PAP laboratory assignments
Lab 2: Implementing a minimal OpenMP runtime

E. Ayguadé

Spring 2018-19

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Contents

The environment

In this laboratory assignment you will build a simplified `OpenMP` runtime system, that should give support to the code generated by the GNU `gcc` compiler, alternative to the `gomp` runtime system that is distributed as part of the `gcc` distribution. Your implementation will be based on the POSIX Pthreads standard library which will provide you the basic support for thread creation and synchronisation. The new created library will be named `miniomp` (`libminiomp.so`).

Two alternative itineraries are proposed, each one implementing different functionalities of the runtime:

- **Itinerary 1 – Implementation of the worksharing model**: `parallel` regions, work distributors (`single` and `for`) and synchronizations (`barrier` and `critical`).
- **Itinerary 2 – Implementation of the tasking model**: unified `parallel-single` region, `task`, `taskloop`, and synchronizations (`critical` and `taskwait`).

In each itinerary optional components are possible, to be presented in each chapter.

All files necessary to do this laboratory assignment are available in `/scratch/nas/1/pap0/sessions`. Copy `lab2.tar.gz` inside the `sessions` directory in your home directory in `boada.ac.upc.edu` and uncompress it with this command line: `"tar -zxvf lab2.tar.gz"`. Don't forget to parse `"environment.bash"` in order to set all necessary environment variables before continuing. Inside the `miniomp` directory that you just uncompressed you will find three directories:

- `src` with a set of files where you will implement the functionalities requested for the runtime (most of them are empty at this moment or just contain the prototypes for the functions to be implemented during the sessions devoted to this laboratory assignment). The directory also includes a `Makefile` to compile the library.
- `lib` where the compiled `libminiomp.so` library will be generated.
- `test` with some simple codes to benchmark the library at the different stages of its implementation. The directory also contains a `Makefile` to compile them and scripts to execute and instrument their parallel execution. You should extend this basic set in order to make sure that you appropriately test all functionalities requested.

Next proceed to do the following steps in order to check that everything is appropriately setup:

1. Go into `src` inside the `miniomp` directory, list the existing files to get familiar with their names and type `"make libminiomp.so"`. This should compile the `miniomp` library in its current implementation status and generate a `.so` file in `lib`. No errors should be reported by now.
2. Next go to the `test` directory and compile the first OpenMP benchmark code by typing `"make tparallel-omp"` and execute it by typing `"OMP_NUM_THREADS=4 ./tparallel-omp"` or simply using the provided script file `"./run-omp.sh tparallel-omp 4"`. Check that the result is the expected one by inspecting the source code of `tparallel.c` and verifying the correctness of the output displayed.
3. You can also check if the functionality of `miniomp` conforms to the original `gomp`. Type `"make tparallel-gomp"` to compile and `"OMP_NUM_THREADS=4 ./tparallel-gomp"` or `"./run-omp.sh tparallel-gomp 4"` to execute using the original `gcc` library. Check if the result displayed is the same as with `miniomp`.

4. Of course you can (I mean, should) generate *Extrae* traces for *Paraver* visualisation. Just use the `run-extrae.sh` script with the same arguments to execute the program and generate the trace and then visualise with `wxparaver`.

The paths defined in `environment.bash` assume that the `miniomp` directory has been uncompressed from your home directory, i.e. in `/scratch/1/papXX/miniomp`. We recommend that you follow this unless you want to set up the `MINIOMP` and `LD_LIBRARY_PATH` environment variables in a different way.

As a standard library, `libminiomp.so` includes two functions to initialise and finalise its execution, both defined in `libminiomp.c`:

- `void init_miniomp(void) __attribute__((constructor));`
- `void fini_miniomp(void) __attribute__((destructor));`

Once the constructor has been executed, the `main()` function in the OpenMP application will be executed. Once `main()` exits, the destructor will be executed and close the library.

