HIGH PERFORMANCE PROGRAMMING
UPPSALA UNIVERSITY
SPRING 2023
LAB 11: OPENMP (PART II)

This lab gives an introduction to shared-memory parallel programming using OpenMP.

The following aspects are included in the lab:

(1) Private vs shared variables in OpenMP parallel code blocks

(2) The "`#pragma omp sections`" directive

(3) The "`#pragma omp for`" directive

(4) The "`schedule`" clause used with "`#pragma omp for`"

(5) The "`reduction`" clause

(6) "*False sharing*" – performance problems related to cache coherence issues

(7) Task-based parallelism

Note about compiler optimization flags: As you know, our end goal in this course is to achieve the best performance we can. However, some of the example codes in this lab are so simple that the compiler may just optimize away everything if we turn on compiler optimizations. Therefore, for the purposes of this lab, it is OK to compile without compiler optimization. Just remember that in real cases, and in assignments in this course, you should aim for the best possible performance and then of course use both compiler optimization flags and your own code optimizations and parallelization efforts.

1. Main Tasks

Task 0

The code for this task is in the `Task-0` directory.

Here we will take a look at an issue with how valgrind works for OpenMP programs. The code given in `smallcode.c` is a very simple example of a program with a loop parallelized using OpenMP.

First, compile it without the `-fopenmp` flag, and run it with valgrind. Since the program has no memory leaks, this should give the "no leaks are possible" message. As noted previously in this course, it is good practice to always check your programs using valgrind, and verify that you get the "no leaks are possible" message.

Date: February 12, 2023.

Unfortunately, when a program is using OpenMP some memory used by the OpenMP implementation is not freed, so for OpenMP programs valgrind reports some "possibly lost" and somde "still reachable" blocks under "LEAK SUMMARY". Compile the `smallcode.c` program with the `-fopenmp` flag and run it with valgrind, to see what this looks like.

So, because of this behavior of OpenMP programs, we can not expect to get the "no leaks are possible" message from valgrind even if our program is correct. However, valgrind is still very useful for finding memory errors, we just have to look more closely at its output regarding memory leaks and pay attention to any "definitely lost" blocks reported under "LEAK SUMMARY". Just keep this in mind when using valgrind for OpenMP programs; some reports of "possibly lost" and "still reachable" blocks may be due memory allocations within the OpenMP implementation that you cannot do anything about.

For some further explanation of this issue, see the comment by Jakub Jelinek (one of the gcc developers) here:

https://gcc.gnu.org/bugzilla/show_bug.cgi?id=36298

Task 1

The code for this task is in the `Task-1` directory.

In an OpenMP program, by default all variables declared outside a parallel code block are shared, i.e., accessible by all threads. When necessary, variables can be made private to the threads by using the `private` clause. Then these variables are reallocated on the stack, which is private for each thread. Note that each thread then gets its own private copy of the variable, allocated on that thread's own stack.

The program in `datasharing.c` has both shared and private variables. Look at the code. Can you predict its output?

Compile and run the program. Is the output what you predicted? Is it at all possible to predict the output of this program as it is written now?

All private variables are uninitialized when entering the parallel region. However, in OpenMP these can be initialized from the original variable by using the `firstprivate` clause. Modify `datasharing.c` to make the variable `a` firstprivate. Compile and run, is the output now what you expect?

In OpenMP we also have a clause `lastprivate`, what is the purpose of this clause and when is it useful?

Task 2

The code for this task is in the `Task-2` directory.

Here we will have a look at the `sections` directive. Look at the code in `ompsections.c` and compare to the information about the `sections` directive for example in the OpenMP "Summary Card" linked to in the reading list in Studium. Can you understand what the program is doing?

Compile and run the program, and check the real and user time spent using the `time` command and/or `top`. Does the program work as you expected?

Also try changing the number of threads used from 2 to other values. Check how the program behaves when you set the number of threads to 1, and when you set it to 3. Do you understand what is happening?

Next, modify the code so that you add one more "`#pragma omp section`" with a call to some other function that you create, e.g. "`funcC()`" doing a similar amount of work as the other functions. How does this affect the wall time it takes to run the program when using 2 threads? What about if you use 3 threads, what timing do you get then? Can you understand why?

Task 3

The code for this task is in the `Task-3` directory.

