
You can find the code snippets referred to in the assignments via the embedded and attached files for this PDF.

Further reading:

- Man-pages of GDB, GCC, ...
- https://www.openmp.org/spec-html/5.2/openmp.html
- https://hpc-tutorials.llnl.gov/openmp/
- https://hpc.gwdg.de/hpc/

Contents

Task 1: Tutorial: Reduction Clause (30 min)

We will work through the first 4 tasks together and discuss the results of the remaining tasks after 30 minutes.

The file `e1_1.c` contains the computation of a sum.

Perform the following tasks:

1. Parallelize it by adding the OpenMP `parallel for` clause
2. Compile it with and without OpenMP and compare the output
3. Try to repair the sum calculation by using a shared variable
4. Check the correctness of the algorithm - it doesn't work, why?
5. Use the atomic pragma to repair the code
6. Check the correctness

7. Use the reduction pragma instead

Hints

- Compile using `gcc -fopenmp -o e1_1 e1_1.c`

Task 2: Basic OpenMP Directives (30 min)

The file `e1_2.c` contains blocks/instructions marked as Snippets in comments. Perform the following tasks:

1. Compile the code and investigate the output.
2. Parallelize the "Hello World" message in *Snippet#1* by using the `parallel` directive.
3. How many threads are running your code? What's the default number of threads? How can you change this? Can you start more threads than there are physical processors?
4. Compile the code again with and without the openmp flag and compare the output for both programs. (you may need to add -lgomp at the end of the command line for this part)
5. Parallelize the vector multiplication code in *Snippet#2* by adding the `parallel` directive in combination with the `critical` directive. Use the latter to serialize access to the addition operation over the `sum` shared variable.
6. Parallelize the vector multiplication code in *Snippet#3* by adding a `parallel` directive with `reduction` clause.
7. Investigate the `multiply()` function - create an alternative function that allows the use of the *schedule* directive.

Hints

- Look into the `omp_get_schedule function`

Task 3: OpenMP Sections (30 min)

The program in `e1_3.c` uses OpenMP `sections` for MIMD work sharing. Most of the time it does not exit cleanly (it hangs) but there might also be another bug.

Try to repair the program. To do so:

- Modify the code to output proper debug messages.
- Identify the root cause of the bug.
- Modify the code to work properly.

Hints

- You may want to play with the code.
- What is the problem with the array `c[]`?

- Are all barriers necessary?

Task 4: Parallelizing Matrix Multiplication (30 min)

The file `e1_4.c` contains code that performs multiplication of two square matrices (say A and B) with varying sizes.

1. Using OpenMP, parallelize the loop that performs the multiplication such that the computation of rows of the product matrix is parallelized among threads. Do not use default variable scope, instead use `private` and `shared` clauses explicitly.
2. Add a way to distinguish between different threads on the line marked with "DISTINGUISH HERE".
3. Add a way to calculate the total execution time of the matrix multiplication using `omp_get_wtime()`. Give the execution time in seconds!
4. Add a way to calculate the execution time for each thread separately. What do you notice?
5. Perform the multiplication of square matrices of varying sizes (say 2x2, 4x4, ... , 1024x1024..) with a varying number of threads up to 4096. Plot a graph showing the dependency between the execution time and the number of threads for a 1024x1024 matrix. What behavior does it show?

Hints

- **Make sure that you don't allocate more memory than your computer has!**
  Which size of matrix fits at maximum in 1/4 of your computers main memory? You might look it up with the linux command `free`.
- **Make sure that you don't create too many threads!**

Task 5: Data/Loop Dependence (20 min)

The file `e1_5.c` contains code that uses an array x of fixed size to store the factorial of i in an array x[i].

1. What is the difference between
   - `#pragma omp parallel`,
   - `#pragma omp for` and
   - `#pragma omp parallel for`?
2. What is the problem of data/loop dependence in the example?
3. Run the `parallel for` directive with more than 1 thread. What might go wrong?
4. How can you fix this without modifying the computation logic, i.e., using OpenMP features?
5. Modify the program logic such that the program behaves correctly with multiple threads.
6. Identify how the iterations are divided among threads. Use the `schedule` clause and try two alternative schedules.
7. Discuss in your group: Is there any benefit of parallelizing this algorithm using OpenMP?

Task 6: Error Fixing (15 min)

The file `e2_1.c` contains simple code snippets each containing a different error. Fix the errors.

Task 7: Synchronization Clause (10 min)

1. Describe the difference between the `critical` and the `atomic` clause.
2. Describe the `firstprivate` clause.

Task 8: Partial Differential Equation (60 min)

This task is optional and you may not have time to complete it - that is OK. You can optionally complete it for your own training at home.

The file `partial.c` contains the basics for solving the time dependent partial equation also called the heat equation. It uses a 3rd order Runge-Kutta time stepping scheme.

Your goal is to incrementally improve the performance.

Approach the problem as follows:

- Incrementally increase the performance using OpenMP.
- After each modification, verify that the output is correct (and the same as the initial version) after non-trivial modifications.
- You may want to reduce the verbosity of the output.
- Identify what can be parallelized.
- Summarize your conclusions.

After compiling the code you can call it with two arguments, e.g., `partial 100 0.01`, where the first parameter gives you the number of grid points and the second the length of the timestep. When using more grid points you have to use a smaller time step. If the time step is too big you will get NaN.

Hints

- To compile the code, run `$ gcc partial.c -o partial -O3 -lm`