

# Skeleton driven transformations for an OpenMP compiler

J. Balart, A. Duran, M. González, X. Martorell, E. Ayguadé and J. Labarta

## Abstract

In this paper we present a technique based on code templates, oriented to source to source code transformations for OpenMP parallelization. Our goal is to provide an OpenMP compilation infrastructure that includes a reconfigurable code generation phase, targetting different OpenMP runtime systems or explore different translation strategies for OpenMP constructs. We describe the main OpenMP transformations needed in any OpenMP compiler as well as the characteristics in the code generation phase dependant on the target runtime system. Our implementation has been done in the Open64 compiler infrastructure. We have added a new compiler phase, which takes a file containing WHIRL intermediate representation generated from source code, and applies transformations at specific points, indicated by OpenMP directives. Each OpenMP directive has a source code template associated.

## 1 Introduction

Shared-memory parallel architectures are becoming affordable as common platforms for the development of applications demanding huge amount of computational power. Users of such architectures require simple but powerful programming models to develop and to tune their parallel applications with reasonable effort. These programming models are usually offered as library implementations or extensions to sequential languages that allow the programmer to express the available parallelism in the application. Language extensions are defined by means of directives and language constructs that are translated by the compiler to library calls. These libraries offer mechanisms to create threads, distribute work among them and synchronize their activity. This paper focusses on this transformation process in the framework of OpenMP [6].

OpenMP has become the standard for parallel programming in shared memory parallel architectures. The language is composed by a set of directives or pragmas that are included in the specification of high-level languages such as C or Fortran. The directives allow the programmer to express the available parallelism in the program, coded in the native language (C, C++ or Fortran) where OpenMP is included.

The specification of OpenMP is open to extensions discussed in the OpenMP Architecture Review Board. Application developers as well as researchers in compiler optimizations can submit their proposals for possible extensions of the specification. This process could be more effective if there were open source compiler and runtime infrastructures for OpenMP simple to modify and extend. There have been several recent attempts, as for instance NanosCompiler [?] for Fortran77, Omni [9] for Fortran77 and C and OdinMP [5] for C/C++. All of them are source-to-source translators that transform the code into an equivalent version with calls to the associated runtime system.

However, modifying the compiler to include new language features, to explore alternative code generation strategies or to target different runtime systems is not an easy task that requires to go into the deep internals of the compiler.

In this paper we present an approach based on the modification of an already existing compiling platform for C or Fortran. The compiler front-end has to be updated with an OpenMP parser in order to include the parsing of the new constructs. The existing definition of the AST structures need to be modified as well, to allow the memory representation of the OpenMP directives. Once the compiler is able to represent the OpenMP constructs and link them to the original AST, it is needed to implement a compiler phase which performs the required transformations according to the semantics of the directives. In that phase, the compiler activity mainly concerns the following main issues: parallelism creation/completion, work distribution during the parallelism execution and synchronization. These issues are typically supported by a thread library which contains the appropriate services to create threads, to specify how work is distributed among them and how they synchronize. Therefore, the compiler mainly has to the introduction of call statements to those runtime services, extract code and encapsulate it in a procedure as well as gather references to symbols and create new ones. All these operations are available in most compiler infrastructures.

It is important to mention that all the required transformations can be performed outside the compiler back-end. The compiler is acting like a preprocessor on the source code. The directives are processed and the original source code is transformed according to them. All the transformations are done over the AST structures. It is quite uncommon that the OpenMP directives cause any transformation performed in the compiler back-end, where the compiler activity is related to the assembler code generation and optimization, totally dependent on the computer architecture specific parameters. This is due to the fact that the main issues on the parallelism execution (thread creation/completion, work distribution and synchronization) are supported by the thread library. It is the thread library implementation what releases the compiler of being aware about specific hardware support for the parallelism execution.

