Friedrich-Alexander-Universität
Technische Fakultät

Computer Architecture Practical Exercise

8 Parallel Jacobi

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

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

January 13, 2025

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 before the grids are swapped! **BUT** synchronization points like thread barriers can add a very significant run-time overhead. Very advanced and problem specific solutions are needed to implement an efficient synchronization point which works for 72 threads. Therefore, we make the use of the thread barrier an optional task. Instead we use a very large grid size to keep a single run through the grid in the order of seconds.

pthreads

main() function

```
#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

    start = get_time_micros();
    // for loop with pthread_create (THREADS-1 times)
    // main runs worker thread 0
    worker_thread(...);

    // for loop with pthread_join (THREADS-1 times)
    stop = get_time_micros();
    actual_runtime_us = stop - start;
    ...
}
```

pthreads

Function Parameters

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

// Function to be executed by each thread
void * worker_thread(void *void_args) {
        struct work_package_s *args = (struct work_package_s*) void_args;
        jacobi_subgrid(...)

        // If not main thread
        if(/* your condition */) {
                pthread_exit(NULL);
        }

        return 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 many cache misses which affect the performance. Thus, we need to assign the threads to fixed cores to prevent dynamic thread placement. This technique is called **thread pinning** and can be implemented with the `likwid-pin` command.

```
module load likwid/5.3.0

# Prnt usage information
likwid-pin -h

# Print domain information
likwid-pin -p
Domain N:    # On node level
        0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,...,71
Domain S0:   # On socket level
        0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,...,35
...

# Pinning in your sbatch file
srun likwid-pin -q -c N:0-$(($THREADS-1)) ./bin/jacobi <params> >> <file>
```

Task 8.1: 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 sub-grid
- Update `jacobi.h`
- Test `jacobi_subgrid()` with a single thread by splitting the grid into multiple sub-grids and check the result ppm

Task 8.2: Parallel Jacobi

`main()`

- Update the `makefile` to compile with `-pthread`
- Update your `main.c` to support parallel execution of `jacobi_subgrid()`
- Use `likwid-pin` to assign threads to cores
- Benchmark with a grid size of 54058x54058
- Choose $b_x = 216$ and $b_y = 25$
- Update the `sbatch` script to allocate 72 cores with `#SBATCH --cpus-per-task=72`
- Update the `sbatch` script srun cores `export SRUN_CPUS_PER_TASK=72`
- Benchmark from 1 to 72 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
- **NOTE: The main thread should also perform some work!**
- Do not kill the main thread with `pthread_exit()`

Optional Task 8.3: Thread Barrier

`main()`

- Extend 8.2 to support multiple runs
- Extend 8.2 to support synchronization between runs
- Only create `THREADS-1` many *pthreads* during the program execution (otherwise likwid-pin causes trouble)
- Create an atomic variable `running` which is set to a specific value by the main thread if the minimal run-time is exceeded
- The worker threads should check for `running` and optionally stop with `pthread_exit` or double runs and start iterating
- After one round of runs, the main thread checks the time, updates `running` if needed and only then calls `sync_barrier`
- Test the solution with a small grid and many threads and check the result ppm
- Benchmark like in 8.2 but change it to a much smaller grid size < 1000x1000

Task Overview

- E 8.1: `jacobi_subgrid()`
  - Start with the 2d blocked jacobi implementation
  - Implement `jacobi_subgrid()`
  - Test the implementation
- E 8.2: `main()`
  - Benchmark the updated implementation with fixed grid 54058x54058
  - Benchmark from 1 to 72 threads
- Optional E 8.3: `main()`
  - Benchmark the updated implementation with a very small grid size
  - Implement / Use the thread barrier
  - Benchmark from 1 to 72 threads

Appendix: `C11` Atomics

`sync_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 (`icx`)
  - 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**