Problem1

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/pc_dynamic <number of threads> <number end number>
./bin/pc_static_block <number of threads> <number end number>
./bin/pc_static_cyclic <number of threads> <number end number>
```

- **The first three executables are for the first problem**, which is a prime number checker.
- The first three executables take two arguments: the number of threads and the end number.

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.

Prime number checker

🧪 Experiment Setup

We measured the execution time and derived performance index using:
- **Static load balancing (block-wise)**
- **Static load balancing (cyclic with task size 10)**
- **Dynamic load balancing (task size 10)**

Each implementation was tested using 1, 2, 4, 6, 8, 10, 12, 14, 16, and 32 threads, with execution times measured in milliseconds (ms).

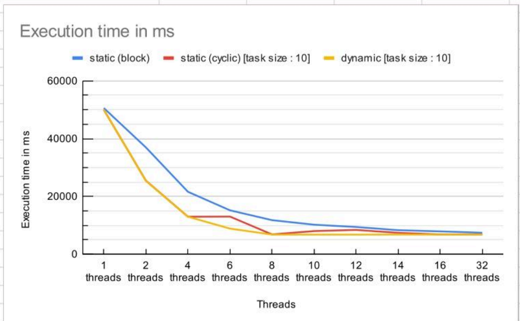
The performance index was defined as:
- Performance Index = (1/Execution Time) × 10,000
This means that higher values indicate better performance.

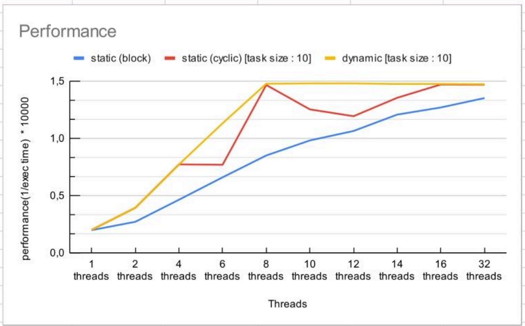
- All the tests were done with the same number of threads and the same end number.
- The end number was set to **1000000**.
- The performance (1 / exec time) was multiplied by **10000** to have a better view of the evolution of the performance.

📊 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
static (block)	50587	36957	21591	15175	11769	10194	9402	8288	7881	7402
static (cyclic) [task size : 10]	49985	25455	12960	13006	6825	7987	8385	7389	6805	6817
dynamic [task size : 10]	50044	25399	12964	8843	6774	6761	6762	6782	6781	6797

performance (1/exec time) * 10000	1 threads	2 threads	4 threads	6 threads	8 threads	10 threads	12 threads	14 threads	16 threads	32 threads
static (block)	0,197679	0,270585	0,463156	0,658979	0,84969	0,980969	1,0636	1,20656	1,26887	1,35099
static (cyclic) [task size : 10]	0,20006	0,39285	0,771605	0,768876	1,4652	1,25203	1,19261	1,35336	1,46951	1,46692
dynamic [task size : 10]	0,199824	0,393716	0,771367	1,13084	1,47623	1,47907	1,47885	1,47449	1,47471	1,47124





🧠 Interpretation

- **Static block** shows the worst scalability: the fixed partitioning results in some threads finishing early, causing CPU underutilization.
- **Static cyclic (task size = 10)** improves balance by distributing work in round-robin fashion. This reduces idle time and improves throughput.
- **Dynamic scheduling (task-stealing style)** performs the best or ties with static cyclic at almost all thread counts. It efficiently distributes remaining work to idle threads, particularly useful when some rows are more computationally expensive.
- **Beyond 16 threads, there's almost no performance gain** — this suggests we're reaching the limits of physical cores and memory/cache throughput.

🏁 Conclusion

- **The best performance is achieved with dynamic scheduling using 8 to 16 threads.**
- Static cyclic offers nearly the same benefits with slightly more predictable performance curves.
- Static block is easy to implement but inefficient for larger thread counts due to uneven workload distribution.
- The speedup stabilizes after 16 threads due to hardware constraints, indicating that further performance gains would require either different algorithms, GPU acceleration, or architecture-specific optimizations.