

# Nanos Mercurium: a Research Compiler for OpenMP

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

Computer Architecture Department, Technical University of Catalonia,

cr. Jordi Girona 1-3, Mòdul D6, 08034 - Barcelona, Spain

{jbalart, aduran, marc, xavim, eduard, jesus}@ac.upc.es

## Abstract

*OpenMP is still in the process of being defined and extended to broad the range of applications and parallelization strategies it can be used for. The proposal of OpenMP extensions may require the implementation of new features in the runtime system supporting the OpenMP parallel execution and modifications in an existing OpenMP compiler, either at the front end (parsing of new directives and clauses) or back end (code generation targeting a specific runtime system). It may even imply to modify the internals of the compiler to accommodate new concepts, data structures or code manipulation routines. The objective of the Nanos Mercurium compiler is to offer a compilation platform that OpenMP researchers can use to test new language features. It needs to be: 1) robust enough to enable testing with real applications and benchmarks, and 2) easy to add new features into it, hiding as much as possible all its internals. To achieve the first objective, we decided to build Nanos Mercurium on top of an existing compilation platform, the Open64 compiler. In order to achieve the second objective, Nanos Mercurium uses templates of code for specifying the transformations of the OpenMP directives. In order to validate Nanos Mercurium, we show how to implement dynamic sections, a relaxation of the current definition of SECTIONS to allow the parallelization of programs that use iterative structures (such as while loops) or recursion. It is shown as an alternative to the workqueueing execution model proposed by KAI/Intel.*

## 1 Introduction

Shared-memory parallel architectures are becoming common platforms for the development of CPU demanding applications. Today, these architectures are found in the form of small symmetric multiprocessors on a chip, on a board, or on a rack with a modest number of processors (usually less than 32 or 64). In order to benefit from the potential parallelism they offer, programmers require pro-

gramming models to develop their parallel applications with a reasonable performance/simplicity trade-off. 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.

OpenMP [5] has become the standard for parallel programming in shared memory parallel architectures. OpenMP offers a set of directives or pragmas that are included in the specification of high-level languages such as C or Fortran. OpenMP results successful for the parallelization of a broad range of numerical applications coming from different engineering fields. Basically, the success comes from the fact that OpenMP makes the parallelization process easy and incremental.

However, the OpenMP community is conscious about the fact that certain parallelization structures are not possible (or very difficult to implement) with the current definition. For example, the parallelization of non-numerical applications, which usually deal with memory linked data structures and/or rely on divide and conquer strategies. Those applications are coded through the use of iterative structures like while loops and recursion. For this reason, the language specification is open to extensions. The OpenMP Architecture Review Board is in charge of studying and discussing proposals that come from the OpenMP community. Application developers as well as researchers in compiler optimizations can submit their proposals for possible extensions of the specification. For example, KAI/Intel proposed workqueueing to handle the parallelization of the above mentioned non-numerical codes [9, 10]. The proposal is based on the introduction of a new work-sharing construct, which specifies a queue (TASKQ) where tasks are queued for execution (TASK). Threads that belong to the team dynamically extract tasks from that queue.

The process of designing, implementing and evaluating

a new proposal could be more effective if there were open source compiler and runtime infrastructure for OpenMP simple to modify and extend. There have been several recent attempts, as for instance NanosCompiler [1] for Fortran77, Omni [7] for Fortran77 and C and OdinMP [4] 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. Nanos Mercurium is presented as a new approach to build an infrastructure for OpenMP extensions. It has been designed with two objectives: 1) it needs to be robust enough to enable testing with real applications and benchmarks, and 2) it has to be easy to add new features into it, hiding as much as possible all its internals. To achieve these objectives we decided to build Nanos Mercurium on top of an existing compilation platform, the Open64 compiler, and to use templates to specify the transformations for OpenMP construct.