This task is about the "`#pragma omp for`" directive. Look at the code in the file `simple-loop.c` and try to understand what it is doing. If we ignore the `#pragma omp` directives (as the compiler will do if OpenMP is not enabled) then we see that the code is just using a simple loop to compute some values in the array `arr`. Each loop iteration is completely independent from the others, so it is an easy case for parallelization. The "`#pragma omp parallel`" directive means that a given number of threads will be created, and that team of threads will execute the following block of code. Inside the parallel block is the loop, and there the "`#pragma omp for`" directive means that the loop iterations should be divided among the threads.

First compile and run the code without using OpenMP (skip the `-fopenmp` flag). Note the time it takes and the result printed.

Then compile the code with `-fopenmp` and run it again. Is the result still correct? What happens with the time? Try with different numbers of threads – does the program behave as you expect?

Now this code is written using two separate OpenMP directives:
"`#pragma omp parallel`" followed by "`#pragma omp for`".
OpenMP provides a more convenient way of doing this using just a single directive, the "`#pragma omp parallel for`" directive. Change the code so that this is

used instead. Does it work? Note that the `num_threads` clause can be used with "`#pragma omp parallel for`" in the same way as with "`#pragma omp parallel`".

Note that the "`#pragma omp for`" and "`#pragma omp parallel for`" directives will only work properly when the loop iterations are independent. It is the programmer (you!) who is responsible for checking that this is really the case, to make sure that you only use these directives when loop iterations are independent.

Look at the code in `other-loop.c`. It is similar to `simple-loop.c`, the function `f()` is the same, but now the loop in main() looks different: there is a variable `y` used in the loop, and the values returned by `f()` are multiplied by `y` before being stored in the array. Are the loop iterations independent in this case?

Now suppose the programmer was careless and did not notice the dependency between loop iterations in `other-loop.c` but just went ahead and inserted a directive "`#pragma omp parallel for`" in the same way that worked so well in the `simple-loop.c` case. Try this, compile and run the code and see what happens. Do you get the correct result? Is the result the same every time you run the program? Do you understand why it behaves like this?

Note that the variable `y` is declared before the parallel block, so it will be a shared variable. Different threads will be trying to use it at the same time.

Task 4

The code for this task is in the `Task-4` directory.

Here we have another example of "`#pragma omp for`" usage. Look at the code in `loop.c` to understand what it is doing. Is it OK to use the "`#pragma omp for`" directive in this case – are the loop iterations independent?

Note that the function `work()` that is called in each loop iteration here is such that it takes a longer time to run the larger the input argument `c` becomes. Compile and run the program for different numbers of threads. Does it give correct results when using several threads? What speedup do you get?

Because of how the `work()` function looks in this case, the default way that the directive "`#pragma omp for`" splits up the work may not give good performance – we get a *load balancing* problem. In this situation, the `schedule` clause can be used to specify a different scheduling. Try these different variants:
`schedule(static, chunk_size)`
`schedule(dynamic, chunk_size)`
`schedule(guided, chunk_size)`
where "`chunk_size`" is the number of loop iterations that are lumped together. Which scheduling choice seems to give the best performance in this case?

Note about default "chunk size": if no `chunk_size` is specified, the default value used is different for the different scheduling policies. For the `static` case, the default is to use as large chunk size as possible, giving approximately the same

number of iterations to each thread. For the `dynamic` and `guided` cases, the default chunk size is 1.

Task 5

The code for this task is in the `Task-5` directory.

If we want to compute the sum of all elements in an array, e.g. computing the norm, we can parallelize the computations with a "`#pragma omp for`" directive and compute partial sums in parallel. These local sums must then be added to a global sum. To avoid simultaneous updates of the global sum we can use the "`#pragma omp critical`" directive which allows only one thread at a time to update the shared variable.

Look at the code in `reduce.c` which is using the above approach to sum up the elements of an array. The outer "`repeat`" loop is only to let the work take longer so that we get a bit longer and more reliable timings. Look at the code and make sure you understand what it is doing. Note that in this program a combination of private and shared variables is used: the `sum` variable is private to each thread, while the `globsum` variable is shared.

Run the program using different numbers of threads, check that the result is correct and investigate the speedup. Does it work as you expected?

