Problem2

Table of Contents

Introduction

This project was written in **C++** because it's a language that I enjoy using and I am familiar with. It includes an architecture made to be extensible and easy to maintain while reusing as much code as possible which means a bit more overhead.

Folders structure

```
├── xmake.lua
├── bin -> folder created after compilation
│   ├── MatmultD -> second problem executable
│   ├── pc_dynamic -> first problem executable (dynamic)
│   ├── pc_static_block -> first problem executable (static block)
│   ├── pc_static_cyclic -> first problem executable (static cyclic)
├── xmake.lua
├── problem1 -> contains the code for the first problem
│   ├── xmake.lua
│   ├── pc_serial.java.cpp -> translation of the basic java code to cpp
│   ├── dynamic
│   │   ├── xmake.lua
│   │   ├── Main.cpp
│   │   ├── DynamicThread.cpp and DynamicThread.hpp -> Child class of Thread
```

```
│   │   ├── WorkQueue.cpp and WorkQueue.hpp
│   ├── static_block
│   │   ├── xmake.lua
│   │   ├── Main.cpp
│   │   ├── StaticBlockThread.cpp and StaticBlockThread.hpp -> Child class of Thread
│   ├── static_cyclic
│   │   ├── xmake.lua
│   │   ├── Main.cpp
│   │   ├── StaticCyclicThread.cpp and StaticCyclicThread.hpp -> Child class of Thread
├── problem2 -> contains the code for the second problem
│   ├── xmake.lua
│   ├── Main.cpp
│   ├── Matrix.cpp and Matrix.hpp
│   ├── MatrixThread.cpp and MatrixThread.hpp -> Child class of Thread
├── Shared -> contains the code shared between the problems
│   ├── Clock.cpp and Clock.hpp
│   ├── PrimeChecker.cpp and PrimeChecker.hpp
│   ├── Thread.cpp and Thread.hpp
│   ├── ThreadPool.hpp -> No cpp file because it's a template class
├── data -> folder containing the test matrices
```

Installation

Requirements

- C++ compiler (g++, clang++, msvc++)
- XMake
- Git

Installation

```
git clone git@github.com:GlassAlo/CAU_Multicore.git
cd CAU_Multicore/proj1
```

Compile

```
xmake f -m release && xmake -y
```

- **-m release** is used to compile the project in release mode, which is faster than debug mode.
- **xmake -y** is used to compile the project. The **-y** flag is used to skip the confirmation prompt.
- **xmake** will create a **bin** folder with the executables inside.

Usage

```
./bin/MatmultD <number of threads> < <path to matrix file>
```

- Takes one argument: the number of threads and reads the matrix from a file.

Test

🖥️ Hardware Specifications

- **OS**: Garuda Linux Broadwing x86_64
- **Kernel**: 6.13.8-zen1-1-zen
- **CPU**: AMD Ryzen 9 5900HS with Radeon Graphics (16) @ 4.680GHz
  - Cores 8
    - Uniform core design
  - Threads 16
  - Base clock 3.0GHz
  - Max boost clock up to 4.6GHz
  - L3 cache 16MB
  - Memory PCIe 3.0
  - Supports Simultaneous Multithreading (SMT), with each cores supporting two threads
- **Integrated GPU**: AMD ATI Radeon Vega Series / Radeon Vega Mobile Series
- **Discrete GPU**: NVIDIA GeForce RTX 3080 Mobile / Max-Q (8GB/16GB)
- **RAM**: 32GB
- **Disk**: 1TB SSD
- **Shell**: zsh

- Using Arch Linux comes with a cost, the CPU pilots might not be very efficient, stable or up to date.

Matrix multiplication

🧪 Experiment Setup

We measured the execution time and derived performance index using:
  - **Static load balancing** (block-wise) because a matrix is a 2D array without any indication of the difficulty of the rows.

Each implementation was tested using 1, 2, 4, 6, 8, 10, 12, 14, 16, and 32 threads, with execution times measured in milliseconds (ms).
The performance index was defined as:
- **Performance Index = (1/Execution Time)**
This means that higher values indicate better performance.

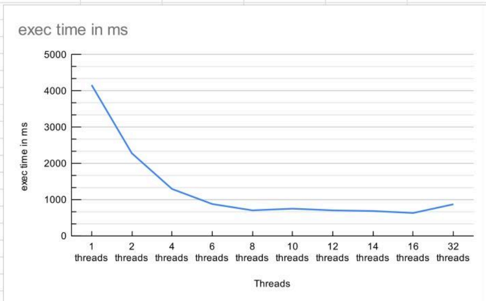
- All the tests were done with the same number of threads and the same end number.

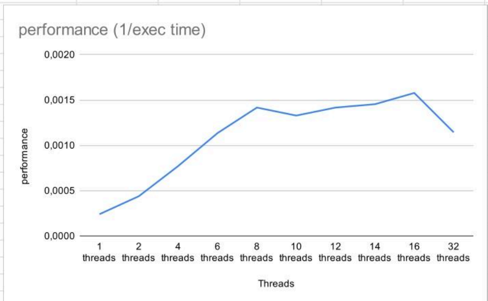
- The task size was evenly distributed between the threads (static block : rows / threads).
- The performance (1 / exec time) to have a better view of the evolution of the performance.
- The matrix size was set to 1000x1000.

📊 Performance Chart

exec time in ms	1 threads	2 threads	4 threads	6 threads	8 threads	10 threads	12 threads	14 threads	16 threads	32 threads
exec time in ms	4162	2280	1296	881	706	753	706	688	634	874

performance (1/exec time)	1 threads	2 threads	4 threads	6 threads	8 threads	10 threads	12 threads	14 threads	16 threads	32 threads
performance (1/exec time)	0,000240269	0,000438596	0,000771605	0,00113507	0,00141643	0,00132802	0,00141643	0,00145349	0,00157729	0,00114416





🧠 Interpretation of Results

- **Static block** decomposition divides the matrix into contiguous row blocks per thread. This is easy to implement and efficient when all rows take similar time to process.
- The performance improves significantly as threads increase from 1 to 8, leveraging parallelism on available physical and logical cores.
- From 10 to 16 threads, performance still increases slightly due to Hyper-Threading.
- At 32 threads, performance degrades because the number of threads exceeds the number of logical cores, leading to context switching and scheduling overhead.
- The best performance (lowest execution time and highest index) is observed at 16 threads, suggesting that this is near-optimal for the hardware used.

🏁 Conclusion

Using static block decomposition, the matrix multiplication program achieves good parallel scalability up to the hardware limit. The optimal performance is reached at 16 threads, with diminishing returns and overhead beyond that point. Static block partitioning is a good choice when matrix rows are uniform in complexity.

However, this method assumes perfect load balance, which is not guaranteed if some rows are more computationally expensive than others or if threads finish at different times. In such cases, dynamic work balancing (where idle threads can steal or request new tasks) would significantly improve efficiency, particularly at higher thread counts. This would reduce idle time and improve cache locality, potentially pushing performance beyond what static methods can achieve.