The paper is structured as follows: Section 2 describes the basic OpenMP compiler transformations. Section 3 describes the approach followed in the design of Nanos Mercurium: specification of code transformations based on templates. As an example, Section 4 presents a relaxation of the OpenMP SECTIONS work-sharing construct and its implementation in Nanos Mercurium. Section 5 evaluates the performance achieved with some codes and compares this performance with the one achieved with the Intel compiler with taskqueues. Finally, Section 6 concludes the paper and outlines future work.

## 2 OpenMP Compiler Support

In this section we summarize the common functionalities available in most runtime systems that support the OpenMP execution model, and show the compiler transformations needed in the source-to-source translation process. The most important features need to be considered: how threads are created/terminated, how threads execute the code inside the parallel region, how threads parcel out the work specified in work-sharing constructs, how threads access data according to the data scoping rules and the mechanisms available to synchronize threads at specific points. All these features are jointly achieved by the runtime and compiler. In this paper, we show the transformations one by the Nanos Mercurium compiler when targeting the NthLib OpenMP runtime. We consider the description of the NthLib internals out of the scope of this paper (for more details the reader is referred to [3, 2]). Both compiler and runtime are available from the project web site at CEPBA ([www.cepba.upc.es/mercurium](http://www.cepba.upc.es/mercurium)).

```

void foo (double A [100][100], double B [100][100],)
{
#pragma omp parallel
{
#pragma omp sections
#pragma omp section
{
initialize (A );
}
#pragma omp section
{
initialize (B );
}
}
}
a) Example code with SECTIONS construct

void par_foo_01 (double A [100][100], double B [100][100] )
{
callsections_foo_01 (A ,B)
}
b) Parallel code encapsulation

void foo (double A [100][100], double B [100][100],)
{
nth_nthreads = nth_cpus_actual ();
nth_depadd (nth_self (), nth_nthreads + 1 );
for (th = 0; th < nth_nthreads; th++)
nth_thread_create_ls_vp (par_foo_01, 0, th, 2, A , B )
nth_block ();
}
c) Code for thread creation/termination

void sections_foo_01 (double A [100][100], double B [100][100],)
{
nth_begin_for (0, 1, 1, DYNAMIC, 1 );
while (nth_next_iters (&nth_down, &nth_up, nth_last) ) {
for (nth_sect = nth_down; nth_sect < nth_up; nth_sect) {
if (nth_sect == 0 ) section_foo_00 (A );
if (nth_sect == 1 ) section_foo_01 (B );
}
}
nth_end_for (1 );
}
d) Code for SECTIONS work sharing construct

```

Figure 1. OpenMP example and compiler transformations.

The transformation process is described using the sample code shown in Figure 1.a. This section will mainly focus on code transformations for thread creation and termination, and for work distribution. The reader is referred to [8] for a description of the other compiler transformations.

### 2.1 Parallel Code Encapsulation

In order to allow threads to execute the code encapsulated in the PARALLEL construct, 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 compiler operations like code extraction and symbol reference gathering. This last one is necessary to declare subroutine arguments that define the correct context to execute the encapsulated code. For the example in 1.a, the compiler generates the subroutine in figure

1.b.

## 2.2 Thread Spawn and Join

Once the parallel code has been encapsulated, the compiler replaces the original code by a block of statements in charge of spawning and joining the team of threads created to execute it. This block of statements mainly contains a loop that adds a new thread to the team in each iteration. The compiler injects a runtime call to the `nth_create_ls_vp` service to create the thread, provide the address of the encapsulated function to execute and the necessary arguments. Before the execution of this loop, the compiler injects code to query the runtime about the number of available threads to create the team. This corresponds to the runtime invocation to `nth_cpus_actual` service. Once all threads are created, the thread that has spawned the parallelism blocks and waits until all threads in the team finish the execution of the parallel code. This is accomplished by the injection of another runtime call to service `nth_block` that implements the implicit barrier at `END PARALLEL`. Figure 1.c shows the code that the compiler should generate for the parallel execution of the example in Figure 1.a.

