OpenMP: what's inside the black box?

Eduard Ayguadé, Lluc Alvarez and Fabio Banchelli
Computer Architecture Department, Universitat Politècnica de Catalunya (UPC–BarcelonaTECH)
Computer Sciences Department, Barcelona Supercomputing Center (BSC-CNS)
Barcelona, Spain
{eduard.ayguade, lluc.alvarez, fabio.banchelli}@bsc.es

***Abstract*—This paper presents the *"Implementing a minimal OpenMP runtime using Pthreads"* assignment that is offered to students of *Parallel Programming and Architectures (PAP)*, a third-year elective subject in the Bachelor Degree in Informatics Engineering at the Barcelona School of Informatics (FIB) of the Universitat Politècnica de Catalunya (UPC) - BarcelonaTech. In this assignment students open the black box behind the compilation and execution of OpenMP programs, exploring different alternatives for implementing a minimal OpenMP-compliant runtime library, providing support for both work–sharing and tasking models. In addition, the assignment allows the students to learn the basics of Pthreads in a very interesting and applied way, as well as the low–level atomic mechanisms provided by the architecture to support the required thread and task synchronisations.**

I. Motivation for this assignment

OpenMP is the de-facto standard API for programming shared–memory parallel applications in C/C++ and Fortran. From the programmers' perspective, OpenMP consists of a set of compiler directives, runtime routines and environment variables. This programming model is ideally suited for multicore and shared–memory multi–socket architectures, as it allows programmers to easily specify opportunities for parallel execution in regular (e.g. loop based) and irregular (e.g. recursively traversing dynamically created data structures) applications.

The two components that support the execution of OpenMP programs are the compiler and the runtime system. The OpenMP compiler transforms code with OpenMP directives into explicitly multithreaded code with calls to the OpenMP runtime library. The OpenMP runtime system provides routines that support thread and task management, work dispatch, thread and task synchronisation, and intrinsic OpenMP functions. Most OpenMP runtimes are implemented using Pthreads (also named POSIX threads), the threads programming interface specified by the IEEE POSIX 1003.1c standard.

II. The `miniomp` assignment

The laboratory assignment that we present as a *"peachy"* assignment consists on building a minimal OpenMP runtime system. This simplified runtime system has to support the code generated by the GNU `gcc` compiler and to correctly work as an alternative to the `libgomp` runtime system that is distributed as part of `gcc`. Figure 1 shows a flowchart with the main components involved in the assignment. The `gcc` compiler is used unmodified and students analyse how the compiler translates the different directives to invocations of services in the runtime system. The `libgomp` library is replaced by the minimal runtime library (`libminiomp`) using the `LD_PRELOAD` mechanism, making use of the library constructor and destructor mechanisms to setup the OpenMP environment necessary to execute the OpenMP programs. The implementation of the simplified runtime is based on the POSIX Pthreads standard library, which provides the basic support for thread creation and synchronisation on top of which the work–sharing and tasking models can be built. So, in addition to the internal organization of OpenMP runtime systems, the students also learn the main features of the Pthreads library and they put into practice the use of dynamically linked shared libraries.

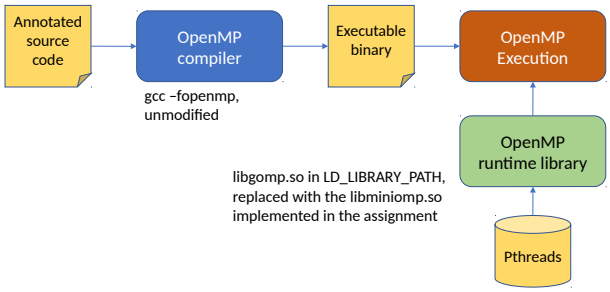

Fig. 1. The OpenMP compilation and execution flow.

In this scenario, our students usually ask the same questions: *"Why do we want to study how an OpenMP program is actually executed on our parallel machine? Is it not enough to be able to write correct OpenMP programs?, ..."*. We believe that as a programmer, this knowledge may help to understand program performance and possible sources of overhead, helping to write more efficient code. But as potential developers the assignment provides the basics to develop a parallel runtime system on top of low-level threading offered by the OS, targeting new programming models and/or multicore architectures.