In the next chapters we proceed with a description of the minimal functionalities that we require in your implementation of `miniomp`. As a way to work the:

"Actitud adecuada devant el treball competence, and in particular *G8.3 – Estar motivat pel desenvolupament professional, per a afrontar nous reptes i per la millora continua. Tenir capacitat de treball en situacions de falta d'informacio."*

you can optionally relax some of the constraints in the initial specification, do the optional parts, or even do the two itineraries ;) Good luck!

:

Itinerary 1

The worksharing model

1.1 Parallel regions

In this first part of the itinerary you will perform a restricted implementation of the `parallel` construct in OpenMP:

- No thread pool: threads are created and finished in each `parallel` region.
- Single-level of parallelism, i.e. no support for nested `parallel` regions.
- Correct format in the specification of the `OMP_NUM_THREADS` environment variable.
- No support for thread binding policies (`proc_bind` clause and `OMP_PROC_BIND` environment variable).

It is optional (but highly recommendable to attempt one or more towards a good mark in the transversal competence) to relax these restrictions, see section 1.4 for options. You will also implement the `omp_get_thread_num` intrinsic function, which, as you know, returns a unique identifier for the invoking thread between 0 and `omp_get_num_threads()-1`.

The files that you need to look at or modify in this section are: `libminiomp.c` and `libminiomp.h` which contain the constructor and destructor of the library and the declaration of some data types and global variables, respectively; `env.c` and `env.h` which include data definition and functionality to parse the `OMP_NUM_THREADS` environment variable; `intrinsic.c` and `intrinsic.h` which contains some OpenMP intrinsic functions related with the parallel region; and `parallel.c` and `parallel.h` which contain the data definitions and naïve implementation of the function invoked by `gcc` to implement `parallel` regions. As you will see, some of the functions are already implemented and the main types for global variables are (partially) defined; the rest should be developed using the Pthreads API for thread management and access to thread–specific data.

In order to see which functions from the original `gomp` library are invoked by `gcc`, we can take a look at the assembly code generated by the compiler. Go into the `test` directory and type `"make tparallel-asm"`. Open with the editor the `tparallel-asm` generated and look for function invocations starting with `GOMP`. The usual interface used by `gcc` to activate a parallel region is:

```
void GOMP_parallel (void (*fn) (void *), void *data, unsigned num_threads, unsigned int flags);
```

which receives the pointer `fn` to the function that encapsulates the body of the parallel region, a pointer `data` to a structure used to communicate data in and out of that function to be executed by each thread. The number of threads `num_threads` is 1 if an `if` clause is present and false, or the value of the `num_threads` clause, if present, or 0; `flags` is related with the `proc_bind` clause not to be implemented. Please also take a look at how the compiler encapsulates the bodies of the parallel sections into functions called `foo._omp_fn.0`, `foo._omp_fn.1`, `foo._omp_fn.2` and `foo._omp_fn.3`, trying to understand the assembly to match them to the C code.

What to do?

1. Do the implementation of `GOMP_parallel`. It is important to look at the types and variables definition in both the `.c` and `.h` files; they are there to guide you in your implementation. Don't forget to implement the implicit barrier at the end of the parallel region, inside `GOMP_parallel`.

2. Do the implementation of the intrinsic function `omp_get_thread_num`.

How to test your implementation? To test your implementation please use the OpenMP program in file `tparallel.c`. As done before in the previous chapter, check if the output of the program is what you would expect and check if the functionality of `miniomp` conforms to `gomp`. We recommend to also test your implementation running the *Extrae* instrumented version of your test programs and visualise the traces generated with *Paraver* (use the `./run-extrae.sh` script and `wxparaver` for this purpose).

1.2 Synchronisations: `barrier`, `critical` and `atomic`

Second you will perform a restricted implementation of two of following synchronisation constructs in OpenMP, using the mechanisms offered by *Pthreads*:

- Explicit `barrier` (optionally, you can do your own implementation of the barrier construct not using the mechanism offered by *Pthreads*.
- Unnamed `critical` regions (i.e. no `name` provided in the `critical` directive).
- Named `critical` regions (i.e. with a `name` provided in the `critical` directive).