### 2.2.1 Work Distribution

For each work-sharing construct in a parallel region, the compiler performs a transformation process similar to the one described for the parallel region. First the code inside the work-sharing 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 code encapsulation process described in previous section. The code inside the new encapsulated function depends on the work-sharing (i.e. `DO`, `SECTIONS`, `SINGLE` or `WORKSHARE`). For example, for a `SECTIONS` construct, the compiler would generate the code shown in Figure 1.d. The runtime services `nth_begin_for` and `nth_end_for` define and conclude the execution of the work-sharing construct. The compiler generates a loop that each thread in the team executes to request work to execute (service `nth_next_iters`). The body of this loop simply redirects the execution to the appropriate `SECTION` according to their lexicographic order in the source code. The reader is referred to [8] for additional details about this code and code generation for other work-sharing constructs.

## 2.3 Parametrization of the Transformations

All the compiler transformations at this point have aspects that depend on the source code that is being trans-

formed while other aspects are independent. For example, the symbol context depends strictly on the symbols that are referenced inside the parallel code and the OpenMP data scoping clauses. Or for example, the injection of the call to `nth_create_ls_vp` is always done in the same way except for some of its arguments. For this reason, the proposed mechanism in this paper to specify the transformation process allows the specification of constant and variable parts, i.e. parts that depend or not on the source code being transformed.

## 3 Template-guided Transformations

The Nanos Mercurium compiler is based on the definition of a set of templates that guide the compiler in the process of transforming the code according to the OpenMP directives. In order to have a robust platform able to handle real applications and benchmarks, we decided to implement Nanos Mercurium on top of the Open64 compiler [6]. We started from Open64 with a built-in OpenMP parser and internal representation, named `whirl`, able to represent OpenMP constructs. In order to drive the transformation process, we added a new phase that is executed after the corresponding Fortran or C 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 Abstract Syntax Tree (AST). When our phase is executed, all the directives are transformed applying a set of templates, obtaining a new `whirl` representation with all the OpenMP directives replaced by calls to the `NthLib` thread library. After that, we use one of the Open64 back-ends over the `whirl` intermediate representation that emits source code in the same input language.

### 3.1 Template Definition

The templates used by our compiler are written in the same language as the application being compiled, with no additional extensions to it. There are three simple rules that are needed 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 repeated a variable number of times.

- Rule 1: All variables in a template whose names start with a certain prefix, `tpl_` in our implementation, are template variable parts that the compiler will replace with an expression. The transformation to do over this variables is coded in the compiler internals. The compiler includes a correspondence between template variable names and the corresponding transformation.

- **Rule 2:** When the logical expression in a conditional construction (if statement) includes 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) includes 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  $n$ , depending on the code being compiled.

Rule 1 needs to be extended in order to handle the possibility of having a variable outside any expression or language construction. 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 the 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 part of code in template.
- **Operation 8:** extract a part of code in a template.
- **Operation 9:** change the name of a template variable.

The current status of the Nanos Mercurium compiler supports near the whole language specification (OpenMP 2.0). All the compiler transformations are specified through the use of code templates. For each language construct there is one template code to be applied. In [8] we presented how the code templates are used to implement the necessary support in the compiler for PARALLEL and work-sharing constructs. In the following section, we show how to build a new template to instruct the compiler to transform a relaxed version of the SECTIONS construct, what we call dynamic sections.

## 4 Dynamic Sections Proposal

The proposal in this section tries to address the limitations in the OpenMP programming model regarding applications based on recursion and/or traversing memory linked data structures. The proposal is based on relaxing the current specification for SECTIONS, mainly allowing a SECTION to be instantiated multiple times inside the scope of SECTIONS and executing code outside any SECTION by a single thread. The proposal has certain similarities with the KAI/Intel workqueueing proposal [9, 10], hiding the concept of queues to the programmer. This proposal is used as an example to show how to extend Nanos Mercurium by defining the appropriate template to handle dynamic sections.

