The perfectly parallel exercise

Zenny Wettersten[a,b]

[a] *CERN*
[b] *TU Wien*

Inverted Cern School of Computing 2024

Abstract—*Welcome to the `Totally Accurate DnD Simulator` exercise for the iCSC2024! In this exercise, your task is to try to use hardware acceleration to speed up the program "testSuite", which is a C++ program that simulates Dungeons and Dragons 5e combat to test the general balancing of the system. In this document you'll find the guidelines, prerequisites, and some general tips and hints as to how to go about the exercise.*

Contents

1. Introduction

Few things are as annoying as waiting, but when running large code bases that is really the only thing we can do. Whether we have a fixed problem size, or a fixed clock time, while the program itself is running we just have to wait for it to finish. However, in order to get rid of this waiting, or to make this waiting more fruitful, we can accelerate our code — that is, we can parallelise it. That is exactly what we will be doing today. In this exercise, your goal is to speed up the program *testSuite* as much as possible (preferably using some parallelism scheme).

2. Background

The tabletop role playing game system *Dungeons & Dragons Fifth Edition* (5e) has player characters (PCs) controlled by players, as well as non-player characters (NPCs) controlled by the game master. Although there are a bunch of ways for PCs to interact with the game world, our game *Totally Accurate DnD Simulator* is primarily focused around combat. Consequently, we've made a small test suite very originally called *testSuite* which, simply put, checks how likely it is that a PC will hit an NPC, and vice versa.

Although 5e has many different PC classes to choose from, for our test suite we've made four different pre-built PCs that reflect roughly how we expect players to progress throughout the game. These four are stand-ins for the four typical styles of play we expect to to see, and we expect to see that reflected in the balance tests *testSuite* runs. Generally, 5e is balanced around two assumptions:

- PCs hit an an NPC of the same level 65% of the time
- NPCs hit a PC of the same level 45% of the time

> **Note**
> While PCs have progression defined by their **level**, NPCs have a **challenge rating** (CR), which is roughly equivalent. An NPC of a given CR is intended to be an appropriate threat for a group of four PCs of the same level as the NPC CR. This is what we mean by "the same level", and we will not differntiate between level and CR henceforth.

Although this might seem pretty simple, different character archetypes won't fit this pattern to a tee — in fact, it is pretty unlikely that any given PC will fit this pattern perfectly. Additionally, while we know hew *often* attacks should hit, we haven't actually described *how* attacks are performed. In fact, the exact method will vary from PC to PC and NPC to NPC. However, they all build on the same foundation: Rolling a 20-sided die (1d20). The four classes used for the pre-built PCs in *testSuite* are described below. In general, whether an attack hits or not is given by whether

$$\texttt{1d20 + mod + prof} > \text{AC},$$

where `mod` is the relevant character statistic, `prof` is a number representing whether the character has *proficiency* with the given type of attack, and AC is the *armor class* of the target character (a measurement of how hard they are to hit). The relevant character statistics are *strength*, *dexterity*, *constitution*, *intelligence*, *wisdom*, *charisma*.

2.1. Barbarian

The barbarian class is your standard frontline warrior, who both hits and gets hit a lot. They have a lot of health points, and can reduce the amount of damage they take from attacks. Additionally, starting at level 2, they can increase the chance hitting enemies at the expense of increasing the risk of enemies hitting them:

- **Reckless Attack**
  Starting at level 2, when you make an attack using your strength, you can choose to roll 2d20 and take the higher result. When you do, any enemy that attacks you also rolls 2d20 and takes the higher result.

Thus, we expect barbarians to hit a lot more often than other PCs, but also to get hit a lot more often.

2.2. Cleric

Clerics can play a lot of different roles, but our pre-built cleric is expected to be a *healer*, meaning they can help their teammates rather than focusing on hitting enemies with attacks. Clerics usually use better armor than most other characters, so we expect them to be a bit harder to hit. On top of that, though, our pre-built cleric doesn't make attacks — instead, they use spells which force the target to make a *saving throw*. The enemy rolls this saving throw as stated below, and checks whether the result is lower than the cleric's *spell save difficulty*, which is calculated from the cleric's wisdom.

- **Saving throw**
  When a cleric attacks an NPC, the NPC has to roll `1d20 + mod`

`+ prof` and check whether the result is equal to or higher than the cleric save difficulty of `8 + wisdom + prof`. If it is not, the attack hits.

2.3. Rogue

For the purposes of *testSuite*, the rogue is the simplest class: They just roll a standard attack roll and add their dexterity and their proficiency, without any additional algorithmic complications. This does not exactly reflect their full functionality in 5e, but it is a sufficient simplification.

2.4. Wizard

Like the rogue, the wizard is very simple — although instead of attacking with weapons, they use their magical spells, but unlike clerics these spells actually make attack rolls. To determine whether an attack hits, the wizard rolls 1d20 and adds their *intelligence* and their proficiency.

