harder to analyze and optimize, but it also lost the information provided by the user through the annotations and we lost the capability of optimizing the parallelization itself. In the original version, as the loop is declared to be parallel with a shared data structure a, we know that the right-hand-side of the assignment $a[i] = ...$ is not partaking in any loop-carried dependences or that calls to the function foo have no ordering restrictions and can happen concurrently. In the expanded version, however, that information is lost and must be found through analyses that may, and quite likely will, fail. Among other possibilities, the loop annotated as parallel may have been fused with the second loop, but that is no longer an option once expansion has taken place.

Figure 2 illustrates the compilation flow of three parallel programming languages that are representative of this type of languages. OpenMP [5], StarSs [2] and HMPP [4] each in their own way suffer from this issue. StarSs and HMPP rely on source-to-source compilers as a first step. The ad hoc compiler they rely on is capable of generating optimized parallel code, either directly expanded to calls to the runtime system or translated into another high-level parallel programming language like OpenMP. From that point on, their compilation flow either goes through an early expansion pass that generates parallelized code and issues calls to the runtime along with OpenMP, or as is the case for HMPP, the code is parsed and directly represented in the compiler's intermediate representation. At that point, most of the potential for further optimization is lost.

In order to preserve the high-level semantics of user annotations and to avoid clobbering important optimizations or analyses, we replace the early expansion

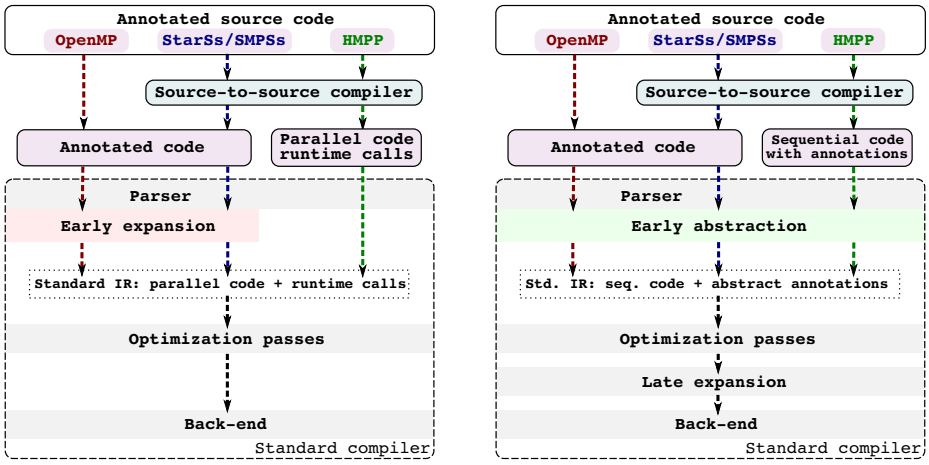

Fig. 2. Compilation flow of high-level parallel-programming languages, current situation (left) and our objective (right).

of user annotations by an early abstraction pass. This pass extracts the semantics of the annotations and inserts it into the compiler's original intermediate representation, using constructs that preserve the information in a state that is usable by analysis and optimization passes and that can ultimately be expanded to parallel code and runtime calls at the end of the compilation flow.

We believe that even languages like HMPP, with a dedicated optimizing compiler, can benefit from our approach as the source-to-source compiler is generally intended and specialized to perform the domain-specific optimizations corresponding to the original source language. This compiler is unlikely to benefit from as large a base of optimizations as, for example, GCC. Extending our framework to such a language should not be overly complicated, but getting access to the ad hoc optimizations implemented in its compiler would require writing a new code generation backend for the source-to-source compiler.

We attempt to address the following issues:

1. High-level parallel programming languages, in particular OpenMP, are poorly optimized by current compilers, even for simple and crucial sequential scalar optimizations.
2. Opportunities for optimizing the exploitation of parallelism are lost[5] (e.g., possibility to compute optimized static schedules, verification ...).
3. User information on concurrency, dataflow and synchronization requirements is wasted. It can be used for more than only parallelization.