### 4.1 Relaxing the SECTIONS construct

The SECTIONS work-sharing construct allows the programmer to define several portions of code to be executed in parallel. OpenMP does not allow to have code outside any SECTION (in fact in the Fortran specification the first SECTION can be omitted but the compiler assumes that it exists). We relax this constraint so that this code outside all SECTION is executed single threaded. This thread would be the responsible for work generation, allowing the programmer to implement the work generation strategies that are commonly used in the applications mentioned before. In addition, a SECTION could now be instantiated multiple times, for example because it is included in a loop that traverses a linked-data structure or because it is inside a recursive code structure.

When a team of threads executing a PARALLEL region encounters a SECTIONS construct, only one of them is going to execute the code inside the SECTIONS construct and outside any SECTION construct. When the thread executing the single-threaded part of SECTIONS finds an instance of a SECTION, it simply queues the SECTION in an internal queue of work to be executed by the team of threads.

The code in figure 2 shows an example of use. Subroutine `queens_par` contains the definition of a parallel region. Inside this region, there is a SECTIONS construct that includes a `for` loop. In each iteration of this loop a SECTION construct is instantiated creating a new branch in the recursion tree.

One of the main differences with the workqueueing model is that this proposal does not allow the nesting of SECTIONS while KAI/Intel allow the nesting of TASKQ. The nesting should be done by nesting PARALLEL regions, each including the SECTIONS. The first implication of this is that the same team of threads is executing the nested TASKQ, while in our proposal a new team is created in each PARALLEL region, disabling the possibility of doing work

```

void queens_par (int board, int level, ...)
{
    int copy_board [MAX_SIZE];
    if ( level == MAX_SIZE ) return;
    #pragma omp parallel
    #pragma omp sections
    {
        for ( i = 0; i < MAX_SIZE; i++ ) {
            #pragma omp section
            {
                for ( j = 0; j < level; j++ )
                    copy_board [j] = board [j];
                ...;
                queens_par (copy_board, level+1,...);
            }
        }
    }
}

```

**Figure 2. Dynamic sections example: queens code.**

stealing. The second implication is that the implicit barrier at the end of PARALLEL does not exist when simply nesting TASKQ. We are investigating the effects that these implications may have in applications and see if they limit the parallelism that can be exploited. In general, it should be considered the possibility of nesting any kind of OpenMP work-sharing construct, but this is considered out of scope of this paper and subject of further discussion.

If the compiler detects that there is no code outside any SECTION, it can turn to the initial code generation strategy (what we name static sections).

## 4.2 Template Specification for Dynamic Sections

Figure 3 shows the template specification that instructs the compiler how to transform the code for a dynamic SECTIONS work-sharing construct. The code uses the support provided by the NthLib thread runtime, but the mechanisms to implement the compiler transformation are valid independently of the target runtime library. The template is written in C and includes all the necessary declarations to be a correct C code. Some of these declarations correspond to template variables which are going to cause the compiler apply the appropriate transformations.

The name of the subroutine where the template code is located is identified by a template variable TPL\_DYNAMIC\_SECTIONS which is going to cause the

compiler to generate a new identifier for the subroutine. The template variable TPL\_SECTIONS\_PARAMS appears as a subroutine argument. It is going to be replaced by the declaration of all the symbols referenced by all the dynamic sections. The template code includes an if statement that checks if the thread is the master of the current team. This ensures that only the master thread is going to execute the code related to the work distribution. This code consists of several statements that cause the executing thread communicates to the runtime the number of dynamic sections that are going to be generated. This number is obtained by the compiler after the transformation applied to the template variable TPL\_NUM\_SECTIONS is done. This variable expands to the number of sections in the SECTIONS work-sharing construct. The runtime call nth\_depadd communicates this number to the runtime. After that, a do while loop is coded in the template, but controlled by a template variable: TPL\_SECTIONS. This is going to cause the compiler traverses the list of sections. For each section, the code in the loop body is replicated and the appropriate transformations are applied in it. The loop body in the template is formed by several statements where the TPL\_SECTION\_NUM\_PARAMS, TPL\_SECTION\_SUBROUTINE, TPL\_SECTION\_PARAMS variables appear. The compiler replaces the variable TPL\_SECTION\_NUM\_PARAMS by the number of arguments in the section's subroutine. The TPL\_SECTION\_SUBROUTINE variable appears as argument of the runtime call to service nth\_create\_ls. This variable is replaced by the identifier of the routine generated for the section being treated. Finally, the variable TPL\_SECTION\_PARAMS is replaced by the list of arguments that are referenced inside the section's code.