The computation of a global sum can however be done simpler, OpenMP provides support for global reduction operations through the `reduction` clause used with the `omp for` directive. Modify the program in `reduce.c` to use the `reduction` clause. The code should become shorter now; you no longer need to use the "`#pragma omp critical`" directive, and the private `sum` variable is also not needed, you can work directly with `globsum`. Compile, run and verify the correctness of your program. Does it work as expected? Are you now able to parallelize the program by adding just one `#pragma` line in the code?

Task 6

The code for this task is in the `Task-6` directory.

Now we return to the example of computing π using numerical integration, that you parallelized using Pthreads in Lab 9. Now we will see how this can be done using OpenMP.

Parallelize the code in `pi.c` using appropriate OpenMP directives. Remember to check that the result is correct. What speedup do you get? Are you able to perform this parallelization more conveniently, using fewer code changes, with OpenMP compared to the Pthreads case?

Task 7

The code for this task is in the `Task-7` directory.

Here we will look at an example of "*false sharing*" — performance problems caused by different threads writing to memory locations that are separate but so close that they may be in the same cache line. The CPU hardware may then be forced to invalidate that cache line in the cache of other CPU cores, so that the cache line must be fetched from memory again.

Look at the program in `program.c` and try to understand what it is doing. The program does some computations using data in the arrays A and B, and stores results in C. The dimensions of the arrays are called N and M and are set by command-line arguments. The total amount of work to perform scales as $N * M$. Since the results are written to the C array, and N determines the size of C, it is for small values of N that we may expect to see the "*false sharing*" problem.

Build the code (using make) and run it for some different choices of N and M, for example the following cases (keeping $N * M$ fixed):
Case A: N=80000 M=80
Case B: N=800 M=8000
Case C: N=8 M=800000

For each of those cases, do test runs with 1 thread and 4 threads, and check the speedup. Do you get approximately the same speedup for all the three cases A, B and C? If not, can you understand why you get different speedups?

Use the `time` command and note both the real time and user time.

Note that the way this code is written, small values of N have the effect that different threads change values at nearby memory locations. Do you see signs of the "false sharing" problem for the N=8 case?

It is interesting to look at how the "user" time behaves for the different cases. Normally, when parallelization works well and the different threads are not disturbing each other in any way, the user time is expected to be approximately the same regardless of the number of threads. However, if the threads are disturbing each other, that can give a larger user time for performing the same amount of work.

Note that the "false sharing" issue is not specific to OpenMP — the same thing will happen if the threads were created using e.g. Pthreads also.

Task 8

The code for this task is in the `Task-8` directory.

This task is about using the "`#pragma omp task`" directive.

One case when the "`#pragma omp task`" directive can be useful is when you have some fairly complicated procedure (not just a simple for loop) in your program where at some points some heavy computational work needs to be done. Then you can have one thread that does the complicated procedure, and each time that thread finds that a piece of heavy computational work, it creates a "task" for that. The other threads can then help out by executing the tasks that were generated by the first thread (and the first thread itself can also execute tasks when it is done with the task creation part of the code).

Look at the `task_example_code.c` code and try to understand what it is doing. The part we are interested in parallelizing is the part in `main()` where the main loop is. To parallelize the work using tasks is in this case fairly easy, you just need to add the following three directives:

```
#pragma omp parallel
#pragma omp single
#pragma omp task
```

in appropriate places, and perhaps additional `{` and `}` brackets to clarify which parts of the code are meant for each directive.

Use `#pragma omp parallel` for the whole part of the code you want to parallelize. Inside the parallel region, you can use `#pragma omp single` to specify that you only want one of the threads to actually go through the loop generating tasks. Finally, to specify the actual tasks, use the `#pragma omp task` directive. Does the code run faster, use e.g. N=500?

(*Note, in this example we could have used the for-directive as an alternative solution but we want to keep the example simple to demonstrate the task-directive.*)

Another way to use the task directive is that any thread can create a new task, e.g., inside a task you can create a new task. Now, implement the merge-sort algorithm using the `task`-directive instead of nested parallelism with threads as in lab 10, task 9. You can use a separate task for each half of the list and sort recursively creating new tasks until a certain level of recursion or size of sublist. Before you merge the two sublists you also need to do a `taskwait` to be sure that both tasks have finished. Compare the two approaches (from lab 10 and lab 11), which one is more efficient and why? The original serial code can be found in lab 6, task 1 directory.