In Section 7 we present a semantic abstraction pass that we substitute to the early expansion pass. This *early abstraction* pass extracts important information from the user annotations and stores this information in the compiler's

[5] This is more an issue for OpenMP than for StarSs and HMPP as they have optimizing compilers.

intermediate representation using the technique we discuss in Section 3. In the subsequent Section **??**, we survey some important applications where the information we add to the intermediate representation, as well as the way this information is represented, are used to address the three aforementioned issues. Finally, in Section 8, we give a short list of some of the exciting areas we have planned to explore before concluding.

3 Intermediate representation

The semantics of user-level annotations is generally defined with a direct correspondence to specific parallelization techniques or to specific runtime calls. This makes them well-suited for early expansion as they are self-contained and require no static analysis or verification. A direct translation, or expansion, can be performed at the earliest stages of the compilation flow, which is a convenient way to avoid the interactions with the optimization passes of compilers.

A common constraint in extending the intermediate representation of a compiler is that it requires modifying most compiler passes, if only to keep the new information consistent after code transformations. Instead of modify the representation, we circumvent this issue by making use of the existing infrastructure.

Our approach is based on the use of calls to factitious builtin functions that carry the high-level semantics. These builtins are completely opaque to existing compiler analyses, which ensures their persistence. In order to define the scope to which these semantics apply, we use the return value of our builtins to predicate the targeted blocks of code. As we will further detail in Section 5, this will play an instrumental role in the preservation of the parallel semantics even through aggressive optimization passes.

```
int *X;

void bar () {

  for (int i = 0; i < ...; ++i) {
#pragma omp task shared (X) firstprivate (i)
    {
      X[i] = foo (i);
    }
  }
#pragma omp barrier
  // use X;
}
```

```
void bar () {

  for (int i = 0; i < ...; ++i) {
    if (__builtin_omp_task ()
        && __builtin_shared (X)
        && __builtin_firstprivate (i)) {

      X[i] = foo (i);
    }
  }
  __builtin_omp_barrier ();
  // use X;
}
```

Fig. 3. Using builtin functions to represent the OpenMP semantics.

The example, on Figure 3, illustrates the use of this intermediate representation for a simple OpenMP program. Each parallel construct or clause is directly translated to a builtin call. When the original directive or clause had parameters, they are passed as parameters to their builtin counterpart.

An important distinction must be made between the high-level semantics of the parallel program that is carried by this intermediate representation and the semantics of the representation itself. All the compiler analysis and optimization passes that are unconcerned by the parallel semantics will only perceive such a representation as builtin function calls and conditional blocks. As we will discuss more in detail later, this intermediate representation needs further tuning to ensure the interaction between our representation and such unconcerned compiler passes does neither break the optimizations' and analyses' validity, nor clobber the high-level information. This separation is key to enabling the preservation of the parallel program's semantics without adjusting the optimization passes.

As any function, builtins can be tagged, inside the compiler, either as `pure` functions, which means they are allowed no side-efects and can only read from memory and the input parameters[6], or as `const` functions that can only read the input parameters. This distinction impacts the way the compiler can optimize the code around these builtin function calls. As builtins are opaque, the compiler has no knowledge of the behaviour or side-effects that result from calling them and can only rely on the qualifiers that we provide for each one of them.

Aside from adjusting the qualifiers of the builtin function, we can also change the way parameters are passed to the builtin function in order to adapt the intermediate representation's semantics to our needs. Table 1 sums up the options available in classical intermediate representations that we can use to tune the low-level semantics of our representation.

Function qualifier	Low-level semantics (what the compiler can assume)
default	Call may read or write to any part of global memory and modify its input parameters.
pure	Call may only read from global memory and from its input parameters.
const	Call may only read from its input parameters.
Parameter passing style	
by value or const-qualified	The function can only read the input parameter.
by pointer/reference	The function may read or write to the input parameters.

Table 1. Semantics of the representation.

One of the imperative requirements to make our representation robust, despite not requiring to modify optimization passes, is that it naturally prevents any transformation that would invalidate the semantics of the annotations.