The OpenMP definition forces the threads to perform a barrier synchronization at the end of the SECTIONS execution. Notice that the SECTIONS construct is the only work-sharing inside the parallel region. This allows for optimization and removing the barrier as it is going to be performed at the END PARALLEL construct. This is achieved in the template code by the runtime calls to nth\_successor and nth\_depadd. Both calls are responsible to link the dynamic sections with the thread that spawned the parallel region, forcing that this one can not be terminated until all the dynamic sections are terminated too.

Notice that most of the variable parts in the template correspond to information that the compiler has to compute, no matter the code generation associated to the dynamic sections. If this code generation has to be changed, the changes can be done in the template, with no need of implementing the changes inside the internals of the compiler.

```

void TPL_DYNAMIC_SECTIONS (int TPL_SECTIONS_PARAMS)
{
    long long nth_succ;
    int nth_nsections;
    int ndep;
    int nth_num_params;
    long long nth;
    int TPL_NUM_SECTIONS;
    bool TPL_SECTIONS;
    int TPL_SECTION_NUM_PARAMS;
    int TPL_SECTION_PARAMS;

    if (nth_iam_master ())
    {
        nth_succ = nthf_successor (nthf_self ());
        nth_nsections = TPL_NUM_SECTIONS + 1;
        nthf_depadd (&nth_succ, &nth_nsections);
        nth_ndep = 0;
        while (TPL_SECTIONS)
        {
            nth_num_params = TPL_SECTION_NUM_PARAMS;
            nth = nthf_create_1s (TPL_SECTION_SUBROUTINE,
                                &ndep,
                                &nth_succ,
                                &nth_num_params,
                                TPL_SECTION_PARAMS);

            nth_to_rq (nth);
        }
    }
}

```

**Figure 3. Template for dynamic sections.**

## 5 Testing and Evaluation

In this section we describe the tests performed to validate the proposed dynamic sections and the transformation done by the compiler to implement them.

### 5.1 Benchmarks

In order to test our proposal and implementation, we have used some of the examples that KAI/Intel used to motivate and evaluate their implementation of taskqueuing. We mainly replaced TASKQ with SECTIONS and TASK with SECTION. All of them have in common that the algorithms used in the computations are expressed through recursion. Parallelism appears among the recursion branches that each application defines:

- **Queens:** The algorithm computes the different solutions for placing  $n$  chess queens in a chess board, none of them "killing" another. The computation is done recursively. Each recursive branch in the recursion tree, can be executed in parallel.
- **Strassen:** Strassen matrix multiplication algorithm decomposes square matrices into four square sub-matrices and then compute the whole matrix multiplication performing only seven sub-matrices

multiplications and eighteen sub-matrices additions/subtractions. Recursive decomposition can be applied to create as many matrix multiplication tasks as desired and execute all them in parallel.

- **MultiSort:** Multisort is a variation of the mergesort algorithm. A recursive divide and conquer approach is used. An array of  $N$  64-bits elements is divided in four sub-arrays, each sub-array is sorted recursively and finally the four sorted sub-arrays are merged first to obtain two sorted parts and lastly only one.

