Hasso Plattner Institut

IT Systems Engineering | Universität Potsdam

Assignment Feedback #2

Parallel Programming Concepts
Winter Term 2013 / 2014

Dr. Peter Tröger, M.Sc. Frank Feinbube

Assignment 2: Problems & Solutions

- Learning Goals:
  - Foster / Optimization on Heatmap @Threads
  - Monitor Concept
- Heat Map
  - Parallel Simulation, Ghost Cells
  - Guessing Thread Counts right
- Parallel Grep wth Java Monitors
  - Think critical section.. Inverted!

2.1 Heat Map with Threads

```
./heatmap 231 257 123 task2.1_hotspots_medium.csv
task2.1_coords_medium.csv
```

```
output.txt
11112221111111111100
11123432111111111110
11124X42211111111111
11124442111111222111
11122222111112222211
11111211111112232211
01111111111111222111
```

```
output.txt
1.0
0.030563410733335933
```

Good or bad?

```
void simulateRound() {
        [...]
        memcpy(old_heat_map, new_heat_map, heat_map_size * sizeof(double));

        // calculate heat
        [...]
        // synchronize
        [...]
}



for(int r= 0; r<n_rounds; r++)
        simulateRound();
```

Good or bad?

```
for(int r= 0; r<n_rounds; r++){
    pthread_barrier_wait(&barr1);                    // stop for worker threads
    map_val_temp= map_val_old;
    map_val_old= map_val_new;
    map_val_new = map_val_temp;
    for(x=0;x<num_cols; x++){
        for(y=0; y<num_rows;y++){
            map_val_new[x][y] = 0;
        }
    }
    pthread_barrier_wait(&barr2);                    // start for worker threads
}
```

Good or bad?

```
for(j=0;j<number_of_rounds;j++) {
    //set heatsources to 1
    set_heat_sources();

    for(i=0;i<thread_num;i++){
        // set thread arguments
        […]
        pthread_create( &thread[i], NULL, calculate_round, (void*) &data[i]);
    }
    for(i=0;i<thread_num;i++){
        pthread_join( thread[i], NULL );
    }

    //swap old and new values
    swap_fields()
}
```

Good or bad?

```
int threadCount = getNumberOfCores();        // set thread count to number of cores
// Prepare thread arguments
heatmapThreadArg* threadArgs = new heatmapThreadArg[threadCount];
for(int i=0; i<threadCount; i++){
        threadArgs[i].heatmapDesc = heatmapDesc;
        threadArgs[i].xStart = i*ceil((double)heatmapDesc->width/(double)threadCount);
        int possibleWidth = (i+1)*ceil((double)heatmapDesc->width/(double)threadCount);
        threadArgs[i].xEnd = (possibleWidth > heatmapDesc->width) ?
                                    heatmapDesc->width : possibleWidth;
        threadArgs[i].yStart = 0;
        threadArgs[i].yEnd = heatmapDesc->height;
    }
[…]
for(int i=0; i<threadCount; i++){
            pthread_create(&threads[i], &attr, thread_heatmap_computation,
                        (void *)&threadArgs[i]);
}
```