Many compiler passes have the potential to break the semantics if they are to perform without any constraint. However, the representation implicitly introduces a few constraints that we believe to be sufficient. The conditional expres-

[6] This is a somewhat more lax definition than the classical functional programming definition that requires such a function's return value to depend only on the input parameters.

sions it introduces, relying on opaque builtin function calls, ensure the integrity of the blocks of code they are attached to.

4 Case study: the OpenMP language

In this section we analyze the semantics of the parallel programming constructs and clauses of the OpenMP language and we describe how we translate them in the intermediate representation of the GCC compiler. These constructs are essentially a way of providing the compiler with information that cannot be expressed in the underlying sequential programming language and that the compiler may not be able to extract through static analyses. They supplement the sequential semantics of Fortran, C or C++ with information about the data-flow, with control-flow restrictions that stem from the lack of precision on the dynamic data-flow and with hints on the best strategies to exploit the concurrency resulting from the absence of conflicts.

The core elements of the OpenMP language are directives for thread creation (`parallel, task`), worksharing directives (`for, single, sections, workshare`), synchronization directives (`barrier, atomic, taskwait`), data sharing clauses (`shared, private, firstprivate, lastprivate`) and tuning clauses (`schedule, num_threads`). These constructs can be classified in two broad semantic categories based on the type of information they provide. Data sharing clauses provide data-flow information, while all the other constructs focus on describing the control-flow restrictions necessary to properly orchestrate the program execution in order to avoid concurrency conflicts (data races).

Translating these constructs in our builtin-based representation requires more than replacing a compiler directive with a function call and a conditional statement. We propose the translation scheme presented on Table 2.

OpenMP construct	Builtin equivalent	Function qualifier	Argument passing style
Thread creation directives			
#pragma omp parallel	__builtin_omp_parallel ()	pure	—
#pragma omp task	__builtin_omp_task ()	pure	—
Worksharing directives			
#pragma omp for	__builtin_omp_for ()	—	—
#pragma omp sections	__builtin_omp_sections ()	—	—
#pragma omp single	__builtin_omp_single ()	—	—
#pragma omp workshare	__builtin_omp_workshare ()	—	—
Synchronization directives			
#pragma omp barrier	__builtin_omp_barrier ()	—	—
#pragma omp taskwait	__builtin_omp_taskwait ()	—	—
#pragma omp atomic	None[7]	—	—
Data sharing clauses			
shared (x)	__builtin_shared (&x)	—	by pointer
private (x)	__builtin_private (x)	const	by value
firstprivate (x)	__builtin_firstprivate (x)	const	by value
lastprivate (x)	__builtin_lastprivate (&x)	—	by pointer

Table 2. .

[AP: Table still incomplete]

[AP: The parallel for directive needs special handling. The loop body also needs to be marked to prevent optimizations that would not keep its data-parallel property.]

The argument passing style allows selectively adjusting the compiler's understanding of the conditional block with respect to the parameters alone, while the function qualifier has a much broader scope and is a blunt method for inhibiting some optimizations.

The correspondence between the two semantic categories of OpenMP constructs and these two means of controlling static analysis is all too obvious. The constructs providing precise data-flow information only require restrictions on the builtins parameters, the functions themselves being const-qualified to let the analysis passes know there is no other side-effect. The remaining constructs merely allow to cope with the impossibility of providing precise enough data-flow information, and they will be adjusted using the function qualifiers.

[AP: Discuss the semantics of the representation for some interesting cases]

5 Interaction with analysis and optimization passes

Despite the diversity of the compiler analysis and optimization passes in modern optimizing compilers, like GCC, they all share a common constraint: they must conservatively preserve data flow dependences. *[AP: Cite Felleisen PDG (86)]*

A complete study of all optimization passes is impossible within the limits of this paper, but the principles underlying the semantic preservation of our representation are similar in all cases. We will detail the analysis of two common optimization passes, conditional constant propagation and partial redundancy elimination, to illustrate this technique. We will also briefly discuss the interaction of our representation with other relevant optimization passes.

[AP: Mention special cases, e.g., graphite.]

