Lab 3
OpenMP

Martin Söderén
marso329, 9009291098

12 maj 2016

1 Introduction

The problem consists of solving a stationary heat conduction problem using OpenMP on a shared memory computer. The problem consists of a grid of points in a 2D configuration. All points have a initial value and for each time step each points temperature changes depending on the average temperature of the surrounding points. This keeps on going until the solution converges.

2 Method

The grid is implemented as a array of arrays of doubles. The grid is split up row wise into chunks. The number of chunks is equal to the number of points in each column divided by the number of threads. So the number of rows in a chunk is equal to the number of threads. In each iteration the chunks are calculated one after another from top to bottom in the grid. Each time a chunk is calculated one thread takes one row each, synchronize and then writes back the data to the grid. But before the write back the last row is stored so that it can be used in the next chunk since that depends on data in the last row of that chunk.

Algorithm 1 Stationary heat conduction

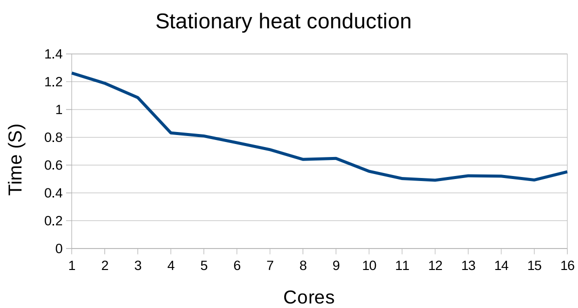
```
procedure HEAT
    if Master then
        Init grid
    end if
    for maxiter do
        Error=0 (shared)
        copy first row into copy_before
        Calculate numberOfChunks
        for numberOfChunks do
            calculate the start row for this chunk
            if Master then
                copy_after=last row of chunk
            end if
            Synchronize using omp barrier
            calculate which row to work on
            temp_error=0.0
            for each element in row do
                calculate new temp
                calculate temp_error
                if local_error> temp_error then
                    temp_error=error
                end if
            end for
            Synchronize using omp barrier
            access critical section using omp_set_lock
            if if temp_error>error then
                error=temp_error
            end if
            Write back data to grid
            leave critical section using omp_unset_lock
            if Master then
                Switch copy_before and copy_after
            end if
        end for
        if error<tolerance then
            exit
        end if
    end for
end procedure
```

2.1 Design

The whole program runs in parallel because at first the parallel part was started in the iteration loop and that meant that in each iteration new threads had to be created which took a lot of time and had a huge impact on performance so instead the program runs in parallel from top to bottom. More or less all threads runs the same loop that is synchronized before the calculation and before the write back.

3 Result

The stationary heat conduction calculation scales fairly well up to 11 cores. With 11 cores there is reduction of calculation time 2.5 lower than the time with a single core. With more cores there is hardly any difference in speed. The times were measured with a grid of 1000x1000 points. With a larger grid there might be a more significant change since each core can do more job in each chunk. The reason for the poor increase in performance can be because of the critical section which all threads needs to access to write back the error of chunk.



Figur 1: Times heat conduction calculation with a 1000x1000 grid

A lapsolvomp.c

```
#include "omp.h"
#include "stdlib.h"
#include "stdio.h"
#include "string.h"
#include "math.h"
#include "semaphore.h"

#define CEILING_POS(X) ((X-(int)(X)) > 0 ? (int)(X+1) : (int)(X))
#define CEILING_NEG(X) ((X-(int)(X)) < 0 ? (int)(X-1) : (int)(X))
#define CEILING(X) ( ((X) > 0) ? CEILING_POS(X) : CEILING_NEG(X) )

int n = 1000;
int maxiter = 1000;
double error = 0.0;
double tol = 0.001;

double** init_t() {
        double** T = (double**) malloc(sizeof(double*) * (n + 2));
        for (int i = 0; i <= n + 1; i++) {
                T[i] = (double*) calloc(n + 2, sizeof(double));
        }
        for (int i = 0; i <= n + 1; i++) {
                if (i != n + 1) {
                        T[i][0] = 1.0;
                        T[i][n + 1] = 1.0;
                }
                T[n + 1][i] = 2.0;
        }

        return T;
}

void free_t(double** T) {
        for (int i = 0; i <= n + 1; i++) {
                free(T[i]);
        }
        free(T);
}

void print_t(double** T) {
        for (int i = 0; i <= n + 1; i++) {
                printf("\n");
                for (int j = 0; j <= n + 1; j++) {
                        printf(" %lf ", T[i][j]);
                }
        }
        printf("\n");

}

int main(int argc, char ** argv) {
        omp_lock_t writelock;
```

```
        omp_init_lock(&writelock);
        double** T = init_t();

        double* temp;
        double* copy_before = (double*) malloc(sizeof(double) * (n + 2));
        double* copy_after = (double*) malloc(sizeof(double) * (n + 2));
        //for each computing part(couple of rows)
        int start = 0;
        int end = 0;
        //omp_set_num_threads(1);
        int chunks;
#pragma omp parallel shared(T,start,end,n,copy_before,copy_after,writelock,error,maxiter,to
        {
                for (int iteration = 1; iteration < maxiter; iteration++) {
#pragma omp barrier
                        memcpy(copy_before, T[0], sizeof(double) * (n + 2));
                        error = 0.0;
                        chunks = (int) CEILING(
                                        (double ) n / (double ) omp_get_num_threads());
                        if (chunks <= 0) {
                                chunks = 1;
                        }
                        int element_per_chunk = n / chunks;
                        for (int i = 0; i < chunks; i++) {
                                start = element_per_chunk * i + 1;
                                end = element_per_chunk * (i + 1);
                                if (i == chunks - 1) {
                                        end = n;
                                }
                                if (omp_get_thread_num() == 0) {
                                        memcpy(copy_after, T[element_per_chunk * (i + 1)],
                                                        sizeof(double) * (n + 2));
                                }
#pragma omp barrier
                                int j = start + omp_get_thread_num();
                                double temp_error = 0.0;
                                double copy_during[n + 2];
                                if (j <= end) {
                                        memcpy(copy_during, T[j], sizeof(double) * (n + 2))
                                        // for each column
                                        for (int k = 1; k <= n; k++) {
                                                if (j == start) {
                                                        copy_during[k] = (T[j][k + 1] + T[j
                                                                        + T[j + 1][k] + cop
                                                } else {
                                                        copy_during[k] = (T[j][k + 1] + T[j
                                                                        + T[j + 1][k] + T[j
                                                }
                                                if (fabs(copy_during[k] - T[j][k]) > temp_e
                                                        temp_error = fabs(copy_during[k] -
                                                }
                                        }
                                }
```

```
#pragma omp barrier
                    if (j <= end) {
                        omp_set_lock(&writelock);
                        if (temp_error > error) {
                            error = temp_error;
                        }
                        memcpy(T[j], copy_during, sizeof(double) * (n + 2))
                        omp_unset_lock(&writelock);
                        //switch places on temp storage
                    }
                    if (omp_get_thread_num() == 0) {
                        temp = copy_after;
                        copy_after = copy_before;
                        copy_before = temp;
                    }
                }
#pragma omp barrier
                if (error < tol) {
                    if (omp_get_thread_num() == 0) {
                        printf("iterations: %i\n", iteration);
                    }
                    break;
                }

            }
        }
        free(copy_before);
        free(copy_after);
        free_t(T);
        omp_destroy_lock(&writelock);

}
```