The `atomic` construct is handled by the compiler directly generating machine instructions preceded by the `lock` prefix which forces the memory access to be atomically executed.

The files that you need to look at or modify in this section are: `libminiomp.c` and `libminiomp.h` which should contain the allocation/initialization of the synchronization objects implemented in this section; and `synchronization.c` and `synchronization.h` which contain the data definitions and prototypes for the functions invoked by `gcc` to implement unnamed `critical` sections and explicit `barrier` constructs. The declaration of the global variables to implement them (`miniomp_default_lock` and `miniomp_barrier`) are included in `synchronization.h`.

In order to see which functions from the original `gomp` library are invoked by `gcc`, we can take a look at the assembly code generated by the compiler for `tsynch.c`. Open with the editor the `tsynch-asm` generated in your last compilation and look for function invocations starting with `GOMP`. As you can see, the interface used by `gcc` to enter to and exit from an unnamed `critical` section is as follows:

```
void GOMP_critical_start (void);
void GOMP_critical_end (void);
```

while for the implementation of the named section the compiler includes one argument that points to a `void *` associated to the name that is provided by the programmer in the pragma.

```
void GOMP_critical_name_start (void **pptr);
void GOMP_critical_name_end (void **pptr);
```

How is the compiler defining memory space for the pointer associated to the name?

For the implementation of a barrier the compiler simply injects a call to:

```
void GOMP_barrier(void);
```

All the initialisation for to properly execute the barrier should be done somewhere in your code before the invocation to `GOMP_barrier`.

Finally, verify in the assembly code how the compiler performs the translation for the `atomic` construct, looking for additional information about the `lock` prefix.

What to do?

1. Do the implementation of `GOMP_critical_start`, `GOMP_critical_name_start`, `GOMP_critical_end` and `GOMP_critical_name_end` using Pthread mutexes.
2. Do the implementation of `GOMP_barrier` using Pthread barriers.

How to test your implementation? To test your implementation please use the OpenMP program in file `tsynch.c`. As done before in the previous section, check if the output of the program is what you would expect and check if the functionality of `miniomp` conforms to `gomp`. We recommend to also test your implementation running the *Extrae* instrumented version of your test programs and visualise the traces generated with *Paraver*.

1.3 Work distributors: `single` and `for`

Next you will perform a restricted implementation of two of the worksharing constructs in OpenMP:

- `single` construct.
- `for` construct with `static` and `dynamic` schedules. No support for the `guided` and `runtime` schedules nor for the `ordered` clause.

Your implementation has to consider the possibility of including the `nowait` clause, which implies that multiple worksharing constructs may be active at a time.

In addition to the files that you already know, the files that you will need to look at or modify in this section are: `single.c` and `loop.c` which contain the prototypes for the functions invoked by `gcc` to implement `single` and `for` constructs, respectively. The declaration of the global variables to implement them (`miniomp_single` and `miniomp_loop`) should be completed in `single.h` and `loop.h`.

In order to see which functions from the original `gomp` library are invoked by `gcc`, we can take a look at the assembly code generated by the compiler for `tworkshare.c`. Open with the editor the `tworkshare-asm` generated in your last compilation and look for function invocations starting with `GOMP`. The following function is used to implement `single`:

```
bool GOMP_single_start (void);
```

which is called by all threads encountering the `single` but returns true only for the thread that should execute the body of the `single` contract, i.e. the first one reaching it. Looking at the assembly code, you can see how the result of the function is used.