3. Prerequisites

The first thing you need in order to do this exercise is the git repo (url https://github.com/zeniheisser/parallelism_exercise). If you're using the terminal to clone the repo, you can pull the code base with the

```
git clone
https://github.com/zeniheisser/parallelism_exercise.git
```

or

```
git clone
git@github.com:zeniheisser/parallelism_exercise.git
```

command, and if you additionally want to try making your own FlameGraphs you can also download Brendan Gregg's FlameGraph repo alongside the exercise by using the command `git clone -recurse-submodules URL`.

To actually compile *testSuite*, you need:

- g++, version 9 or later
- OpenMP library, `libomp-dev`

where `g++` is necessary to compile at all*, and `libomp-dev` is necessary to compile with multithreading. If you want to generate the heatmaps, you also need:

- Python 3
- (matplotlib)
- Pandas
- Seaborn

which are not needed for parallelisation, but are useful to check *testSuite* statistics, and make some very pretty pictures.

> **Installing Python packages**
>
> If you do not have Pandas or Seaborn on your machine, they are easily installed using **pip**. Simply run the command `pip install pandas seaborn` in your Python environment. If you do not have **pip** available either, **the documentation is very accessible**.

If you also want to try out making FlameGraphs, you need to make sure you have a FlameGraph visualiser on your machine. Additionally, you will need:

- perf

which is (most often) what you would use to get the stack traces used for FlameGraphs.

*If you want to use another C++ compiler, you're free to modify the makefile!

4. Getting the testSuite up and running

4.1. Compilation

Once you have the necessary tools installed and the code base cloned, compiling testSuite is as simple as writing `make` in your command line interface. This will compile the three files `rng.cpp`, `dndSim.cpp`, and `testSuite.cpp` into the executable file *testSuite*.

> **Note**
>
> If you aren't going for the bonus exercises, the only file you will need to look at is `testSuite.cpp`. For the bonus exercises, which files are relevant will be mentioned alongside the problem description.

The makefile allows for four different "make commands":

- `make`
  Compiles the three files with the flags `-std=c++17 -g -O0`.
- `make parallel`
  Identical to `make`, but adds the flag `-fopenmp`.
- `make clean`
  Removes the files `rng.o`, `testSuite.o`, and *testSuite*.
- `make cleanall`
  Removes `rng.o`, `dndSim.o`, `testSuite.o`, *testSuite*, and all `.csv` and `.png` files in the directory.

> **Note**
>
> We compile without any compiler optimisation (`-O0`). This means any speedup you see when running `make parallel` is exclusively from multithreading. In one of the bonus exercises, you can check whether compiler optimisation would be better than just multithreading.

4.2. Running

To run *testSuite*, in your command line interface you just need to type

```
./testSuite n
```

where `n` is an integer. *testSuite* will then run `n` attack simulations for each pre-built PC for each character level for each enemy level, both attacking and defending. More simply put: For each of the four classes, for each player level between 1 and 20, for each enemy level between 1 and 20, we test whether the player hits the enemy and if the enemy hits the player `n` times.

After a successful run, you should see some text in your command line interface. The final line should say *"Time taken: x ms"*, where *x* is the total runtime of the measured section of the code. Additionally, you should now have eight different `.csv` files in the run directory, representing the probability distributions of player characters hitting enemies and enemies hitting player characters. Then, to generate the heatmaps, you run the command

```
python plotHitRate.py
```

after which the beautiful `.png` files should also be available in that same directory.

If you also want to try making your own FlameGraph, you just need to run the command

```
./flamegraph.sh
```

where you will likely discover that you do not have access to frame pointers or something similar. Googling the warning message should easily give you a solution for this.

By the way, the exported `.svg` FlameGraph file is intractable — try opening it in your browser!

Note

The runtime written by *testSuite* isn't the full program runtime, but the time it takes to initialise all the necessary vectors/arrays and to then evaluate the hitrates. We ignore the time taken by any text written to the command line interface, as well as the time taken to export the `.csv` files. If you calculate speedup using this output, you will get the Gustafssonian acceleration — if you want to get Amdahlian acceleration, you may want to move the time measurement variables *t1* and *t2* to the top and bottom of the `main` function, respectively.

5. Exercise

5.1. Profiling

In the git repo, we've included a FlameGraph (`flamegraph.svg`) for *testSuite* showcasing where the runtime is spent[†]. A screenshot of this FlameGraph is given in Fig. 1. As we can see in the FlameGraph, more than half of the runtime is spent in the different types of attack functions, suggesting that this is indeed the part we should prioritise parallelising.