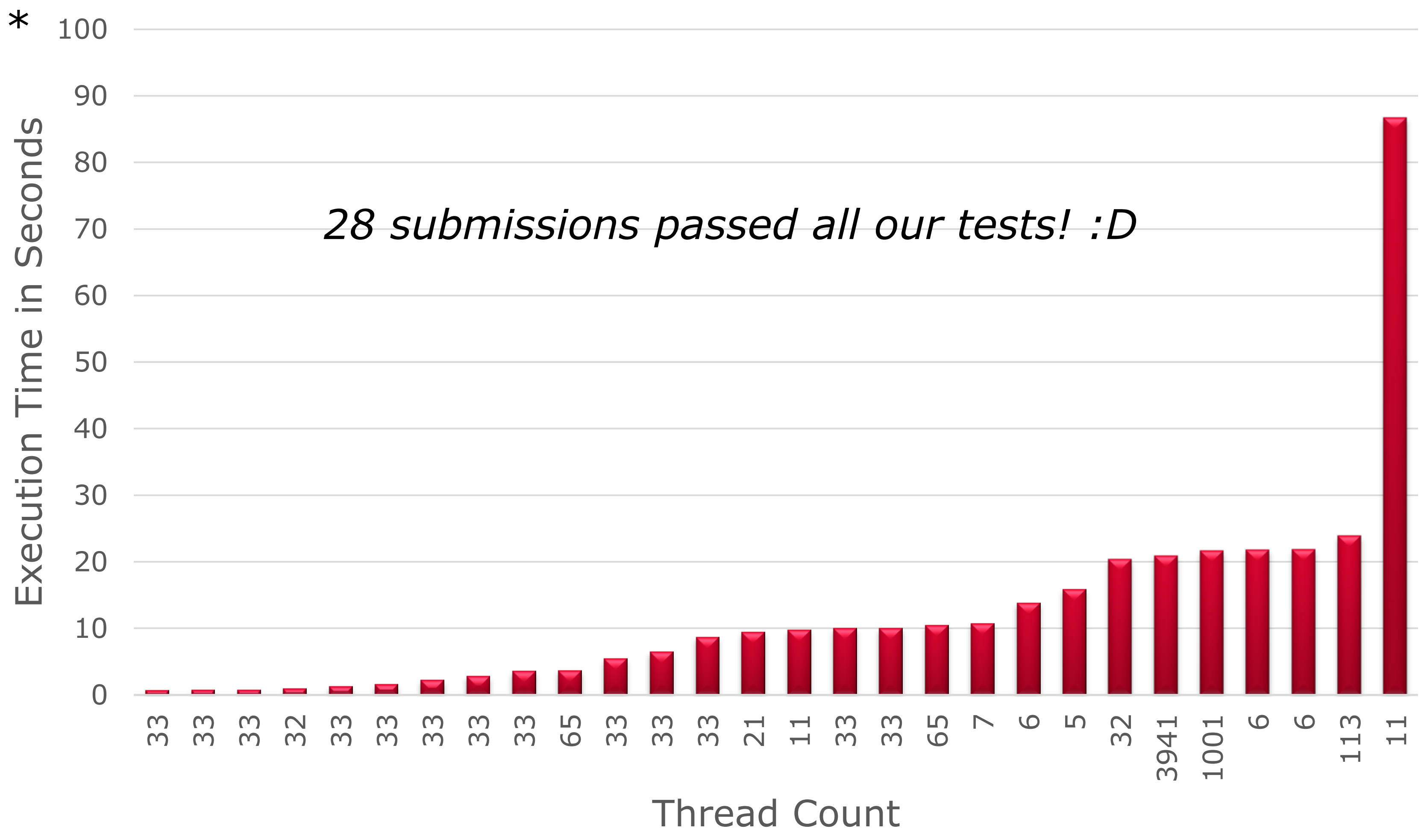
Good or bad?

```
// worker thread code
while (num_rounds--) {
    while (!data->running) usleep(10);          // wait for main thread
    calculate_own_area_of_heat_map();
    data->running = false;                      // sets itself to sleep
    }
}

// main thread code
for (round = 0; round < roundc; ++round) {
    heatmap.set_hotspots();
    heatmap.swap();
    for (const auto& threadData : data)
        threadData->running = true;             // allows workers to calculate
    while (!allDone(data)) usleep(10);
}
```

Good or bad?

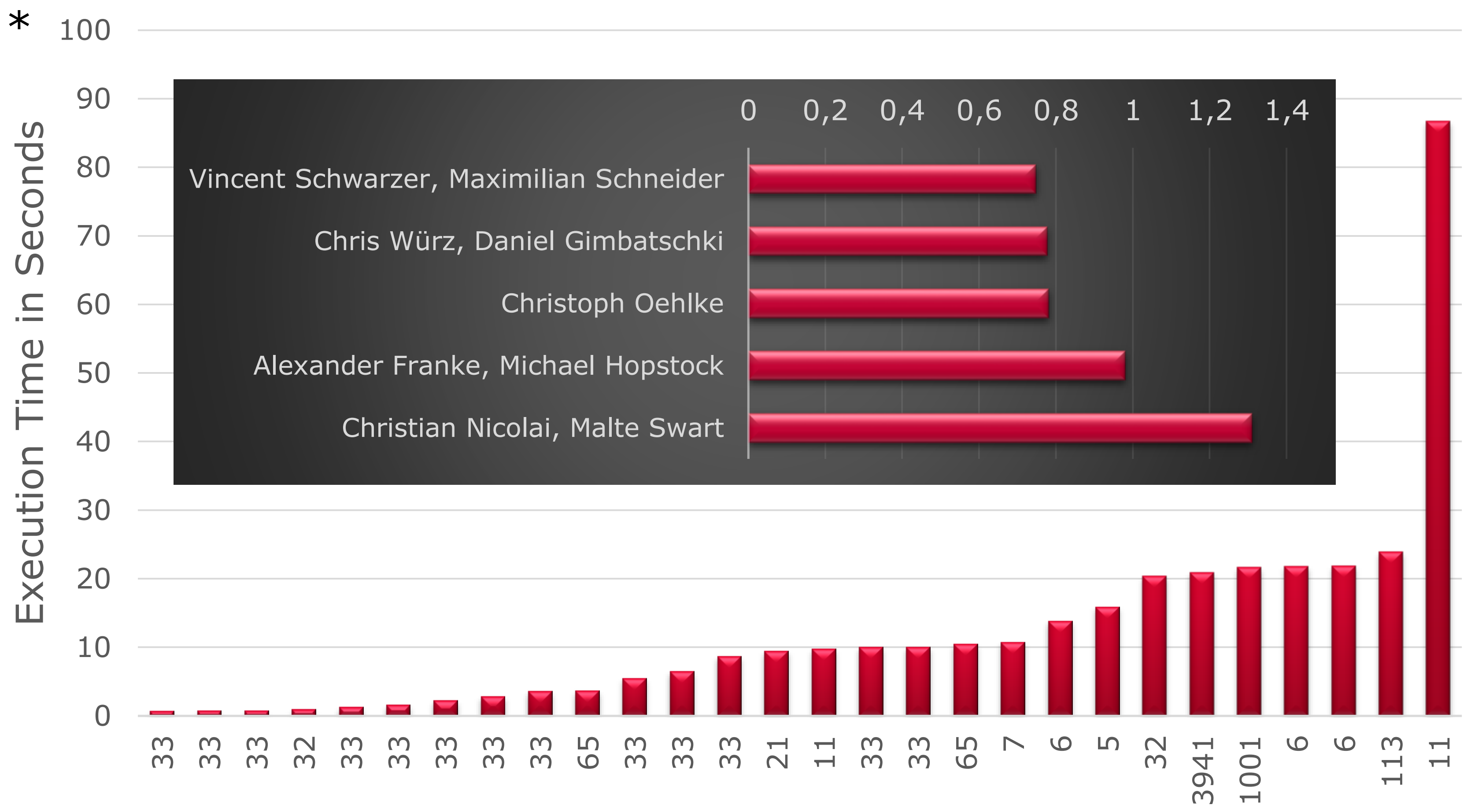
```
for (unsigned int round = 0; round < roundCount; round++) {
    for (unsigned int x = rect->xMin; x <= rect->xMax; x++) {
        for (unsigned int y = rect->yMin; y <= rect->yMax; y++) {
            destinationHeatMap[x][y] = (   sourceHeatMap[x-1][y-1] +
                                                [...]
                                            sourceHeatMap[x+1][y+1]
                                          ) / 9.0;
        }
    }
    // activate hotspots:
    set_hotspots_for_own_rect(rect);
    pthread_barrier_wait(barrier);

    if (rect->xMin == 1 && rect->yMin == 1)         // designated thread switches buffers
        switch_buffers(sourceHeatMap, destinationHeatMap);
    pthread_barrier_wait(barrier);
}
```