For the implementation of `for` the compiler is using the following functions:

```
bool GOMP_loop_dynamic_start (long start, long end, long incr, long chunk_size,
                         long *istart, long *iend);
bool GOMP_loop_dynamic_next (long *istart, long *iend);
void GOMP_loop_end (void);
void GOMP_loop_end_nowait (void);
```

For example, the following `for` loop:

```
#pragma omp for schedule(dynamic, 10)
for (long i = 0; i < n; i++)
    body;
```

is translated by gcc in a code similar to the following one:

```
long i, _s0, _e0;
if (GOMP_loop_dynamic_start (0, n, 1, 10, &_s0, &_e0)) // all threads invoke it,
                                                       // returns true if there is work
    do {
        for (i = _s0, i < _e0; i++)
             body;
    } while (GOMP_loop_dynamic_next (&_s0, _&e0));     // Returns true if there is work
GOMP_loop_end (); // invoked after the thread is told that all iterations are complete
```

The first thread invoking the `_start` function should initialise the work descriptor for the loop and get a chunk of iterations; the other threads should just get a chunk of iterations after registering to the worksharing. The `_next` function simply returns a new chunk of iterations to the invoking thread. The `_end` functions are executed by each thread when completing the last chunk assigned, with or without the implicit barrier at the end. A brief description of each function, its arguments and result are available inside the `loops.c` file.

For the `static` schedule the compiler generates code so that there is no need to invoke the `OpenMP` runtime system to distribute loop iterations. The only runtime function that is invoked is the `GOMP_barrier` in case the `for` construct has an implicit barrier at the end.

What to do?

1. Do the implementation of `GOMP_single_start`.
2. Do the implementation of the functions required to implement the `OpenMP for` construct with `dynamic` schedule.

Both constructs may include the `nowait` clause, bypassing the implicit barrier at the end of the worksharing. This

How to test your implementation? To test your implementation please first use the OpenMP program in file `tworkshare.c`. As done before in the previous section, check if the output of the program is what you would expect and check if the functionality of `miniomp` conforms to `gomp`.

To make a more extensive testing of your implementation, you can use `tmandel1.c` which contains the parallelisation of the Mandelbrot set that you did in PAR; in this version, the computation is parallelised by rows (you can try with other schemes and see if your implementation has any limitation). You can also use your worksharing versions of the *Erathostenes* `sieve` program that you did in the first laboratory assignment.

1.4 Optional implementations

Finally, although optional but highly recommended towards a high mark in the transversal competence, we propose to extend your implementation in order to relax some of the constraints initially defined and/or to consider some additional features initially not considered. For example:

- Implementation of a thread pool, so that threads are not created and finished in each `parallel` region.
- Implementation of your own barrier construct, using the atomic intrinsic operations available for the `gcc` compiler.
- Nested `parallel` regions: basically, each parallel region is totally independent, with its own barriers, worksharing constructs and number of threads. The only construct that is global to all parallel regions is `critical`. Extend your implementation to consider test programs such as `tnested.c`.
- Adding thread to processor affinity in your implementation of `parallel`, so that threads are mapped to processors in a fixed way (cpus inside socket or socket).

Itinerary 2

The tasking model

2.1 Parallel regions

In this first part of the itinerary you will perform an implementation of the `parallel` construct in OpenMP that is tailored to the tasking model. In the proposed implementation, only one of the threads executes the body of the parallel region; the other ones just wait at the implicit barrier at the end of the parallel region waiting for the availability of tasks to execute. In this way it is equivalent to the combination of the `parallel` and `single` constructs that is usually done when using tasks. Your implementation will be simplified in the sense that:

- Threads are created and finished in each `parallel` region (i.e. no thread pool).
- Single-level of parallelism (i.e. no support for nested `parallel` regions).
- Correct format in the specification of the `OMP_NUM_THREADS` environment variable.
- No support for thread binding policies (`proc_bind` clause and `OMP_PROC_BIND` environment variable).

You will also implement the `omp_get_thread_num` intrinsic function, which, as you know, returns a unique identifier for the invoking thread between 0 and `omp_get_num_threads()-1`.