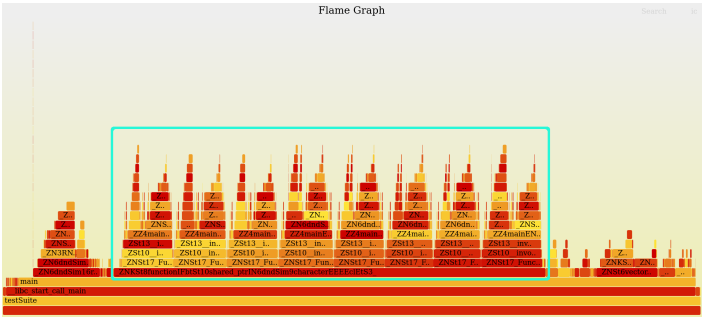

Figure 1. FlameGraph to profile *testSuite*, where the attack functions (making up ∼62% of runtime) have been highlighted.

5.2. Multithreading

For this exercise, you need to write exactly *one*(!) line of code. The line in question is

```
#pragma omp parallel for
```

which tells the compiler that the next `for`-loop should be split across all available threads on your machine using the OpenMP library. As noted, this is not SIMD parallelism — technically, it is MIMD multiprocessing — but it can speed up the code by quite a lot[‡], and it showcases the "loop unfolding" method of thinking you should use for "truly" data parallel implementations.

5.3. Task

First and foremost, you need to determine where to put the `pragma` mentioned in the previous subsection. As mentioned, the `pragma` tells the compiler that the next `for`-loop should be multithreaded — play around and see what happens when multithread different loops! How much can you speed up the program just with multithreading? Although we said you'd only need to write one line of code, you can of course add the `pragma` to each of the loops separately. What's the effect when multithreading different loops?

Once you have *testSuite* running multithreaded, you can start looking at the actual loops themselves. Are all these different loops actually necessary? Could the loop order be optimised? Could some loops be combined? Could some parts be reused between loops without worsening statistics in any notable fashion?

[†]Note that the horizontal axis in the FlameGraph represents runtime, but is sorted alphabetically rather than chronologically.

[‡]On my local machine, just moving some loops and compiling with OpenMP I could reduce the runtime for fixed `n` by ∼55% without writing any new code or any compiler optimisation.

Level	Total	Normal	Spell.
1	199	160	39
2	298	200	98
3	260	192	68
4	192	130	62
5	261	191	70
6	144	90	54
7	131	78	53
8	128	83	45
9	121	71	50
10	90	56	34
11	82	46	36
12	74	39	35
13	59	34	25
14	49	28	21
15	52	23	29
16	40	24	16
17	35	15	20
18	33	20	13
19	16	10	6
20	21	14	7

Table 1. The number of NPCs at each level between 1 and 20, ordered by level. The second column tells the total number of NPCs at the given level, while the third and fourth columns detail the number of "normal" NPCs and "spellcaster" NPCs, respectively. Spellcasters are defined as NPCs that cause the PC to make a saving throw, rather than making its own attack roll.

Note

Remember that you need to compile using the `make parallel` command to use OpenMP multithreading — if you just run `make`, the compiler will ignore the `pragmas`. To recompile with OpenMP, you might need to run `make clean` first.

As a sidenote, `random_encounter(int lvlNPC)` generates a random NPC from the list of NPCs in `all_monsters.txt` at the given level. As reference, the number of distinct monsters at each level is given in Table 1, where we also note that `random_encounter` additionally takes a `string` as argument, which allows the user to choose between spellcasting NPCs (defined as NPCs whose attacks cause saving throws, rather than making their own attack rolls), regular NPCs (NPCs who make standard attack rolls), or any type of NPC using the arguments `"spellcaster"`, `"regular"`, or `"any"`, respectively[§].

The single goal of this exercise is to refactor and multithread *testSuite* in order to speed it up as much as possible. Once you feel like you've gotten a feel for how to structure your code to maximise concurrency, minimise inter-thread communication, and get rid of superfluous function calls, you can consider yourself done! Good job!

6. Bonus exercises

Following you will get some additional possible continuations to the main exercise, which you can try out if you'd like to work further on understanding optimisation and parallel programming. If you aren't familiar with C++, compilation flags, or data management, they might prove to be a welcome introduction to the subject(s).

6.1. Compiler optimisation

As you might have noticed, the makefile uses the flag `-O0`, meaning we explicitly tell the compiler to do no optimisation whatsoever — the

[§]Using any argument other than `type = "any"` is irrelevant for multithreading, but you can try to set up the structure necessary for a proper SIMD implementation if you'd like.

final executable will function exactly in the manner defined by the original C++ code. Since the average compiler is quite a bit smarter than the average programmer, we might expect compiler optimisation to be better than our direct OpenMP implementation. So let's try it!

To change the compiler optimisation level, you just need to change a single character in the second line of the makefile,

```
CXXFLAGS = -std=c++17 -g -O0
```

where the flag you may want to change is `-O0`, which you can change to `-O2` or even `-O3`.