And the WINNER is…



* execution was aborted after 350 seconds runtime

And the WINNER is...



* execution was aborted after 350 seconds runtime

2.2 Parallel Grep with Java Monitors

```
java -jar pargrepmon.jar /tmp/strings.txt /tmp/input.txt


// program structure
void lookforit() {
 // get string to search for
 // look for the string in buffer
 // write string to result list
}


// output.txt
abc;3
def;10
```

The ababa-Problem

```
Test description: ababa
Running:
java -jar pargrepmon.jar task2.2_ababa_strings.txt
task2.2_ababa.txt
Max thread count: 28 [Estimated!]
Runtime:            0.08
[ERROR]-----------------------------
Result file does not match regular expression.

output.txt
ab;2
aba;1
roses;0
```

Good or bad?

```
synchronized public boolean lookForIt() {
    // ... Read next search string using Monitor pattern

    int i = 0;
    int j;
    int occurences = 0;

    while (i < this.data.length) {
        j = 0;
        // ... look for search string at position j
        occurences++;
    }

    // ... Write results using Monitor pattern

    return true;
}
```

This is not using the Montior pattern

```
private String getNextString()
{
     synchronized (Shared.strings)
     {
          return Shared.strings.poll();
     }
}
```

… But this is

```
private String getNextString()
{
    synchronized (Shared.strings)
    {
        while (Shared.gettingNextString) {
            try {
                Shared.strings.wait();
            } catch (...) { ... }
        }
        Shared.gettingNextString = true;
        String result = Shared.strings.poll();
        Shared.gettingNextString = false;
        Shared.strings.notify();
        return result;
    }
}
```

Non-synchronized File access

```
private void lookForIt() {
    // ... get search string using Monitor pattern

    this.waiting = false;
    String line = this.searchStrings[index];

    /** look for the string **/
    int lastIndex = 0;
    int count = 0;

    while (lastIndex != -1) {
        lastIndex = dataToAnalyze.indexOf(line, lastIndex);
        if( lastIndex != -1) {
            count ++;
            lastIndex += line.length();
        }
    }

    /** write string to result list **/
    this.writer.println(line + ";" + count);
}
```

Choosing number of threads

Bad: constant number

- does not scale on different machines
- does not scale for different problem size

Simple: Based on problem size

+ Easy to program
- Potentially big overhead when creating too many threads
- Might leave CPUs without work

Good: Based on number of CPUs

+ Uses resources that are actually available
- Requires sophisticated distribution of workload onto threads
- IO work in threads might leave CPU unoccupied

  => use 2*#CPUs, depends on problem

Assignments to come...

Shared Memory Parallelism

- ✓ Decrypt with OpenMP (25.11. - 08.12.)
- ✓ HeatMap with OpenMP
- ✓ Noise with OpenMP

Assignment 3: Questions?

Accelerators

- ➢ HeatMap with OpenCL/CUDA (09.12. - 05.01.)
- ➢ Game of Life? Gauss Filter? Noise? Fractals?
- ➢ Crypt? Sorting? String-Search?
- ➢ Nqueens?

Shared Nothing Parallelism