In summary, there are two main compiler parts where developers have to put efforts in order to build an OpenMP compiler. The parser has to be updated to recognize the OpenMP constructs. This just requires the modification of the native language grammar. The implementation of all the OpenMP transformations are usually done in the compiler front-end where typically involves modifying the AST structure, which normally for any minor transformation involves a dozen of lines of code, making this process slow and cumbersome. Our proposal tries to reduce the time spent in the developing of new transformation inside the compiler to accelerate the research process. For achieving that objective we introduce the use of templates in the final language of the transformation so modifications to it can be quickly done.

The paper structure is as follows: Section 2 describes the OpenMP compiler and runtime coordination. Section 3 describes the proposal in this paper based on template specification for the compiler transformations. Section 4 describes some related work. Section 5 concludes the paper and outlines future work.

## 2 Compiler and Runtime Coordination

In this section we describe the common functionalities available in most runtime systems that support the OpenMP execution model, and the compiler transformations needed in the source-to-source translation process. Five main issues have to be faced: how threads are created, how threads execute code in the parallel region, how threads access data according to the data scoping rules, how threads determine which portions of work they have to execute, and which thread synchronizations have to occur at runtime. Next we briefly describe each of these points:

- **Executing code in the parallel region:** Typically this is achieved by the compiler through the definition of a procedure that encapsulates the parallel code. This causes the definition of the procedure name symbol as a pointer to the code. Threads access to the parallel code through this pointer. This corresponds to one of the OpenMP transformations to be implemented in the compiler.
- **Thread creation:** Usually there is a specific runtime service responsible for creating the necessary structures for the thread representation and execution. That is, supply a stack and implement the appropriate mechanism for firing the thread execution. It is reasonable to think that the subroutine symbol allocated by the compiler (see previous point) appears as argument of the thread creation runtime service. Variables and data structures can be passed as parameters of the subroutine mentioned in the previous point.
- **Work distribution:** This is achieved through some compiler transformations on the parallel code, after the code has been encapsulated. For each worksharing construct that delimits how the parallelism has to be scheduled, the compiler injects the appropriate runtime calls. These calls force that each executing thread queries the runtime about which portion of the defined parallelism has to execute.
- **Data Scoping:** The use of the OpenMP clauses `PRIVATE`, `SHARED`, `FIRSTPRIVATE`, `LASTPRIVATE` and `REDUCTION` defines several attributes on the data that determine the its scope. The clauses `PRIVATE`, `FIRSTPRIVATE` and `LASTPRIVATE` make a symbol being privatized and allocated in the thread stack plus some other actions concerning to the initial/final value for the symbol. Regarding the compiler and runtime coordination, these functionalities are supported by the injection of specific runtime calls and the definition of extra arguments in the subroutines encapsulating the parallel code.
- **Thread Synchronizations:** OpenMP considers a few synchronization schemes. They correspond to barrier synchronization, mutual exclusion execution and ordered execution. Depending on the available technology in the runtime system, the compiler is stressed. In case the runtime includes specific services for these synchronization schemes, the compiler activity is limited to the injection of `call` statements to the appropriate runtime services. This is the case that this paper considers.

In this paper we are not going to cover the runtime and compiler support for the thread synchronizations and the data scoping. The paper just describes how we have implemented in the compiler the creation of threads and its termination, and the work distribution schemes among threads.

<pre> program test integer a(100) ... !\$omp parallel !\$omp do schedule(static,4)   do k = 1, 100     a(k) = 0   end do !\$omp end parallel ... end </pre>	<pre> integer a(100) C Runtime declarations integer*8 nth_mask, nth, nth_selfv integer*4 nth_nprocs, nth_p ... nth_nprocs = nthf_cpus_actual () nth_selfv = nthf_self () nth_mask = 010101b CALL nthf_depadd (nth_selfv, nth_nprocs + 1) do nth_p = 0, nth_nprocs - 1   nth=nthf_create_ls_vp(par_reg_01,0,nth_p,nth_selfv, &gt; nth_mask,1,a) end do CALL nthf_block () ... </pre>
a) Example of OMP code.	b) Compiler transformation.

Figure 1: Code for thread spawn and join.

