NanosCompiler: A Research Platform for OpenMP Extensions

Eduard Ayguadé, Marc Gonzàlez, Jesús Labarta,
Xavier Martorell, Nacho Navarro and José Oliver
Computer Architecture Department, Polytechnic University of Catalunya,
cr. Jordi Girona 1-3, Mòdul D6, 08034 - Barcelona, Spain

Abstract

This paper describes the main functionalities of the OpenMP NanosCompiler. It is a source-to-source parallelizing compiler implemented around a hierarchical internal program representation that captures the parallelism expressed by the user (through OpenMP directives and extensions) and the parallelism automatically discovered by the compiler through a detailed analysis of data and control dependences. The compiler is finally responsible for encapsulating work into threads, establishing their execution precedences and selecting the mechanisms to execute them in parallel. One of the main features of the NanosCompiler is the abilitity to exploit multiple levels of parallelism and generate work from multiple simultaneously executing threads.

1. Introduction

Parallel architectures are becoming affordable and common platforms for the development of computing-demanding applications. Users of such architectures require simple and powerful programming models to develop and tune their parallel applications with reasonable effort. These programming models are usually offered as library implementations or extensions to sequential languages that express the available parallelism in the application. These extensions are usually offered by means of directives and language constructs (e.g. OpenMP [10] and HPF [6]).

Most current systems (compilers and run–time threads support) are based upon the exploitation of a single level of parallelism around loops (for example, the current version of the SGI MP library, the SUIF compiler infrastructure [4] or the MOERAE portable thread–based interface for the Polaris compiler [5]). Exploiting a single level of parallelism means that there is a single thread (master) that produces work for other processors (slaves). Once parallelism is activated, new opportunities for parallel work creation are ignored by the execution environment. Exploiting this parallelism may incur in low performance returns when the number of processors to run the application increases.

Multi–level parallelism enables the generation of work from different simultaneously executing threads. Once parallelism is activated, new opportunities for parallel work creation result in the generation of work for all or a restricted set of processors. We believe that multi–level parallelization will play an important role in new scalable programming and execution models. Nested parallelism may provide further opportunities for work distribution, both around loops and sections; however, new issues may arise in order to attain high performance. OpenMP [10] includes in its definition the exploitation of multi–level parallelism through the nesting of parallel constructs.

Another alternative to exploit multi–level parallelism consists on combining the use of two programming models and interfaces, like for example the use of MPI coupled with either OpenMP or HPF [2]. The message passing layer is used to express outer levels of parallelism while the data parallel layer is used to express the inner ones.

The Illinois–Intel Multithreading library [3] targets shared–memory systems. It also supports multiple levels of general (unstructured) parallelism. Application tasks are inserted into work queues before execution, allowing several task descriptions to be active at the same time. Kuck and Associates, Inc. has made proposals to OpenMP to support multi-level parallelism through the WorkQueue mechanism [7], in which work can be created dynamically, even recursively, and put into queues. Within the WorkQueue model, nested queuing permits a hierarchy of queues to arise, mirroring recursion and linked data structures. These proposals offer multiple levels of parallelism but do not support the logical clustering of processors in the multilevel structure, which may lead to better work distribution and data locality exploitation.

The NanosCompiler targets the exploitation of multiple levels of parallelism using OpenMP as a single programming paradigm. The paper first discusses the internal representation based on a hierarchical task graph. Then the paper presents some extensions to the current definition of OpenMP which are currently supported by the compiler. The proposals can be grouped into two major cathegories: the first one deals with the specification of processor groups when the application may benefit from the exploitation of multiple levels of parallelism; the second one deals with the specification of precedence relations among different sections in a `SECTIONS` work–sharing construct. The code generated by the compiler contains calls to a highly tuned user-level threads library (NthLib [9, 8]). The paper also discusses the requirements needed at this level to efficiently support multiple levels of parallelism and processor groups, and the execution of parallel tasks expressed by means of a general unstructured hierarchical task graph.

2 The NanosCompiler internal representation

The NanosCompiler is based on Parafrase–2 [11] and in its internal representation taking the form of a Hierarchical Task Graph (HTG). From the HTG, the back–end generates parallel code using the services provided by the NthLib threads package. The hierarchical nature of the HTG enables the NanosCompiler to capture multiple levels of parallelism available in the application.

2.1 The Hierarchical Task Graph

The HTG is implemented as a hierarchical acyclic directed graph composed of nodes and edges. Each node represents a collection of statements to be executed. There are five different types of nodes: compound, loop, simple, start and stop. Compound and loop nodes are defined recursively, encapsulating new nodes and thus creating a hierarchical representation of the program.

