Friedrich-Alexander-Universität
Technische Fakultät

Computer Architecture Practical Exercise

7 Parallel Jacobi

Kenan Gündogan[1] **Philipp Gündisch**[1]

[1]Friedrich-Alexander Universität Erlangen-Nürnberg, Chair of Computer Science 3 (Computer Architecture)

December 19, 2023

Parallel Jacobi

Motivation

How can we improve performance even further?

- Finished single core optimization with the last exercise
- Cluster node consists of many more unused cores
- Can the Jacobi algorithm be executed in parallel?

Parallel Jacobi

Motivation

To analyze how the Jacobi algorithm can be run in parallel we need to analyze its data dependencies.

- The calculation of one pixel depends on the pixels to the north, south west and east
- The source grid is accessed **read-only**
- The target grid is accessed **write-only**
- The source and target grids are **swapped**

Yes, the Jacobi algorithm can be parallelized but we need a synchronization point when the grids are swapped!

pthreads

main() function

```c
#include <pthread.h>

#ifndef THREADS
#define THREADS (1)
#endif

int main(int argc, char **argv) {
    ...

    // Create thread arguments
    struct work_package_s pkgs[THREADS];
    // for loop to initialize pkgs

    for(runs = 1u; actual_runtime_us < min_runtime_us; runs=runs<<1u) {
        start = get_time_us();
        // for loop with pthread_create
        // for loop with pthread_join
        stop = get_time_us();
        actual_runtime_us = stop - start;
    }
    ...
}
```

pthreads

Function Parameters

```c
#include "barrier.h"

// Struct containing all parameters for a thread
struct work_package_s {
        double * grid1;
        double * grid2;
        uint32_t size_x;
        uint32_t size_y;
        uint32_t runs;
        ...
};

// Function to be executed by each thread
void * worker_thread(void *void_args) {
        struct work_package_s *args = (struct work_package_s*) void_args;
        for (uint32_t i = 0 ; i < args->runs ; i++) {
                jacobi_subgrid(...)
                sync_barrier(THREADS);
                // TODO swap
        }
        pthread_exit(NULL);
}
```

likwid-pin

Thread Pinning

If we run our program with more than one thread on any cluster node, threads might get rescheduled to a different core during execution. On the new core there is no cached data resulting in a loss of performance. Thus, we need to pin the threads to logical cores with the `likwid-pin` command.

```
likwid-pin -c E:<domain>:<cores>:<batch>:<stride>

<domain> : S0 or S1 (CPU socket)
<cores>  : 1-10
<batch>  : 1
<stride> : 1
<@>      : chain operator

Example with 12 threads (one per core):
likwid-pin -c E:S0:10:1:1@S1:2:1:1  ./jacobi ...
```

Task 7.1: Parallel Jacobi

`jacobi_subgrid()`

- Update the `makefile` to compile with `-std=c11` (required for `atomics`)
- Take the jacobi implementation of the last exercise to create a new function `jacobi_subgrid()`
- Extend the signature of `jacobi_subgrid()` to support calculating an arbitrary large subgrid
- Update `jacobi.h`
- Test `jacobi_subgrid()` with a single thread by splitting the grid into multiple subgrids and check the result ppm

Task 7.2: Parallel Jacobi

`main()`

- Update the `makefile` to compile with `-lpthread`
- Update your `main.c` to support parallel execution of `jacobi_subgrid()`
- Use `likwid-pin` to assign threads to cores
- Benchmark with 4 GiB of RAM
- Choose $b_x = 768$ and $b_y = 50$
- Update the `sbatch` script to allocate 20 cores with `#SBATCH --cpus-per-task=20`
- Benchmark from 1 to 20 threads (step size: 1)
- Share the work evenly throughout the threads by dividing the grid in (almost) equally large grid sizes
- Create a plot with MUp/s on the y-axis and the number of threads on the x-axis
- The plot should compare the performance with and without thread pinning
- **NOTE: The main thread should also perform some work!**
- Do not kill the main thread with `pthread_exit()`

Task Overview

- E 7.1: `jacobi_subgrid()`
  - Start with the 2d blocked jacobi implementation
  - Implement `jacobi_subgrid()`
  - Test the implementation
- E 7.2: `main()`
  - Benchmark the updated implementation with fixed 4 GiB
  - Benchmark over 1 to 20 threads
  - Create a plot to evaluate thread pinning

Appendix: `C11` Atomics

`barrier()`

- Compile with -std=c11
- All threads will wait at the barrier
- The last arriving thread will release the barrier
- After the barrier every thread can safely swap the grids
- Barrier implementation uses atomic functions *(with sequential consistency)* to be thread-safe

Appendix: Checklist

Performance Optimization (1/2)

During the timeline of this class new bullet points will be added. Recently added entries are bold.

- Compiling
  - Choice of the compiler (`icc`)
  - Compiler flag to optimize aggressively (e.g. `-O3`)
  - Compiler flag to adapt for specific hardware (e.g. `-xHost`)
- Programming Techniques (if applicable)
  - Use `#define` and `const` instead of variables
  - Data type aware programming
  - Use aligned memory (e.g. with `_mm_malloc()` or `posix_memalign()`)
  - Consecutive address iteration
  - Variable declarations outside of loops
  - Reduce function calls
  - Use intrinsics (to utilize SIMD)
  - Cache aware programming (Spatial Blocking)
  - Prefetcher aware programming (L1 Cache Blocking)

Appendix: Checklist

Performance Optimization (2/2)

During the timeline of this class new bullet points will be added. Recently added entries are bold.

- Measurement
  - Reasonable benchmark time
  - Reasonable benchmark workload
  - Reduce interference factors to a minimum
- Optimization Process
  - Check assembler code while optimizing
  - Check performance gains while optimizing
  - Use profiling tools
  - Ensure correctness of code
  - Optimize iteratively
  - **Optimize single core performance first**