## 2.1 OpenMP Compiler Transformations

In this section we describe all the compiler transformations that have to be implemented to support the OpenMP programming model. Although what here is going to be presented depends on specific characteristics of the NthLib thread library, we think that the compiler transformations are common and usual in any OpenMP compiling platform. Some parts of the transformations are very conditioned to the NthLib runtime implementation. We consider the description of the NthLib internals out of the scope of this paper. For more details on the runtime topics, please see [3, 2]. The reader should see the generated code by the compiler as an example illustrating the AST transformations required in our OpenMP compiling infrastructure.

In order to show the transformation process, we use the the example in figure 1.a. The code defines a parallel region that contains a do statement.

### 2.1.1 Parallel Code Encapsulation

In order to allow the threads to execute the parallel code, the compiler extracts the code enclosed in a parallel region and defines a subroutine where the code is located. This transformation is quite simple and just requires basic operations like code extraction and symbol reference gathering. This last one is necessary to declare the necessary subroutine arguments to define a correct context for the encapsulated code. For the example in 1.a, the compiler generates the subroutine in figure 2. Notice that in this transformation nothing is done for the iteration distribution for the parallel loop, according to the specified scheduling: *STATIC* with chunk equals 4. This is going to be manage in the transformation supporting the work distribution.

### 2.1.2 Thread Spawning and Joining Code

Per each parallel region, the compiler replaces the code enclosed in the parallel region by a block of statements in charge of the thread spawn and join. This code is usually formed by a

```

subroutine par_reg_01(a)
integer*4 a(100)
integer*4 k

do k = 1, 100
  a(k) = 0
end do
end

subroutine par_reg_01(a)
integer*4 a(100)
CALL par_ws_do_01 (a)
end

```

a) Thread function.                      b) Thread function after worksharing code encapsulation.

Figure 2: Parallel code in thread function before/after worksharing code encapsulation.

loop where one thread is created in each iteration. The compiler injects a runtime call to the `nthf_create_ls_vp` service to allow the thread creation at runtime, inside the body of a `do` statement. Before the loop execution, the compiler has injected the appropriate statements for querying the runtime about the available number of threads. This corresponds to the runtime invocation to `nthf_cpus_actual` service. After thread creation, the thread that has spawned the parallelism, has to block and wait until the termination of all created threads. This is accomplished by the injection of another runtime call to service `nthf_block`. Figure 1.b shows the code that the compiler should generate for the parallel execution of the example in the same figure.

### 2.1.3 Work Distribution Code

For each worksharing construct in a parallel region, the compiler produces the same transformation as for the parallel code. The code affected by the worksharing construct is extracted and encapsulated in a new subroutine. The compiler injects a call statement to the new allocated subroutine that replaces the extracted code. Notice that this is done in the subroutine containing all the code in the parallel region, as this transformation is done after the first code encapsulation described in previous section. Independently of the worksharing type, the compiler builds the code structure showed in figure 4. The showed code contains some symbols with prefix `tpl_` that its meaning is explained in the next section. What we want to point out here is the code structure and the runtime calls injected by the compiler. The runtime services `nthf_begin_for` and `nthf_end_for` define and conclude the execution of the parallelized `do` statement. Two nested loops are defined. The outermost one is in charge of the work demand to the runtime system through the runtime service `nthf_next_iters`. The innermost loop is in charge of the execution of the supplied iterations. The loop body for this inner loop is obtained by the extraction of the worksharing code. All worksharing constructs are supported with the code in figure 4, but with several specific values for the arguments in the runtime services. Section 3 describes how these arguments are defined.

## 3 Template Guided Transformations

The proposal in this paper is based on the definition of several templates guiding the compiler transformations described in the previous section. In the implementation of our OpenMP com-

pilers we have used the Open64 compiler [7] as the base compiling platform. Open64 already has an OpenMP parser and is able to represent the OpenMP constructions in its intermediate representation, called *whirl*. We have added a new phase that is executed after the corresponding front-end. The input of this new phase is the *whirl* intermediate representation created by the front-end with the OpenMP directives represented in the AST. When our phase is executed, all the directives are transformed applying the corresponding template, obtaining a new *whirl* representation with all the OpenMP directives replaced by calls to the NthLib thread library. At this point, one could use the Open64 backend over the *whirl* intermediate representation and obtain an executable, or decompile and obtain a new program compilable with any native compiler in any platform supporting the threads library.