6 Evaluation

We evaluate the impact of our approach, both positive and negative, using the OpenMP implementation of the GCC compiler. We compare the performance of OpenMP applications either compiled with the traditional compilation flow where the parallel constructs are expanded in the front-end, or with our approach where the front-end translates the parallel constructs to our builtin-based representation which is only expanded late in the middle-end.

7 Semantic abstraction

The semantics of user-level annotations is generally defined with a direct correspondence to specific parallelization techniques or to specific runtime calls.

[7] The atomic directive should be directly replaced by the appropriate compiler builtin for atomic operations.

Because of this, if instead of the early expansion we only represent the annotations, as they are, in the intermediate representation, the interpretation of their semantics will be necessary for each compiler pass that needs to use the information they carry. Multiple interpretation layers, in optimization passes and then in the late expansion pass, would severely reduce the genericity of this framework and make its extension cumbersome.

The solution we advocate is to replace the early expansion pass by an *early abstraction* pass that extracts the necessary information from user annotations and represents it using a unique set of abstract annotations irrespectively of the original language, lowering the annotations to a language-independent representation, which should provide a unified view of the user information whether it comes from OpenMP, HMPP or StarSs annotations.

The key insight is that the high-level user annotations mostly provide information on data-flow, with also some restrictions on control-flow that stem from the lack of precision on the dynamic data-flow. The concurrency is just a result of the absence of conflicts. We also recognize the importance of the additional information a user provides as hints on the best strategy, like for example which is the scheduling technique likely to yield the best results.

Adapting a new language, or an extension, to this early abstraction pass requires understanding and abstracting the underlying semantics of the annotations, but it should not require any modification in the optimization passes of the compiler. Additional ad hoc semantics for tuning new architectures or accelerators can easily be added in the form of user hints.

Following is the set of required abstract annotations, and a gist of their semantics.

Data-flow annotations.

- **use**: the variable or memory area is read within the associated block.
- **def**: the variable or memory area is written.
- **may-use**: the variable or memory area may be read within the associated block.
- **may-def**: the variable or memory area may be written.
- **safe-ref**: the variable is used or defined, but the user guarantees that all potential conflicts are handled, e.g., with manual synchronization.
- **reduction**: the associated variable is part of a reduction.

Control-flow annotations.

- **SESE**: the associated block of code is a Single-Entry Single-Exit region. There is no branching in or out and exceptions are caught within the region.
- **single**: the associated block can only be executed on one thread.
- **barrier**: either an explicit barrier or when a barrier is implied at the end of a block.
- **synchronization point**: point-to-point synchronization.
- **memory barrier**: a memory flush is required at this point.

User hints. Many of the decisions involved in tuning the parallel code generation and the execution are hard to decide from static analysis alone. We store as *hints* all the information provided by the programmer. If the optimization passes can find provably better choices, then these hints can be ignored, otherwise they should take precedence.

- **parallel**: this hint may be important for loops, because even if static analysis can recognize the loop is parallel, the profitability of the parallel execution may not be obvious. If the programmer annotates a loop as parallel, it should not be overlooked.
- **schedule**: the choice of the schedule for a parallel loop.
- **num_threads**: number of threads available.
- More generally, any ad hoc information can be stored as a hint. In particular, in case the late expansion pass is too difficult to perform using the abstract annotations alone, it would be trivial to keep the whole set of original annotations in this form. As we will see in Section **??**, this is the easy way to solve the problem of enabling classical sequential optimizations for such languages as OpenMP.

These abstract annotations provide readily usable information to the optimization passes. They can also be refined through static analysis as, for example, OpenMP sharing clauses will generally only provide *may-def/may-use* information which can be promoted to *def/use*.

Depending on the compiler pass, annotated blocks of code can be either seen as black boxes, that have well-specified memory effects and behaviour, or they may need to be perfect white boxes to allow unrelated optimizations to be transparently applied. The representation of these annotations needs to allow access to the code, yet restrict optimizations that would break the semantics of the optimizations.

Default clauses. In languages that have default clauses, or default specified behaviour, all defaults must be made explicit by the early expansion. This is part of the interpretation of the language's semantics and keeping any information implicit would hamper the genericity of the approach. The abstract annotations should be self-contained.