The files that you need to look at or modify in this section are: `libminiomp.c` and `libminiomp.h` which contain the constructor and destructor of the library and the declaration of some data types and global variables, respectively; `env.c` and `env.h` which include data definition and functionality to parse the `OMP_NUM_THREADS` environment variable; `intrinsic.c` and `intrinsic.h` which contains some OpenMP intrinsic functions related with the parallel region; and `parallel.c` and `parallel.h` which contain the data definitions and naïve implementation of the function invoked by `gcc` to implement `parallel` regions. As you will see, some of the functions are already implemented and the main types for global variables are (partially) defined; the rest should be developed using the Pthreads API for thread management and access to thread–specific data, if needed.

In order to see which functions from the original `gomp` library are invoked by `gcc`, we can take a look at the assembly code generated by the compiler. Go into the `test` directory and type `"make tparallel-asm"`. Open with the editor the `tparallel-asm` generated and look for function invocations starting with `GOMP`. The usual interface used by `gcc` to activate a parallel region is

```
void GOMP_parallel (void (*fn) (void *), void *data, unsigned num_threads, unsigned int flags);
```

which receives the pointer `fn` to the function that encapsulates the body of the parallel region, a pointer `data` to a structure used to communicate data in and out of that function to be executed by each thread. The number of threads `num_threads` is 1 if an `if` clause is present and false, or the value of the `num_threads` clause, if present, or 0; `flags` is related with the `proc_bind` clause not to be implemented. Please also take a look at how the compiler encapsulates the bodies of the parallel sections into functions called `foo._omp_fn.0`, `foo._omp_fn.1`, `foo._omp_fn.2`, `foo._omp_fn.3` and `foo._omp_fn.4`, trying to understand the assembly to match them to the C code.

What to do?

1. Do the alternative implementation of `GOMP_parallel` tailored to the tasking execution model. At this stage threads waiting at the implicit barrier never see a task to execute; so we recommend that in your current implementation those threads occasionally print a `*`, or even better their identifier (between 0 and number of threads minus one), in order to check that they are alive willing to execute work. Those threads should stop waiting as soon as the master arrives to the implicit barrier too.

2. Do the implementation of the intrinsic function `omp_get_thread_num` which, as you know, returns a unique identifier for the invoking thread between 0 and `omp_get_num_threads()-1`.

How to test your implementation? To test your implementation please use the OpenMP program in file `tparallel.c`. Observe that you can not compare the results printed with `miniomp` with the results printed when using `gomp`.

2.2 Task construct

Second you will perform a restricted implementation of the tasking model in OpenMP, which should include:

- Tied `task` with no `depend` clauses.
- No nesting of `task` constructs and no `if` and `final` clauses.

In addition to the files you already know, the files that you need to look at or modify in this section are: `task.c` and `task.h` which contain the prototypes for the functions invoked by `gcc` to implement `task`, as well as the declaration of the main global variable to implement them: `miniomp_taskqueue`.

In order to see which functions from the original `gomp` library are invoked by `gcc`, we can take a look at the assembly code generated by the compiler for `test5.c`. Open with the editor the `test5-asm` generated when typing `"make test5_omp` and look for function invocations starting with `GOMP`. The following function is invoked every time a task is found:

```
void GOMP_task (void (*fn) (void *), void *data, void (*cpyfn) (void *, void *),
           long arg_size, long arg_align, bool if_clause, unsigned flags,
           void **depend, int priority);
```

which you can easily match to the original OpenMP directives. For the `GOMP_task` we will only consider the arguments related with the pointer `*fn` to the function that encapsulates the body of the task and the pointer `*data` to the arguments needed to execute it. The compiler may provide a helper function `*cpyfn` and some additional arguments to correctly access them (`arg_size` and `arg_align`).

