Assignment 3

Bjørn Christian Weinbach   Thameez Bodhanya

October 2, 2021

Instructions

For compiling the code that was provided in the assignment as well as the code that was edited for solving the exercises, the makefile provided was used. In some cases, the programs would not compile with -std=c++11 flag. This was mentioned in the lectures. If that is the case. Replace the flag with -std=c++0x flag.

We have used *siegnbahn.it.uu.se* for checking that the code runs. The code is structured in the following way:

- Q2
  - Game_Of_Life.c
  - Makefile
  - results.text
  - test.sh

There is a makefile for every question and subquestion. To check a question just navigate to the corresponding code and run:

*Make PROG=**filename***

Where ***filename*** is the name of the C/C++ file to compile.

Question 1: Sieve of Eratosthenes - OpenMP

In this task we were asked to rewrite our implementation of the *Sieve of Eratosthenes* using OpenMP instead of Posix Threads.

- Step one of this task was to remove any references to posix threads (this included thread creation, locks and joins).
- Step two was then to implement OpenMP by using specific OpenMP constructs such as:

  - #pragma omp parallel num_threads(numThreads) - with the number of requested threads as a command line argument
  - #pragma omp single - to run the serial initiation of the seeds
  - #pragma omp for - to split up the workload between the threads in an as even fashion as possible

- Step three was to compile and test the code, and finally compare results

Note that one could have set the number of requested threads using the environment variable *OMP_NUM_THREADS*, however to stay true to the implementation of the original code it was decided to set it explicitly using a command-line argument.

For the complete implementation please see the attached code file *sieve.cpp*

The OpenMP implementation was far easier to write when compared to the Posix version. Many functions were able to be dropped and the code became far more readable and concise. The code is also far safer as it is easier to understand. And the written implementation allows it to be run on a single-threaded system without any code changes.

Results

All results are from the *vitsippa.it.uu.se* system.

As per previous tests, we can see that for small max numbers (10 or 100), the overhead of threads actually increases the amount of time taken to compute the results INSERT GRAPH HERE However as we hit medium max numbers (1000000 or 10000000), we already see an approx 50% reduction of time as we increase number of threads INSERT GRAPH HERE And as we hit large max numbers we see that we have hit 50% reduction per addition of threads INSERT GRAPH HERE This time decrease hits substantially as we break into larger numbers as we overcome the overhead that parallelizing the application adds (and thus can take advantage of the threads)

note: the difference between 8 and 16 threads are nominal, this indicates that the system was unable to give us a full 16 threads at testing time, however should we have gotten them, we would expect to see a full 50% reduction in time

Interestingly if we compare the runs of the Posix version to the OpenMP, we see a significant reduction in times (almost 75%). This is likely due to the less overhead OpenMP adds to parallelize the application vs our implementation Posix threads (with the additional functions and jumps). INSERT GRAPH HERE

Question 2: Conway's Game of Life

Question 4: Gaussian Elimination in OpenMP

In this task we were required to parallelize the back-substitution portion of a Gaussian Elimination algorithm. Two versions of the algorithm were presented, each containing two for-loops. Additionally once parallelized, we were asked to then gauge performance of our solutions with varying scheduling types.

Row-orientated algorithm

The algorithm to parallelize can be seen below:

```
    for (row = n-1; row >= 0; row--)
    {
      x[row] = b[row];
      for (col = row+1; col < n; col++)
        x[row] -= A[row][col] * x[col];
      x[row] /= A[row][row];
    }

```

If we look at the algorithm, we see that it is not possible to parallelize the outer-loop as the inner loop relies on values generated in previous iterations of the outer-loop. Namely in line _x[row] -= A[row][col] * x[col];_, we see *x[row]* is being updated to a value which contains *x[col]*. Should the outer-loop be parallelized, the value of *x[col]* would depend on the order at which the threads execute, thus leading to a race condition. In addition, at the start of each thread, the value of *x[row]* is being set to *x[row] = b[row]* which would again influence the value of *x[col]*.