In particular, the OpenMP default sharing or a *default* clause allows the programmer to leave some of the sharing clauses implicit. We convert all implicit clauses to explicit ones during the early abstraction pass, which allows to decouple the intermediate representation from the OpenMP-specific semantics of the default sharing.

Example: abstract semantics for OpenMP. Without attempting to provide a full characterization of the OpenMP semantics, we present on Figure 4 a subset of the abstract semantics of the language.

Adapting this framework for an OpenMP extension for streaming [1, 3], consisting in two additional clauses for task constructs, would require also adding

	OpenMP annotation	Abstract annotations counterpart
Main directives	parallel	SESE & barrier
	single	SESE & single & barrier
	task	SESE
	sections	SESE & barrier
	section	SESE & single
	for	parallel hint & barrier
Synchronization directives	master	master thread hint & single
	atomic {expr}	lower to corresponding atomic builtin operation
	barrier	barrier
	taskwait	synchronization point
	flush	memory barrier
Sharing clauses	shared (X)	safe-ref (X) & may-use (X) & may-def (X)
	firstprivate (X)	use (X)
	lastprivate (X)	def (X)
	private (X)	rename the variable X_p
	threadprivate (X)	rename the variable X_tp
	reduction	reduction(X)
	copyin (X)	use (X) & def (X_tp)
	copyprivate (X)	barrier & use (X) & def (X_p)
Tuning clauses	num_treads	num_threads hint
	schedule	schedule hint
	collapse	—
	ordered	single & static schedule
	nowait	remove the implicit barrier from the directive

Fig. 4. OpenMP semantics.

the same two data-flow annotations. This extension defines an `input` and an `output` clauses for tasks, which can be abstracted to a `use` and a `def` annotations in the simple, scalar version of the extension.

8 Roadmap for future work

In order to experimentally validate our approach and evaluate the impact these techniques have on real applications, we envisage the following roadmap:

- Evaluate the additional code coverage that can be achieved in the polyhedral representation by using the additional semantics of OpenMP annotations in the programs of the OpenMP Benchmark Suite.
- Consider streaming OpenMP extensions carrying explicit dependence information, to enhance the accuracy of data dependence analyses.
- Further evaluate the performance improvement this added coverage has on both the late-expanded version and on the sequential version.
- Evaluate more precisely and more extensively the impact of missed optimization opportunities on the OpenMP Benchmark Suite, by comparing the performance achieved using the original OpenMP code with the classical early expansion to the performance achieved using late expansion.
- Compare the performance results of early expansion to the results of both unoptimized late expansion and optimized late expansion with specific concurrency optimization.

9 Conclusion

We presented an alternative approach to the classical compilation flow of high-level annotation-based parallel programming languages. This alternative solution enables sequential optimizations of parallel codes, in particular it allows OpenMP programs to benefit from many optimizations that until now were out of reach. Further uses, of the intermediate representation we presented include the extension of the scope of polyhedral representation and optimization as well as static verification of user annotations.

Acknowledgements. This work was partly funded by the European FP7 project TERAFLUX id. 249013, `http://www.teraflux.eu`.

References

1. P. M. Carpenter, D. Ródenas, X. Martorell, A. Ramírez, and E. Ayguadé. A streaming machine description and programming model. In *SAMOS*, pages 107–116, 2007.
2. J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta. Hierarchical Task-Based Programming With StarSs. *Int. J. High Perform. Comput. Appl.*, 23(3):284–299, 2009.
3. A. Pop and A. Cohen. A Stream-Computing Extension to OpenMP. Technical report, MINES ParisTech, CRI - Centre de Recherche en Informatique, Mathématiques et Systèmes, 35 rue St Honoré 77305 Fontainebleau-Cedex, France, Jan. 2009.
4. S. B. R. Dolbeau and F. Bodin. Hmpp: A hybrid multi-core parallel programming environment. In *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU 2007)*, 2007.
5. The OpenMP Architecture Review Board. OpenMP Application Program Interface. `http://www.openmp.org/mp-documents/spec30.pdf`.