I order to implement the task queue we provide the prototypes for some additional functions, whose functionality is the following:

```
miniomp_taskqueue_t *init_task_queue(int max_elements); // Initialises the task queue
bool is_valid(miniomp_task_t *task_descriptor); // Checks if the task descriptor is valid
bool is_empty(miniomp_taskqueue_t *task_queue); // Checks if the task queue is empty
bool is_full(miniomp_taskqueue_t *task_queue); // Checks if the task queue is full
bool enqueue(miniomp_taskqueue_t *task_queue, miniomp_task_t *task_descriptor); // Enqueues
                                  // the task descriptor at the tail of the task queue
bool dequeue(miniomp_taskqueue_t *task_queue); // Dequeue the task descriptor at the head
                                  // of the task queue
miniomp_task_t *first(miniomp_taskqueue_t *task_queue); // Returns the task descriptor at the
                                  // head of the task queue
```

You may change them or build your own data structure for the task queue.

What to do?

1. Do the implementation of restricted `GOMP_task`.

2. Update the implementation of the implicit barrier at the end of `parallel` so that threads arriving there can grab tasks from the task queue if tasks are available for execution. Threads should leave the barrier once all threads have arrived to it and all tasks in the task pool have already been executed.

How to test your implementation? To test your implementation please use the OpenMP program in file `ttask.c`. Observe that the result is not deterministic since there is a race condition in the code that can not be solved until we implement one of the two task synchronisation alternatives in the next section.

2.3 Taskwait and taskgroup synchronisations

Third you will perform an implementation of the `taskwait` construct. Since our implementation is not supporting nested tasks, the implementation of `taskgroup` is the same as taskwait, with the only difference of just waiting for those tasks created within the body of the `taskgroup` region.

In order to see which functions from the original `gomp` library are invoked by `gcc`, we can take a look at the assembly code generated by the compiler for `test6.c`. Open with the editor the `test6-asm` generated when typing `"make test6_omp` and look for function invocations starting with `GOMP`. The following two functions are used:

```
void GOMP_taskwait (void);
void GOMP_taskgroup_start (void);
void GOMP_taskgroup_end (void);
```

which you can easily match to the original OpenMP directives.

What to do?

1. Do the implementation of `GOMP_taskwait`, `GOMP_taskgroup_start` and `GOMP_taskgroup_end`.

How to test your implementation? To test your implementation please use the OpenMP program in file `tsynchtasks.c` which exercises the use of the task synchronisation constructs.

You can also check your implementation of `task` and `taskwait` using `tmandel2.c`, which contains the parallelisation of the Mandelbrot set that you did in PAR; in this version, the computation is parallelised by rows using tasks (you can try with other schemes and see if your implementation has any limitation). You can also use your tasking version of the *Erathostenes* `sieve` program that you did in the first laboratory assignment.

2.4 Optional: taskloop construct

Finally, although optional but highly recommended towards a high mark in the transversal competence, we propose to perform a simplified implementation of the `taskloop` construct.

- No nesting of `taskloop` constructs.

- No `if` and `final` clauses.

The `gcc` compiler invokes the following function to implement the `taskloop` functionalities:

```
void GOMP_taskloop (void (*fn) (void *), void *data, void (*cpyfn) (void *, void *),
               long arg_size, long arg_align, unsigned flags,
               unsigned long num_tasks, int priority,
               long start, long end, long step)
```

Most of the arguments are similar to the arguments of `GOMP_task`, the different ones are briefly described next. `num_tasks` which is used to indicate either the number of tasks to generate (if `num_tasks` clause is used) or the granularity of the tasks to generate (if the `grainsize` clause is used); one of the bits in `flags` is used to indicate which of the two options is applied. Arguments `start`, `end` and `step` capture the iteration bounds of the loop to which the `taskloop` construct applies. You can check all this by looking at the assembly code generated for `ttaskloop.c`.

What to do?

1. Do the implementation of restricted `GOMP_taskloop`. Your implementation should handle the possibility of specifying one of `num_tasks` or `grainsize`, or none of them in which case the number of tasks should be the number of threads in the parallel region.

How to test your implementation? To test your implementation please use the OpenMP program in file `ttaskloop.c`. You should also use the version of the *Erathostenes* `sieve1.c` program that you did in the first laboratory assignment using `taskloop`.