Simple nodes are associated to single statements in the source code. All the simple nodes that make up a basic block are organized as an HTG and encapsulated in a compound node. Each compound node also includes two special nodes: start and stop; they have no direct correspondence to the source code and represent decision (scheduling) points in the code generation phase. Loop nodes are used to represent iterative constructs and encapsulate a new level in the hierarchy; each loop node is composed of two nodes: header and body. The header node contains the evaluation of the condition that controls the execution of the loop. The body node is a compound node containing all the statements in the original loop body.

Edges are used to represent control and data dependence information. They are obtained after building the Control Flow Graph (CFG) and the Control Dependence Graph (CDG) over the HTG and determine which nodes are causing control dependences. Through symbolic analysis, the Data Dependence Graph (DDG) is also defined over the HTG. Once the CDG and the DDG are obtained, precedences between nodes can be stablished defining sucessor/predecessor relationships. The compiler assumes a precedence relation when two nodes are connected through a control or data dependence edge.

At this point, the information contained in the HTG concerns to precedences between nodes. This information is sufficient to determine the maximum parallelism that can be exploited at a specific level of the hierarchy. However, for an efficient parallelism exploitation it is necessary to define tasks whose granularity is appropriate for the run–time environment (threads library) supporting the parallel execution. To achieve this, the compiler has to perform an estimation of node granularities (either using profile information or doing static analysis) and apply some node grouping strategies that do not reduce the useful parallelism of the application.

The NanosCompiler parses a large part of the current OpenMP definition. User annotations refering to parallel sections and loops are used to supersed the parallelism automatically discovered by the compiler. Parallelism expressed through `PARALLEL` adds new levels in the hierarchical representation and represent thread creations. Work–sharing directives specify work distribution strategies for the threads previously created. For each `SECTIONS` work–sharing construct, possible precedences detected by the compiler are removed and a new task (compound node) in the HTG is defined for the nodes that represent the statements in each `SECTION`. The `DO` work–sharing construct is used to tag the corresponding loop node in the HTG as parallel, independently of the data dependence analysis done by the compiler. The compound node that contains the statements enclosed by a `SINGLE` work–sharing construct is tagged so that parallel code is generated ensuring that a single processor will execute the node.

2.2 Code generation and execution model

The compiler generates a function that encapsulates the code associated to each task of the HTG. The parallel execution of the program consists on the execu-

tion, in the order specified by the precedence relations, of the functions associated to the HTG tasks following the Nano–threads programming model [12]. The execution of a compound node begins with the execution of the function associated to the start node. Functions are executed through user-level entities called *nanothreads*. A nanothread corresponds to the instantiation of an HTG task in the form of an independent run–to–completion user–level control flow.

The code corresponding to the start node is in charge of the application level scheduling for the tasks enclosed in the compound node. This code evaluates, at run–time, the current availability of resources and creates the task structures that will be later instantiated as nanothreads, following the corresponding precedence relations among them. Virtual processors assigned to the application execute the parallel tasks being able to generate further parallelism when available. The start node is also in charge of the allocation of the local address space used to privatize variables for the enclosed parallelism.

The code associated with the stop node is in charge of waiting for the termination of the parallel tasks in the compound node. When all tasks have terminated, it deallocates the local address space created by the start node. In order to ease code generation, the compiler encapsulates in a single function the start and stop nodes. The stop node checks the status of the execution environment, being able to correct, when necessary, processor preemptions done by the operating system. Finally, it is in charge of satisfying the precedence relations with the successor nodes of the corresponding compound node, allowing them to execute.

Each thread executes the code that corresponds to the work distribution scheme specified by the OpenMP directives or by the parallelism automatically detected by the compiler.

2.3 NthLib support

The execution model defined in the previous section requires some functionalities not available in most of the current user–level threads libraries supporting parallelizing compilers.

The most important aspect to consider is the support for spawning nested parallelism. Current implementations of thread packages spawn parallelism through a work descriptor located at a fixed memory location. This is an optimal form of spawning in which the master and slave processors only take one cache miss to generate and access the work descriptor, respectively. Generating parallelism through a fixed memory location forbids the spawning of multiple levels of parallelism because the location cannot be reused till the parallelism is terminated. Therefore, once parallelism is activated, new opportunities for parallel work creation are ignored by the execution environment. Multi–level parallelism enables the generation of work from different simultaneously executing threads. In this case, new opportunities for parallel work creation result in the generation of work for all or a restricted set of processors.