III. Two alternative itineraries

In this assignment we propose two alternative itineraries that implement different functionalities of the OpenMP runtime:

- **Itinerary 1 – Basic implementation for the work-sharing model**: `parallel` regions, work distributors

This work is partially supported by the Spanish Government through Programa Severo Ochoa (SEV-2015-0493), by the Spanish Ministry of Science and Technology (contracts TIN2015-65316-P and FJCI-2016-30985), and by the Generalitat de Catalunya (contract 2017-SGR-1414).

(single and for) and synchronisations (barrier and critical, both unnamed and named).

- **Itinerary 2 – Basic implementation of the tasking model**: unified parallel-single region, task (with no support for nesting), taskloop and synchronizations (critical and taskwait).

Both itineraries are developed in an incremental way, so students can always generate a working runtime library with the functionalities implemented so far. For example, for itinerary 1 students first implement parallel and intrinsic functions omp_get_thread_num, omp_get_num_threads and omp_set_num_threads, learning how to access to environment variables such as OMP_NUM_THREADS; second they implement barrier and critical (both named and unnamed) and take a look at how the compiler implements atomic; then they implement the single work–sharing construct, paying attention to the fact that there may be multiple instances of single active at the same time; and finally they implement the for work-sharing construct.

Once the selected itinerary is implemented, students are encouraged to implement some optional functionalities. For example, in both itineraries the support for nested parallel regions, implementation of a thread pool (i.e. threads not created and finished in each parallel region) or thread binding policies are not initially considered. Also, students are allowed to initially use the synchronisation constructs provided in Pthreads (locks, barriers, ...). As part of their work, and based on their own skills, interests and time, students decide to relax some of these constraints, make use of atomic instructions to implement synchronisation primitives and/or consider some additional OpenMP features initially not considered in the specification of the assignment.

A number of simple test cases are provided with the aim of testing the functionalities incrementally implemented. For functional validation, students can always switch back to the original libgomp library to know the output that each test case should produce. We encourage students to extend the basis test suite to improve coverage and better validate the functionalities implemented. Students are also motivated to compare, in terms of performance, their implementation with the original libgomp library; although unfair, since the original libgomp library supports many more functionalities, this gives them an idea of what performance should be expected and motivates them to improve their own implementation.

IV. Resources and duration

The proposed assignment can be done in any system booted with Linux, preferably multicore. Although students could use their own laptops, we strongly suggest to use a departamental cluster composed of several nodes, each with a certain number of cores. One of the nodes is used as login node, in which students can edit and compile their code, as well as do some functional tests. The rest of nodes are accessible through execution queues and can be used for performance evaluation/comparison purposes.

To understand the behavior and the performance of their miniomp implementation students make use of Extrae [1], an instrumentation library to transparently trace MPI and OpenMP programs, and Paraver [2], a trace visualizer and analyzer that will allow students to understand the execution of parallel applications. Both are freely available at the Barcelona Supercomputing Center tools website [3].

```
miniomp                          miniomp
    src                              test
        Makefile                         Makefile
        env.{c|h}                        run-omp.sh
        intrinsic.{c|h}                  extrae.xml
        libminiomp.{c|h}                 run-extrae.sh
        parallel.{c|h}                   tparallel.c
        synchronization.{c|h}            tsynch.c
        single.{c|h}                     tworkshare.c
        loop.{c|h}                       ttask.c
        task.{c|h}                       ttaskloop.c
        taskloop.c                       tsynchtasks.c
        tasksync.c                       mandelbrot-gui.h
    lib                                  tmandel1.c
        libminiomp.so                    tmandel2.c
    …                                    tnested.c
```

Fig. 2. Directory structure for the miniomp assignment.

In order to have an idea of the number of files that need to be modified, Figure 2 shows the directory tree and files initially provided to students; a dummy implementation for all runtime functions that need to be implemented is provided. In terms of duration, the assignment is designed to be done in 6 laboratory sessions (one per week), each taking 2 hours; in addition, students usually spend between 2 and 4 additional hours at home per week. So the total load is estimated between 24 and 36 hours, depending on how far students go with optional parts, functional validation and performance evaluation and comparison against the original libgomp.

The assignment has been tested (and evolved) in the past 3 academic years, making it readily adoptable by other educators.

References

[1] Barcelona Supercomputing Center, *Extrae User Guide Manual. Version 2.2.0*, 2011.

[2] V. Pillet, J. Labarta, T. Cortes, and S. Girona, "Paraver: A Tool to Visualize and Analyze Parallel Code," in *Proceedings of WoTUG-18: transputer and occam developments*, vol. 44, no. 1, 1995, pp. 17–31.

[3] https://tools.bsc.es/downloads.