We can however parallelize the inner-loop safely, as we know that the value of *col* is set to *row+1*. And since the outer-loop is being run serially counting down from *numThreads*, the value of *x[col]* is constant. In addition, the operation being run in the inner-loop on the shared variable x is only subtraction, and this does not rely on an order-of-operation (e.g $5-10-2$ is equal to $-2-10--5$) and therefore is thread-safe.

An implementation of this can be seen below:

```
    for (int row = numUnknowns - 1; row >= 0; row--)
    {
      x[row] = b[row];
      #pragma omp parallel for default(shared)
      for (int col = row + 1; col < numUnknowns; col++)
      {
        x[row] -= a[row][col] * x[col];
      }
      x[row] /= a[row][row];
    }

```

Column-orientated algorithm

The algorithm to parallelize can be seen below:

```
    for (int row = 0; row < numUnknowns; row++)
      x[row] = b[row];
    for (int col = numUnknowns - 1; col >= 0; col--)
    {
      x[col] /= a[col][col];
```

```
      for (int row = 0; row < col; row++)
        x[row] -= a[row][col] * x[col];
    }

```

If we look at the algorithm, we see that it is not possible to parallelize the outer-loop, as both the inner-loop and outer-loop update the same variable. Namely the first statement of the outer-loop *x[col] /= a[col][col]* changes the value of *x[col]*, the current value of *x[col]* is changed during the preceding runs of the inner-loop. This can be seen as the inner-loop counts up from 0 towards the value of *col*. Thus the value of *x[col]* changes depending on the order of thread execution, since *x[col]* is updated via division (where the order of operation does matter), this is not thread-safe and thus a data-race.

We can however parallelize the inner-loop safely, as we know that the value of *col* is constant as *row* is always less than *col*, thus *x[col]* is never changed. In addition, the operation being run in the inner-loop on the shared variable x is only subtraction, and this does not rely on an order-of-operation (e.g $5 - 10 - 2$ is equal to $-2 - 10 - -5$) and therefore is thread-safe.

An implementation of this can be seen below:

```
    for (int row = 0; row < numUnknowns; row++)
    x[row] = b[row];
    for (int col = numUnknowns - 1; col >= 0; col--)
    {
      x[col] /= a[col][col];
      #pragma omp parallel for default(shared)
      for (int row = 0; row < col; row++)
          x[row] -= a[row][col] * x[col];
    }

```

Results

When compared to the serial implementation, we have vetted that the algorithm performs as expect and produces the expected results (setting u to *3* runs the application in test mode). At lower number of unknowns (3, 100 and 1000), we see an expected increase in time taken to run the program. This is true for both the row and column based approaches, see figure 1.

However, when we get to 10000 number of variables, we see that the row-based solution still increasing in time as the number of threads are added. This is in contrast to the column-based solution which practically halves in time as the number of threads double. A possible reason for this is that even though the inner-loop of the row-based solution has been parallelized, there exists a substantial calculation (namely *x[row] /= a[row][row];*) is still being done serially. See figure 2.

OpenMP Schedulers

This section documents our experimentation when running both the column-based solutions for 16 threads, with 42000 variables against the different OpenMP

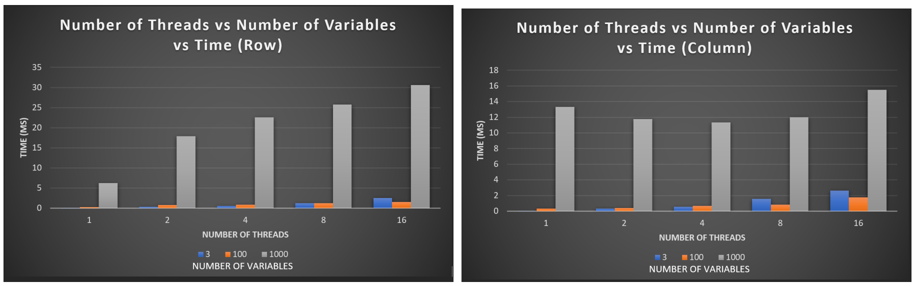

Figure 1: Column- and Row oriented performance vs no of threads.