The NthLib package interface has been designed to provide different mechanisms to spawn parallelism. Depending on the hierarchy level in which the application is running, the requirements for creating work are different. When spawning the deepest (fine grain) level of parallelism, the application only creates a work descriptor and supplies it to the participating processors. The mechanism is implemented as efficiently as the one currently available in most of the current available thread packages. However, extra functionality has been included to allow work supply from several (simultaneously executing) threads [8]. On the other hand, when the application knows that it is spawning coarse grain parallelism, not at the deepest level, it can pay the cost of supporting nested parallelism. Higher levels of parallelism, containing other parallel regions, are generated using a more costly interface that provides task descriptors with a stack [9]. Owning a stack is necessary for the higher levels of parallelism to spawn an inner level; the stack is used to maintain the context of the higher levels, along with the structures needed for joining the parallelism, while executing the inner one.

3. Some extensions to OpenMP

In this section we briefly describe two sets of extensions to the OpenMP programming model that have been implemented using the NanosCompiler and NthLib. The first set is oriented towards the definition of processor groups. The second set is oriented towards the definition of precedences among sections.

3.1 Processor groups

A group of processors is defined by a 'master' thread and a number of 'slave' threads. The definition of groups may be originated in any parallel construct. Once defined, work–sharing constructs inside the parallel construct will assign work to the master threads (instead of assigning the work to all the threads available). The slave threads will cooperate with the master in the exploitation of any additional parallelism inside these work–sharing constructs.

The extensions proposed allow: 1) the definition of the groups (i.e. the number of master threads and the number of slave threads assigned to each master); and 2) the assignment of work to the groups. This allows the user to control the allocation of work, avoiding the default allocations performed by the compiler. This default allocation (lexicographical order of chunks of iterations in the DO work–sharing construct and code segments parceled out by each SECTION in a SECTIONS work–sharing construct. This defautl assignment is a consequence of the unique work descriptor mechanism used to generate work. The descriptor contains the lower and upper bounds and step of the whole iteration space; each thread determines from its own thread identifier the chunk or chunks of iterations that has to execute. Similarly, the usual conversion of SECTIONS to a parallel loop that conditionally branches to each part also establishes the same default lexicographical order.

The GROUPS clause can be applied to any parallel construct. It establishes the groups of processors that will execute any work–sharing construct inside and nested parallel constructs:

```
C$OMP PARALLEL [DO|SECTIONS] [GROUPS(gdef[,gdef])]
 ...
C$OMP END PARALLEL [DO|SECTIONS]
```

where each group definer gdef has the following form:

```
[name:]ncpus
```

The name attribute is optional and is used to identify the group. The ncpus attribute is used to determine, from the number of currently available processors, the number of processors that will be assigned to the group. By default, groups are numbered from 0 to an upper value; this upper value is the number of groups defined within the clause GROUPS minus one.

A shortcut is available to specify the simplest group definition: GROUPS(number). In this case, the user specifies the definition of number groups, each one receiving the same number of processors.

If the number of processors available at the time of reaching the parallel construct is different than the sum of processors specified in the clause, the numbers specified are considered as proportions; the runtime system has to be able to distribute (in the more fair way and with minimum overhead) the total number of processors according to these proportions.

The default assignment of iterations in a DO work–sharing construct or individual sections in a SECTIONS work–sharing construct to processors can be changed by using the ONTO clause.

```
C$OMP DO [ONTO(expr)]
```

When this clause is used, expr specifies the group of processors that will execute a particular chunk of iterations (in fact, only the master of each group will execute the work). If the expression contains the loop control variable, then the chunk number (numbered starting at 0) is used to perform the computation; otherwise, all chunks are assigned to the same processor. In all group computations, a 'modulo the number of active groups' operation is applied. If not specified, the default clause is ONTO(i), being i the loop control variable of the parallel loop.

For the SECTIONS work–sharing contruct, the ONTO(expr) clause is attached to each SECTION directive to specify the group that would execute each section. Each expression expr can be different and is used to compute the group that will execute the statements parceled out by the corresponding SECTION directive. If the ONTO clause is not specified the compiler will assume an assignment following the lexicographical order of the sections.

At the compiler level, this extension required adding some new fields to the data structures that represent the nodes in the HTG. This new fields record the information provided by the user through the different clauses. At the threads library level, it requires the possibility of queuing threads into local queues (i.e. accessed by a single processor) in addition to global queues (i.e. accessed by all the processors). Local queues allow the supply of work to the set of processors that compose a group. It also requires additional fields in the thread descriptor to store, for each thread, information regarding the team of processors it is being or going to be executed.