### 3.1 Template Definition

Templates used by our compiler are written in the same language as the application being compiled. We have not needed to use language extensions for template specification. We have only three simple rules to write templates. The first rule describes how to specify a variable part in a template, the second one how to specify parts of code only necessary under certain conditions and the last one how to specify parts of code that have to be processed a variable number of times.

- **Rule 1:** All variables in a template whose names start with a certain prefix, `tpl_` in our implementation, will be template variable parts that will be replaced by an expression. The transformation to do over this variables is coded in the compiler internals. The compiler includes a correspondance between template variable names and the corresponding transformation.
- **Rule 2:** When the logical expression in a conditional construction (if statement) is formed by a template variable, only the code that would be executed after evaluating the condition will be inserted in the template instantiation. The template expansion module has to evaluate the condition depending on the code being compiled.
- **Rule 3:** When the logical expression of a loop construction (dowhile statement) is formed by a template variable, the code inside the construction is inserted *n* times and the original loop construction is extracted from the template instantiation. The template expansion module has to evaluate the condition and the value of *n* depending on the code being compiled.

Due to the impossibility of some languages for a variable to appear alone in the code, that is not inside an expression or a language construction, an extension to rule 1 is necessary. In these cases we declare external subroutines with names according to rule 1 and insert calls to them in the templates. These subroutine calls will be replaced by the expansion of the template reference by the subroutine name. In order to support the previously defined rules, our implementation has required a template expansion module that performs the following template operations:

- **Operation 1:** replace a template variable by another variable.
- **Operation 2:** replace a template variable by a integer or real constant.

- Operation 3: replace a template variable by a portion of AST.
- Operation 4: replace a template variable by a set of real parameters.
- Operation 5: replace a template variable by a set of formal parameters.
- Operation 6: create a set of local variables in a template.
- Operation 7: replicate a part of code in a template.
- Operation 8: extract a part of code in a template.
- Operation 9: change the name of a template variable.

The following sections describe how we support the compiler transformations through templates.

### 3.2 PARALLEL template

When translating a parallel directive our compiler uses two templates, one for the thread creation code and another one containing the subroutine that will be execute in parallel. The first thing the compiler does when compiling a parallel directive is to create a subroutine containing the code enclosed in the directive. This subroutine will be the one threads will execute. The subroutine is generated according to the template shown in figure 3.b, what means a new instance of this subroutine is generated and the three template variables contained in the code are spawned as indicated:

- `tpl_template_subroutine`: This transformation follows the application of rule 1 and operation 9. The expansion of the symbol is very simple. One could modify the name of this symbol by simply adding a counter at the end to avoid symbol duplications. We expand symbols whose names start by `tpl_template` changing their names by the concatenation of the subroutine name where the directive appeared, a static word depending on the directive being compiled, encapsulation in this case, and the line where the directive appeared.
- `tpl_body`: This transformation follows the application of rule 1 and operation 3. The expansion associated to symbols whose names start by `tpl_body` is very simple too. In this case our expansion module receives the AST associated to the code enclosed inside the directive being compiled and replaces the call to external subroutine `tpl_body` by a copy of this AST.
- `tpl_formal_parms`: This transformation follows the application of rule 1 and operation 5. To be able to expand symbols whose names start by `tpl_formal_parms` is necessary to walk through the previous AST and find all the variables referenced in it. Then `tpl_formal_parms` is replaced by a formal parameter for each of the symbols referenced.