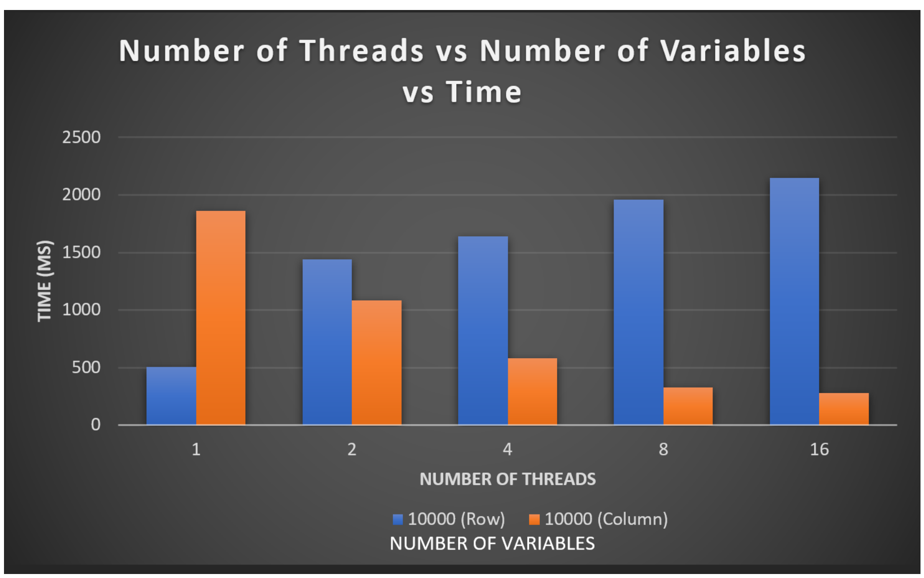

Figure 2: Column- and Row oriented comparison.

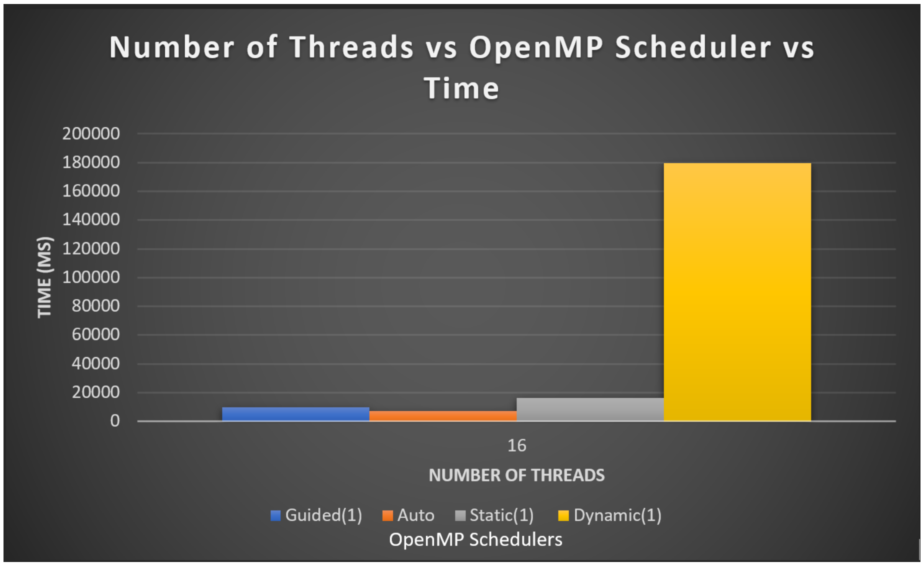

Figure 3: Column- and Row oriented performance vs no of threads.

Schedulers

```
#!/bin/bash
set -eo pipefail

for s in {static,dynamic,guided,auto}; do
    export OMP_SCHEDULE=$s
    filename="results_col_$OMP_SCHEDULE.txt"
    set +e
    rm $filename
    set -e
    OMP_NUM_THREADS=16 ./$PROG -u 42000 >> $filename
done

```

As we can see schedule type **auto** performed the best for this particular application resulting in an average increase of *XXX* per thread. With schedule type **dynamic** performing the worst. This would track as the amount of work per iteration is vastly variable, and therefore a more generic slicing up of iteration per thread would on average perform the best. See figure 3.