Uppsala University
Information Technology
Jarmo Rantakokko

Parallel Computing
OpenMP Lab
2013-02-22

OpenMP programming

1 Compile and run OpenMP programs

Study the `Makefile` to see how to compile OpenMP programs.

- Compile the program helloworld, `make helloworld`. Run the program by typing the program name `./helloworld`. Study the program source file and compare with the output. Re-run the program on more than one thread by changing the environment variable `OMP_NUM_THREADS`. Set the number of threads to 4 by setting `export OMP_NUM_THREADS=4` and re-run the program.

- Modify the program so that each thread prints its number (thread id) in addition to the greetings. Also make the master thread to print the total number of threads that take part in the current execution of the program. For this task you will need to call the OpenMP functions `omp_get_thread_num()` and `omp_get_max_threads()`.

2 Data sharing

By default all variables in an OpenMP program are shared, i.e., globally 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.

- The program `datasharing` has both shared and private variables. Predict its output. Compile (`make datasharing`) and run the program. Is the output what you predicted?

- All private variables are uninitialized when entering the parallel region. However, in OpenMP these can be initialized from the original variable by using `firstprivate`. Modify the program `datasharing` to make the variable `a` firstprivate. Compile and run, is the output what you expect? In OpenMP we also have a clause `lastprivate`, what is the purpose of this clause and when is it useful?

3 Work sharing

In OpenMP we can divide the work with the directive `for` in loops (and `sections` for large independent tasks). Without any sub-clauses the iterations are divided statically in `n/nthr` iterations per thread. But, we can also use different scheduling directives to divided the iterations differently.

- In program `loop` we have a simple loop which is parallelized with the `for` directive. Compile and run the program with different numbers of threads. Note the timings for each run. How does the program perform when you increase the number of threads?

- A problem with the parallelization above is that the work is not constant per iteration resulting in a load imbalance. Try different scheduling of the loop using a sub-clause `schedule(static, chunk)`, `schedule(dynamic, chunk)`, `schedule(guided, chunk)` and choosing different values on `chunk`. Which scheduling directive gives the best performance?

4 Global operations

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 `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 must use the `critical` directive (or `atomic`) which allows only one thread at a time to update the shared variable.

- The program `reduce` implements a parallelization of a norm computation. Compile and run the program. This program has two purposes, to demonstrate the use of critical and to show how to parallelize a globally data dependent application.

- The computation of a global sum can however be done simpler, OpenMP provides support for global reduction operations with the clause `reduction` on the `for` directive. Modify the program `reduce` to use the `reduction` clause. Compile, run and verify the correctness of your program.

5 Examples

Below are two serial applications, a numerical integration algorithm and a sorting algorithm. Your task is to parallelize these with appropriate OpenMP directives. Analyze the data and be sure to use data sharing directives where needed.

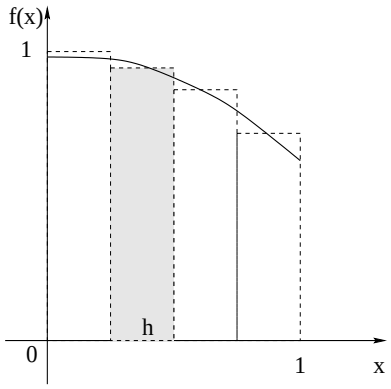
- The program `pi` computes π in parallel using numerical integration. A numerical way to compute π is the following:

$$\int_0^1 \frac{4}{1+x^2} dx = [4\, arctan(x)]_0^1 = \pi$$

If we use the midpoint rule, we can compute the above integral as follows (see Figure 1):

$$\int_0^1 \frac{4}{1+x^2} dx = h \sum_{i=1}^{n} \frac{4}{1+x_i^2} = h \sum_{i=1}^{n} \frac{4}{1+((i-1/2)h)^2}.$$

The trivial parallel implementation of the latter formula is that if we have p threads available, we slice the sum into p pieces, attach one interval to each of them to compute a partial sum, and then



Figur 1: Numerical integration with the midpoint rule.

collect the local sums into one processor, which will know the answer. Parallelize the computations with appropriate OpenMP directives.

- The Enumeration-sort algorithm

  For each element (j) count the number of elements (i) that are smaller ($a(i) < a(j)$). This gives the rank of a(j), i.e. anew(rank(j))=a(j).

  In this formulation equal elements get the same rank. Study the code in `enumsort` and parallelize the algorithm with OpenMP. Sort up to 100000 elements.

  Note: it is possible to parallelize the algorithm in several ways. You can either parallelize the j-loop, the i-loop, or both loops using nested parallelization. Try the different parallelization approaches, measure and compare their results. Explain what the parallel overheads are in each of the approaches. Does this explain your experimental results?

6 LU-factorization, an advanced example

Do a straightforward parallelization of the LU-factorization algorithm (Figure 2) using `parallel for` directives in the program `lu.c`.

In most cases the compiler directives are enough to parallelize an application. However, in some cases more flexibility is needed. OpenMP provides a set of *lock* routines that can be used to write your own synchronization operations (e.g. for synchronizing a *subset* of the threads) and to control the parallelization more in detail. The program `lulock.f90` parallelizes the LU-factorization algorithm without global synchronization in the iterations. Compile, run and study the program. Explain the program flow, i.e., the order of the computations for the different threads. Compare the performance with the straightforward parallelization only using compiler directives.

LU-factorization algorithm

```
for k=1 to n
   for i=k+1 to n
      A(i,k)=A(i,k)/A(k,k)
   end for
   for i=k+1 to n
      for j=k+1 to n
         A(i,j)=A(i,j)-A(i,k)*A(k,j)
      end for
   end for
end for
```

Figur 2: LU-factorization in situ without pivoting. L is stored below the diagonal and U in the upper part of A.

7 Report

The laboration is a part of the examination. To pass you need to either attend the lab and work actively on the tasks or write a short informal report summarizing your results and answers together with your source code for the different tasks.