## 5.2 Initial Experiments

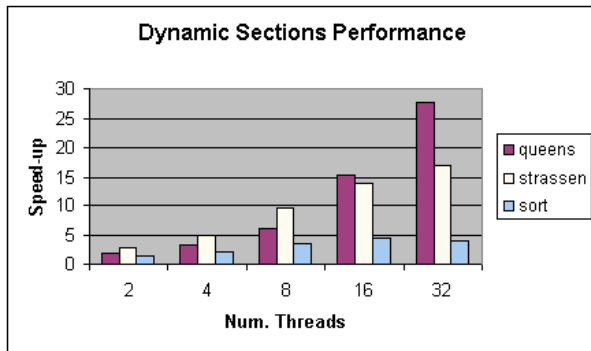
The experimental platform for this paper is a SGI Origin2000 with 64 R10000 processors (250 Mhz) and Irix 6.5. Nanos Mercurium generates a transformed source code that is finally compiled and linked with NthLib using the SGI native compiler to generate the executable code.

Figure 4 shows the performance, in terms of speed-up with respect to the sequential version, that is obtained for the benchmarks. These numbers are initial and there is still more work to do in order to understand them and detect who is the responsible for the performance degradation observed (application, dynamic sections proposal, compiler transformation or OpenMP runtime):

- **Queens:** the benchmark is executed with a chess board of  $13 \times 13$ . Each time the parallelism is spawn, 13 dynamic sections are created. In order to feed 4, 8, 16, and 32 threads it suffices to have 2 levels of recursion. As shown in Figure 4 the application scales quite well with the number of processors.
- **Strassen:** the benchmark is executed with a matrix of 1280 doubles. As shown in Figure 4 the benchmark achieves efficiency up to 16 threads. Using more threads is not providing more parallelism (speedup of 16.95 with 32 threads).
- **MultiSort:** This is the application that behaves worst. Although the speed-up increases up to 16 processors, the efficiency decreases when more processors are used (0.72 with 2 processors and 0.27 with 16).

## 6 Conclusions and Future Work

In this paper we present Nanos Mercurium, a new research compiler infrastructure that we want to offer to the OpenMP community to test new proposals to the language. The compiler source-to-source transformations are specified using templates, which relieves the user from going into the deep internals of the compiler.



**Figure 4. Performance for the Nanos Mercurium implementation of dynamic sections.**

The templates for the main OpenMP constructs are described in a companion paper [8]. In this paper we want to show how to specify the template for a relaxed definition of SECTIONS, what we call dynamic sections. Dynamic sections allow to extract parallelism out of programs that make use of recursion or iterative traversals of data structures to generate work to be executed in parallel. The proposal and implementation are tested using some benchmarks and their performance is evaluated.

## Acknowledgments

This research has been supported by the Ministry of Science and Technology of Spain under contract TIC2001-0995-C02-01 and the European Union under contract IST-2001-33071.

## References

- [1] M. Gonzalez, E. Ayguade, J. Labarta, X. Martorell, N. Navarro and J. Oliver. "NanosCompiler: A Research Platform for OpenMP Extensions", In First European Workshop on OpenMP, Lund (Sweden), October 1999
- [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] C. Brunschen and M. Brorsson, Lund University "OdinMP/CCp – A Portable Implementation of OpenMP for C", First European Workshop on OpenMP, (EWOMP99)
- [5] "Fortran Language Specification , v2.0", <http://www.openmp.org>
- [6] <http://sourceforge.net/projects/open64/>
- [7] <http://phase.hpcc.jp/Omni/>
- [8] J. Balart, A. Duran, M. Gonzalez, X. Martorell, E. Ayguade, J. Labarta, "Skeleton driven compiler transformations", 11th Workshop on Compilers for Parallel Computers, July 7-9
- [9] S. Shah, G. Haab, P. Petersen, J. Throop, "Flexible Control Structures for Parallel C/C++", First European Workshop on OpenMP, September 1999, Lund University, Lund, Sweden
- [10] E. Su, X. Tian, M. Girkar, H. Grant, S. Shah, P. Peterson, "Compiler Support of the Workqueuing Execution Model for Intel SMP Architectures" Fourth European Workshop on OpenMP, September 2002, Rome