An incremental compilation approach for OpenMP applications

Maurizio Giordano and Mario Mango Furnari

Istituto di Cibernetica "E. Caianiello" - C.N.R.
Via Campi Flegrei 34, 80078 Pozzuoli, Naples - ITALY
{m.giordano,m.mangofurnari}@cib.na.cnr.it

Abstract. This work presents a new approach to software development framework design for parallel programming: the *Graphical Parallelizing Environment*[1] (*GPE*). It adopts an incremental compilation process for OpenMP programming based on automatic detection of parallelism and user interaction for its calibration. GPE is extensible via plug-in modules providing new capabilities. It is an experimental OpenMP programming framework targeting shared-memory multiprocessors and clusters of PCs.

1 Introduction

In past years, several techniques were developed in the area of program automatic parallelization, like *data and control dependence analysis* [1], *symbolic and interprocedural analysis* [3, 4]. Several research projects [5, 6] dealt with the development of parallelizing compilers implementing most of these techniques.

Multithreaded applications, that were specifically targeted to shared-memory, may now use Software DSM to run in distributed settings. There are proposals [8, 9] to adopt a single programming paradigm, like OpenMP, independently from where the application will run, that is a multiprocessor, SMP or a cluster.

In this context, a pure automatic compiler-based approach to program parallelization has proved to be insufficient, since compilers cannot use information available only to users. This is even worst if the same parallel program will run on different multiprocessor architectures, or even on clusters of PCs.

In recent years, an alternative approach was proposed [5, 6] that combines automatic and manual parallelization: the programmer interacts with the compiler to supply his knowledge of the application. This additional information helps the compiler in carrying on the hard task of parallelism analysis and discovery.

We developed a new environment for program parallelization, named *Graphical Parallelizing Environment* (*GPE*). It adopts an incremental approach for the parallelization of programs based on both parallelism automatic detection (done by a parallelizing compiler) and the user intervention to drive code restructuring as well as parallelism annotation before generation of program executables.

The rest of the paper is so organized: section 2 describes the GPE architecture; section 3 gives and overview of the GPE modules for the visualization and modification of program parallelism; section 4 reports some conclusive remarks.

[1] GPE software is a result of the POP European project: IST-2001-3307

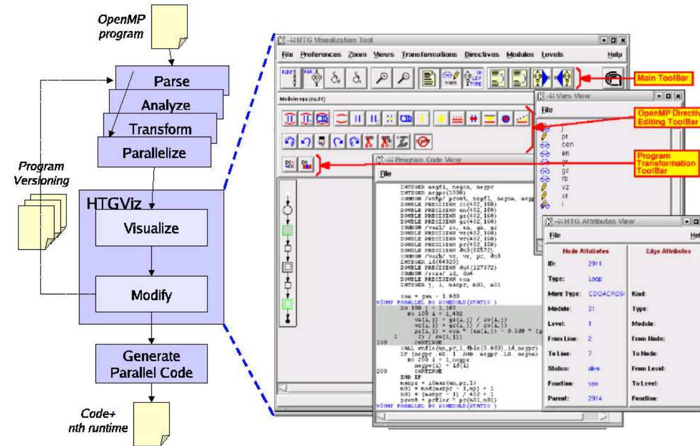

Fig. 1. The GPE architecture

2 GPE program development cycle

According to the GPE approach, OpenMP program parallelization is the result of a cyclic process in which, at each round, the following activities (see figure 1) are carried out:

1. *OpenMP program editing* - first, the programmer writes the FORTRAN source code with OpenMP annotations.
2. *Compilation* - the compiler performs data & control dependence analysis and detects program parallelism accounting also for OpenMP parsed directives.
3. *Parallelism visualization* - program parallelism and its sources are shown to users in a *Hierarchical Task Graph* [2] representation.
4. *Parallelism modification* - users restructure parallelism annotating the code with OpenMP directives and transforming loops to extract/tune parallelism.
5. *Parallel code generation* - OpenMP annotated program tasks are translated to FORTRAN code plus calls to a multithreaded library [7].

Steps 3 to 4 can be iterated to further tune application parallelism. Each modification to the source is saved in a program new version and version history is kept by the GPE *versioning module*. The multithreaded code produced in step 5 is compiled on the target architecture to generate the executable. Performance measurements and execution traces can be used in next rounds of the tuning

Parallelism detection and task formation is done by the *POP compiler* [9]. It is a source-to-source parallelizer that uses an aggressive approach for dependence testing. Parallelism detection is synthesized in a compiler internal representation: the *Hierarchical Task Graph* (*HTG*) [2]. User-compiler interaction relies on HTG handling: parallelism visualization/calibrating is based on graph-manipulation. We think that the HTG could be considered the intermediate program representation closer to the user conceptual view of the application parallel execution.