The second template we use in the compilation of a parallel directive is showed in figure 3.a. We name this template threads creation template. Only four modifications are done during the expansion of this template:

```

subroutine tpl_template_parallel (tpl_formal_parms)
...
nth_nprocs = nthf_cpus_actual ()
nth_selfv = nthf_self ()
nth_cpuv = nthf_whoami ()
mask = tpl_nth_mask
call nthf_depadd (nth_selfv, nth_nprocs + 1)
do nth_p = 0, nth_nprocs - 1
nth=nthf_create_ls_vp(tpl_p_thread_function,0,nth_p,
> nth_selfv,mask,tpl_num_parms,tpl_real_parms)
end do
call nthf_block ()
return
end subroutine

```

a) Template for thread creation code.

```

subroutine tpl_template_subroutine(tpl_formal_parms)
integer*4 tpl_formal_parms
external tpl_body
call tpl_body
return
end subroutine

```

b) Template for thread function.

Figure 3: Template for thread creation code and thread function.

- `tpl_template_parallel`: This template variable is expanded as in `tpl_template_subroutine` in previous template.
- `tpl_formal_parms`: This template variable is expanded as in previous template.
- `tpl_real_parms`: This transformation follows the application of rule 1 and operation 4. The template variable is replaced by a set of real parameters, one for each of the parameters that have appeared in the expansion of symbol `tpl_formal_parms`.
- `tpl_p_thread_function`: This transformation follows the application of rule 1 and operation 9. The template variable is replaced by the symbol associated to the subroutine created when we have applied the template `tpl_template_subroutine`.

Finally, a last and implicit transformation is done when this template is used. The parallel directive and the code enclosed in it are replaced by a call to the instantiation of this template. This corresponds to operation 8.

### 3.3 Worksharing Templates

In this section we present the templates guiding the compiler transformations for the worksharing constructs. The three worksharing constructs: DO, SECTIONS and SINGLE are treated in a similar way, so we only expose the mechanisms for the DO worksharing and then complement for the SECTIONS and SINGLE worksharings.

```

subroutine tpl_template_do(tpl_formal_parms)
...
call nthf_begin_for (tpl_low_bound,tpl_upper_bound,tpl_step,
> tpl_schedule_type,tpl_chunk)
do while (.true. .eqv. nthf_next_iters (nth_start,nth_end,nth_last))
  do tpl_i_ = nth_start, nth_end, tpl_step
    call tpl_body ()
  end do
end do
call nthf_end_for (tpl_barrier)
return
end

```

Figure 4: Template for DO worksharing construct.

Figure 4 shows the template used in the compilation of a work-sharing DO directive. In this template we find the following variable expansions:

- `tpl_template_do`: This template variable is treated as `tpl_template_subroutine` in the PARALLEL template.
- `tpl_formal_parms` and `tpl_body`: are replaced as explained in the PARALLEL template.
- `tpl_low_bound`, `tpl_upper_bound` and `tpl_step`: are replaced by the expressions low bound, upper bound and step of the do statement enclosed by the DO directive. This transformation corresponds to apply rule 1 operation 3.
- `tpl_i`: This template variable is replaced by the variable index of the do statement. The compiler applies rule 1 operation 1.
- `tpl_schedule_type`: This template variable is replaced by an integer constant coding the scheduling to be applied. This corresponds to rule 1 and operation 2.
- `tpl_chunk`: The compiler replaces the template variable by the chunk expression in the DO directive. This corresponds to rule 1 and operation 3.
- `tpl_barrier`: is replaced by a 0 or a 1 depending on if the NOWAIT clause appeared in the end do directive. This corresponds to rule 1 operation 2.

Again, an implicit transformation is done when using this template: the DO directive and the code enclosed in it are replaced by a call statement to the instantiation of this template.

The SECTIONS and SINGLE worksharing constructs can be handled by templates as the DO construct is. The main differences correspond to how the template variables in the arguments for the runtime service `nthf_begin_for` are expanded. For the SECTIONS construct, the compiler defines a loop with as many iterations as sections appear in the construct: one iteration per SECTION. The `tpl_body` variable is expanded with as many `if` statements as sections, all of them evaluating a condition over the `tpl_i` variable. A DYNAMIC scheduling is applied. The SINGLE is treated as a SECTIONS construct with just one section of code.

```

subroutine tpl_template_parallel (tpl_formal_parms)
...
call nthf_create_OpenMP_thread_team(tpl_p_thread_function,
>                                     tpl_num_parms,tpl_real_parms)
return
end subroutine

```

Figure 5: New template for thread creation and termination.