3.2 Named sections

Directive NAME has been designed to extend the parallelism supported by the SECTIONS directive. SECTIONS assumes that all the sections inside can be executed completely in parallel. The NAME directive is proposed with the aim of allowing the definition of precedence relations among these sections.

```
C$OMP SECTIONS
...
C$OMP NAME(name) [PRED(name[,name])] [SUCC(name[,name])]
...
C$OMP END SECTIONS
```

The NAME directive gives a name to a section and allows the specification of its predecessors and successors (through the PRED and SUCC clauses, respectively). Each clause accepts a list of names of named sections.

Named and unnamed sections can be mixed within the same SECTIONS work–sharing construct.

This proposal has a direct representation in the internal HTG structure of the NanosCompiler. However, it requires support at the user–level threads library level in order to support an execution model guided by precedence relations resolved at run-time. This support is provided in NthLib through two entities included in the definition of the thread structure: each thread has an associated counter (that records the number of pending predecessor threads that need to finalize their execution) and a list of successor threads that can be executed as soon as the thread finishes. As soon as a thread finishes its execution, the precedences counter of all its successor tasks is decremented; if one of these counters reaches zero, then the successor thread is queued for execution.

4. Conclusions

In this paper we have presented the main aspects related with the internal representation of the NanosCompiler and the support provided by the NthLib user–level threads library. Finally, and in order to show the system as a research platform for the evaluation of OpenMP extensions, we briefly describe some extensions to the current definition of OpenMP oriented towards the definition of processor groups when multiple levels of parallelism exist in the application and the specification of precedences among sections. Although the majority of the current systems only support the exploitation of single–level parallelism around loops, we believe that multi–level parallelism will play an important role in future systems. In order to exploit multiple levels of parallelism, several programming models can be combined (e.g. message passing and OpenMP). We believe that a single programming paradigm should be used and should provide similar performance. The paper also discuses the requirements and functionalities needed in the threads library.

The reader is referred to [1] for an experimental evaluation of the NanosCompiler based on some SPEC95 benchmark applications.

Acknowledgments

This research has been supported by the Ministry of Education of Spain under contracts TIC98-511 and TIC97-1445CE, the ESPRIT project 21907 NANOS (www.ac.upc.es/nanos) and the CEPBA (European Center for Parallelism of Barcelona).

References

[1] E. Ayguadé, X. Martorell, J. Labarta, M. González and J.I. Navarro. Exploiting Multiple Levels of Parallelism in OpenMP: A Case Study. In *28th Int. Conference on Parallel Processing ICPP'99*, Aizu (Japan), September 1999.

[2] I. Foster, D.R. Kohr, R. Krishnaiyer and A. Choudhary. Double Standards: Bringing Task Parallelism to HPF Via the Message Passing Interface. In *Supercomputing'96*, November 1996.

[3] M. Girkar, M. R. Haghighat, P. Grey, H. Saito, N. Stavrakos and C.D. Polychronopoulos. Illinois-Intel Multithreading Library: Multithreading Support for Intel Architecture–based Multiprocessor Systems. *Intel Technology Journal*, Q1 issue, February 1998.

[4] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion and M. S. Lam. Maximizing Multiprocessor Performance with the SUIF Compiler. *IEEE Computer*, December 1996.

[5] S.W. Kim, M. Voss, and R. Eigenmann. MOERAE: Portable Interface between a Parallelizing Compiler and Shared-Memory Multiprocessor Architectures. Tech. Rep. Purdue Univ. ECE-HPCLab-98210.

[6] C.H. Koelbel, D.B. Loveman, R.S. Schreiber, G.L. Steele and M.E. Zosel. The High Performance Fortran Handbook. *Scientific Programming*, 1994.

[7] Kuck and Associates, Inc. WorkQueue Parallelism Model. http://www.kai.com, Fall 1998.

[8] X. Martorell, E. Ayguadé, J.I. Navarro, J. Corbalán, M. González and J. Labarta. Thread Fork/join Techniques for Multi–level Parallelism Exploitation in NUMA Multiprocessors. In *13th Int. Conference on Supercomputing ICS'99*, Rhodes (Greece), June 1999.

[9] X. Martorell, J. Labarta, J.I. Navarro, and E. Ayguadé. A library implementation of the nano-threads programming model. In *Euro-Par'96*, August 1996.

[10] OpenMP Organization. Fortran Language Specification, v. 1.0, www.openmp.org, October 1997.

[11] C.D. Polychronopoulos, M. Girkar, M.R. Haghighat, C.L. Lee, B. Leung, and D. Schouten. Parafrase–2: An environment for parallelizing, partitioning, and scheduling programs on multiprocessors. *International Journal of High Speed Computing*, 1(1), 1989.

[12] C.D. Polychronopoulos. Nano-threads: Compiler driven multithreading. In *4th Int. Workshop on Compilers for Parallel Computing CPC'93*, Delft (The Netherlands), December 1993.