3 GPE modules

The GPE first design aimed to provide visualization and navigation of the program HTG, that synthesizes results of compiler analysis and parallelism discovery. We experienced that OpenMP program parallelization is often an incremental process involving both compiler techniques and programmer's restructuring decisions. The process is time-consuming as it implies hand-coding of many versions of the same program corresponding to different parallelization strategies.

Therefore, we redesigned the GPE to be an environment supporting the iterative process of OpenMP programming and extensible with new functionalities, added as plug-ins to the core system. With this new design the GPE has become a framework in which new capabilities and tools can be quickly developed and experimented. In what follows we describe the main GPE modules.

Visualization module - The GPE visualization module, named *Hierarchical Task Graph Visualization Tool* (*HTGViz*), displays compiler analysis (parallelism detection) results and provides facilities to navigate and correlate different information about the application parallelism discovered by the compiler. HTGViz offers three views of the application, that are hereafter described.

The *HTG Visualization View* is the main interface where HTGs of program subroutines are drawn. It allows to navigate through the HTG structure across hierarchy levels by means of a task expanding/collapsing facility. This feature simplifies HTG navigation when the program size and complexity increases.

The *Program Code View* illustrates the code in textual format. The interface shows the correspondence between program statements and HTG nodes during all user actions, like HTG navigation and directive insertion.

The *Vars View* shows, for each task node, the list of variables used (read/written) and their occurrences in the program. This helps the programmer in detecting variables to privatize or share in OpenMP parallel sections and loops.

Modification modules - The *Program transformation module* supports the set of loop transformations more frequently used by POP users during the experience in OpenMP programming, i.e *loop interchange*, *blocking* and *coalescing* [10].

Transformation capabilities are based on graph-manipulation with the possibility to choose different equivalent patterns. The module implements checks on transformation applicability and inputs mainly based on data and control dependence analysis. If the compiler detects constraint violations the transformation is forbidden; otherwise the system allows the programmer to apply it.

The *OpenMP editing module* is a GPE extension providing an easy-to-use editor, based on graph manipulation, to assist users in inserting/modifying OpenMP annotations. The module allows to restructure and overwrite parallelism specification in terms of directives during compilation, before parallel code generation.

The editing tool partially automatizes the task of OpenMP directive insertion/modification. It assists the user in generating well-formed directives offering commands for fast pre-formatted editing operations. Upon directive insertion, a form-like interface is prompted for the input of clauses and their arguments: it

displays the variables used (read/written) in the code enveloped by the directive. This information is useful to set variables as private or shared in the parallel threads. The tool performs directive applicability checks and syntax control.

The *Program versioning module* supports the tracking and re-using of intermediate versions during program development. Each code modification or OpenMP editing is saved in a program new version. The module maintains a history of program versions that can be navigated back and forth.

After the generation of the multithreaded binary and its execution, programmers may use runtime performance analysis information to restart application tuning from an intermediate version. To this aim the versioning module has a facility to store/reload program versions and history in/from a "project file".

4 Conclusions

The main novelty of GPE is its design as an extensible environment to support the incremental development cycle of OpenMP programs. At each round of the cycle, the user interacts with the compiler to tune the detected parallelism according to his knowledge of the application. GPE is extensible since new modules and tools can be implemented and plugged-in the GPE core to offer new functionalities, like modules supporting new analysis and transformation techniques.

Experiences of GPE usage in parallelizing OpenMP applications from NAS and SPEC95 benchmarks proved that performance measurements and trace data analysis are crucial to identify sources of performance drawbacks and to further improve program parallelization in next compilation steps. The *Program versioning* module was developed and integrated in GPE to facilitate this task.

References

1. Banerjee, U.: Dependence analysis for supercomputing. Kluwer Academic Publishers, (1988)
2. Girkar, M., Polychronopoulos, C.D.: The hierarchical task graph as a universal intermediate representation. Int. J. Parallel Programming **22** (1994) 519–551
3. Hall, M.W., *et al.*: Interprocedural Compilation on Fortran D. Journal of Parallel Distrib. Comput. **38**(2) (1996) 114–129
4. Haghighat, M.R., Polychronopoulos, C.D.: Symbolic analysis for parallelizing compilers. ACM Trans. on Programming Languages **18** (1996) 477–518.
5. Hall, M.W., *et al.*: Experience Using the ParaScope Editor. Proc. of Symp. Principles and Practice on Parallel Programming (1993)
6. Liao, S., *et al.*: Suif explorer: An interactive and interprocedural parallelizer. Proc. of Symp. on Principles and Practice of Parallel Programming (1999)
7. Martorell, X., *et al.*: A Library Implementation of the Nano-Threads Programming Model. Proc. of the 2nd Intern. Euro-Par Conf. (1996) 644–649
8. Omni OpenMP compiler project. `http:/phase.hpcc.jp/Omni/`
9. POP Esprit Project IST 2001-3307: Performance Portability of OpenMP. `http://www.cepba.upc.es/pop`
10. Wolfe, M.: High performance compilers for parallel computing. Addison–Wesley Publishing Company (1995)