### 3.4 Changing the compiler transformations

This section wants to emphasize the advantages of having the OpenMP compiler transformations guided through template specifications. In OpenMP research areas, it is quite common to make deep changes to the runtime system supporting the OpenMP programming model (either because of changing the target architecture or because of looking for performance improvements). It is reasonable to think that those changes can affect the compiler transformations. Our proposal makes the compiling platform flexible enough to adapt to the new runtime requirements. This can be easily seen with an example. Suppose the you want to change the emitted code concerning to the thread creation and termination. Suppose the runtime system has been modified and a new runtime service is available performing all the operations needed for the thread creation and termination. The new template substituting the one in figure 3.a could be the one showed in figure 5.

Notice that the compiler transformation is simply implemented by modifying the template. It is not necessary to recompile any compiler module.

## 4 Related Work

This section makes a brief overview over most recent efforts in building OpenMP compiling platforms for OpenMP research. Past OpenMP projects like Nanos [4], Intone [1] and the ongoing POP project [8], suffered from a compiler infrastructure being not able to adapt to changes in the runtime system, requiring new compiler support. The recent ORC project [11] includes some OpenMP support (parsing of OpenMP directives and the associated transformations), but it is linked to a particular runtime system based on `pthread`s [12]. Thus it suffers from being not enough flexible.

The approach taken in the `rose` system [10] has to be mentioned due to the similarities to our proposal. Although the `rose` compiler infrastructure has the ability to specify the compiler transformations with variable parameters, the implementor is not allowed to guide the compiler activity through templates. Another limitation comes from the fact that the compiler transformations are included in the compiler internals, so some flexibility is lost.

## 5 Conclusions and Future Work

In this paper, we have shown our experience in using templates oriented to source to source code transformations. We have implemented a new transformation pass for the Open64 compiler infrastructure to generate parallel code given the OpenMP directives in the source code. For each OpenMP directive (PARALLEL, DO, SECTIONS, etc.) our compiler pass takes one templates and generates the parallel code according to them. We are currently working in the testing of this compiler pass and the templates already written. For this task, we use standard OpenMP benchmarks written in Fortran. After that, we plan to use the same technique to support also the C language, writing the associated templates in C and testing the complete infrastructure with OpenMP benchmarks written in C.

## References

- [1] <http://www.cepba.upc.es/intone>
- [2] X. Martorell, E. Ayguad, N. Navarro, J. Corbalan, M. Gonzalez and J. Labarta, "Thread Fork/Join Techniques for Multi-level Parallelism Exploitation in NUMA Multiprocessors", Proceedings of the 13th. ACM International Conference on Supercomputing (ICS99).
- [3] X. Martorell, E. Ayguad, N. Navarro and J. Labarta, "A Library Implementation of the Nano-Threads Programming Model", Proceedings of the 2nd. Europar Conference 1996.
- [4] NANOS project, <http://www.ac.upc.es/nanos>
- [5] C. Brunschen and M. Brorsson, Lund University "OdinMP/CCp – A Portable Implementation of OpenMP for C", First European Workshop on OpenMP, (EWOMP99)
- [6] "Fortran Language Specification , v2.0", <http://www.openmp.org>
- [7] <http://sourceforge.net/projects/open64/>
- [8] <http://www.cepba.upc.es/pop>
- [9] <http://phase.hpcc.jp/Omni/>
- [10] D. Quinlan, M.Schordan, Qing Yi, B. Bronis and R. de Supinski, "A C++ Infrastructure for Automatic Introduction and Translation of OpenMP Directives", International Workshop on OpenMP Applications and Tools , June 2003 Toronto (Canada)
- [11] <http://sourceforge.net/projects/ipf-orc>
- [12] D. R Butenhof, "Programming with POSIX threads", Professional CVomputing Series, Addison-Wesley, ISBN 0-201-63392-2