Project 3

Table of Contents

Introduction

This project was written in **C++** and uses **OpenMP**

Folders structure

```
├── xmake.lua
├── bin -> folder created after compilation
|   ├── prob1 -> first problem executable
|   ├── prob2-> second problem executable
├── problem1 -> contains the code for the first problem
|   ├── xmake.lua
|   ├── prob1.cpp -> problem 1 source code
├── problem2 -> contains the code for the second problem
|   ├── xmake.lua
|   ├── prob2.cpp -> problem 2 source code
```

Installation

Requirements

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

Installation

```
git clone git@github.com:GlassAlo/CAU_Multicore.git
cd CAU_Multicore/proj3
```

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/prob1 <scheduling type> <number of threads>
./bin/prob2 <number end>
```

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.

Problem 1

🧪 Experiment Setup

We measured the execution time and derived performance index using:
  - **Static load balancing with default chunk size**
  - **Dynamicload balancing with default chunk size**
  - **Static load balancing with a chunk size of 10**
  - **Dynamic load balancing (task size 10)**

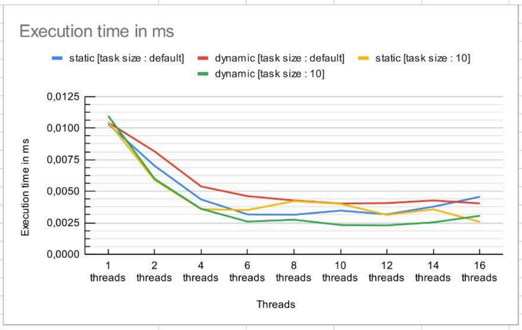
Each implementation was tested using 1, 2, 4, 6, 8, 10, 12, 14 and 16 threads, with execution times measured in milliseconds (ms).
The performance index was defined as:
- Performance Index = (1/Execution Time)
This means that higher values indicate better performance.

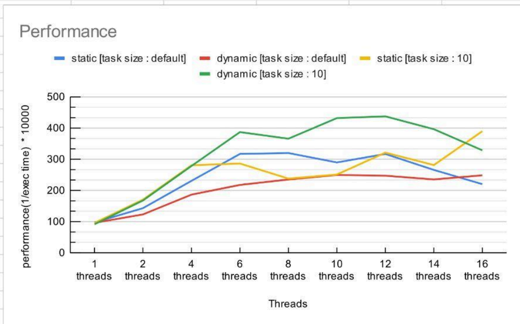
- All the tests were done with the same number of threads and the same end number.
- The end number was set to `200000`.

📊 Performance Chart

exec time in ms	1 threads	2 threads	4 threads	6 threads	8 threads	10 threads	12 threads	14 threads	16 threads
static [task size : default]	0,0103221	0,00700212	0,00434303	0,00315905	0,0031302	0,00345707	0,00316	0,00376892	0,00455093
dynamic [task size : default]	0,0104461	0,00814581	0,005373	0,00460315	0,00426197	0,00401306	0,00405192	0,00425911	0,00403094
static [task size : 10]	0,0104909	0,00587296	0,00356793	0,00349903	0,00421095	0,00399113	0,00311804	0,00356007	0,00256801
dynamic [task size : 10]	0,0109658	0,00597286	0,00359797	0,00258684	0,00273681	0,00231791	0,0022881	0,00252604	0,00304699

performance (1/exec time)	1 threads	2 threads	4 threads	6 threads	8 threads	10 threads	12 threads	14 threads	16 threads
static [task size : default]	96,87951095	142,8138907	230,2539932	316,5508618	319,4684046	289,2622944	316,4556962	265,3279985	219,7353068
dynamic [task size : default]	95,72950671	122,7625	186,115764	217,2425404	234,6332799	249,1864064	246,7965804	234,7908366	248,0810927
static [task size : 10]	95,32070652	170,2718902	280,274557	285,7934913	237,476104	250,5556071	320,7142949	280,8933532	389,4065833
dynamic [task size : 10]	91,19261705	167,4239811	277,934502	386,5720338	365,3889017	431,4231355	437,0438355	395,8765499	328,192741





📊 Detailed Observations

1. Execution Time (Lower is Better)

- All four scheduling strategies show significant improvements from 1 to 8 threads.
- **Dynamic (chunk size = 10)** shows the fastest execution at most thread counts beyond 4.
- **Static (default chunk size)** performs well until 8–10 threads, after which gains taper off.
- After 12 threads, **all methods plateau or degrade**, showing signs of overhead or saturation.

2. Performance Index (Higher is Better)

- **Dynamic with chunk size 10** consistently achieves the highest performance from 4 to 12 threads.
- **Static with chunk size 10** also shows strong performance but with more fluctuation.
- The **default static** and **default dynamic** approaches underperform with high thread counts, likely due to inefficient load balancing or lack of granularity.

🧠 Explanation of Results

✅ Why Dynamic Scheduling + Chunk Size 10 Performs Best

- **Better Load Balancing**: Tasks are dynamically pulled by idle threads, reducing idle time.
- **Fine Granularity**: Smaller chunks allow better utilization of threads, especially in unbalanced workloads like recursive tasks.

⚠️ Why Performance Drops or Plateaus at 14–16 Threads

- **Overhead of too many tasks**: With 200,000 elements and chunk size 10, you create 20,000 tasks — overhead increases with thread count.
- **Diminishing returns**: CPU reaches saturation around 8–12 threads, especially with hyperthreading.
- **Contention and synchronization costs** (e.g., `reduction`, task queue contention) rise with more threads.

🏁 Conclusion

Dynamic scheduling with small chunk sizes (10) provides the best performance in highly parallel divide-and-conquer tasks on modern CPUs with SMT.
Static scheduling performs reasonably well but lacks adaptability to runtime imbalances.
While increasing threads generally improves performance, **beyond 12 threads** there are **diminishing returns** due to overhead, memory bandwidth saturation, and synchronization costs.
The results validate that **task granularity and scheduling policy** are critical to achieving optimal parallel performance in OpenMP.