Having done this you might notice that when you enable compiler optimization, you might notice that the compilation suddenly takes an awful long time. In fact, the code might not even compile at all. Specifically, it seems to get stuck when compiling `dndSim.cpp` into `dndSim.o`. This is pretty weird, considering `dndSim.cpp` is only slightly larger than `testSuite.cpp`, no?

If you take a look into `dndSim.cpp`, you might find the offending part — but if you don't, let us reveal it: Line 403 of `dndSim.cpp` reads

```
#include "all_monsters.txt"
```

meaning that at compile time, the entirety of `all_monsters.txt` is loaded into `dndSim.cpp`. Considering the former is roughly 20 times larger than the latter and exclusively initialises all the NPC objects, meaning there is very little optimisation to be made[¶]. So, if we want to get some compiler optimisation, we will probably need to stop compiling `all_monsters.txt` with the same flags as the rest of the code.

To separate the compilation between `dndSim.cpp` and `all_monsters.txt`, you will need to make `all_monsters.txt` into a separate compilable object which you can then link *testSuite* against, while ensuring that all the any objects accessed outside of `all_monsters.txt` are declared within any scope that accesses those objects. That is to say, you need to make `all_monsters.txt` into a separate `.cpp` file, and then ensure that `dndSim.cpp` is aware of the existence of the three vectors `monsters`, `spell_monsters`, and `non_spell_monsters`, since these vectors are what are actually called in the `random_encounter` functions.

To make sure that your `all_monsters` C++ implementation recognises all the functions used to initialise the NPCs, you will need to include the `dndSim.h` header file within the implementation, since that is where everything used for the actual NPC declarations are pre-declared. Additionally, you will need to tell `dndSim.cpp` that the relevant vectors exist and will be available at runtime — this is done by pre-declaring them (while making sure you have your namespace scopes set correctly), e.g. by including something like

```
extern
std::vector<std::vector<std::shared_ptr<dndSim::npc> >
> monsters;
```

or similar, etc. You then also need to ensure that you have compilation rules for `all_monsters` inside your makefile, and in order to compile the different `.o` files with different compiler flags you'll also need to set the compilation flags for `all_monsters` and the rest of the code separately. Don't forget to add `all_monsters` to the `make clean(all)` commands!

If you get this compilation chain to work, you should be able to compile the actual executable *testSuite* with stronger compiler optimisation, without having to wait an eternity for `g++` to realise that it can't actually optimise the monster instantiations all that much. Voila! Maybe the compiler can outperform a direct OpenMP implementation? Maybe you can use both for an even better speedup? The world is your oyster.

[¶]We as programmers recognise that `all_monsters.txt` probably can't be very optimised, but **the compiler doesn't know that**. The compiler will try to optimise **all** the code.

6.2. Pre-processing

The reason we didn't use SIMD vectorisation in the main exercise (aside from the fact that not all modern consumer-grade CPUs support vector instructions) is that vectorised random number generation is has far less support than other vector instructions — and since the central function call of *testSuite* is simply a random number generator alongside an integer comparison, vectorised implementations of the `attack` functions would be a lot more convoluted than compiler automatic vectorisation. That integer comparison, though — there, we just check which of two integers is greater. That should be easily vectorised, right?

The idea of this bonus exercise is to restructure the `rng.cpp` implementation generate all the random numbers prior to any of the actual numerical evaluations, which should make it possible to to vectorise the attack functions themselves. Although this *pre-processing* itself will probably cause a slight slowdown due to the additional memory management, we can hope that the vectorised speedup in the attack functions themselves will outweigh this overhead.

If you take a look into `dndSim.cpp` and `rng.cpp`, you might notice that there are several different versions of the `roll1d20()` (and `roll2d20dl()`) routines — one generic one and one for each of the classes (as well as a generic `roll2d20dl()` routine alongside one for the barbarian class). This means that we could replace the dice functions while keeping track of how they will be used at runtime, i.e., we can keep track of *how* and *when* each random number is used. When attacking an NPC, the PC will use their own die function unless they cause a saving throw, in which case the generic function `roll1d20()` will be used. When an NPC attacks a PC, the PC die function is always used.

To implement a pre-processing procedure for *testSuite*, you're going to need to override the dice functions so that instead of generating a random number at call time the random numbers have already been generated and are then returned by the dice functions themselves. You can do this either by function overloading or compiler time `if`-statements.

> **Function overloading and `ifdef`s**
>
> C++ supports so-called function overloading, where different functions can use the same name and instead be distinguished by argument type. Thus, you could e.g. override `roll1d20()` by defining a function `roll1d20(int index)`, which instead of generating a random die roll instead accesses a pre-defined vector `pre_rolled` and returns the value `pre_rolled[index]`.
> Alternatively, you could use `#ifdef` to set which part of the codebase to actually compile. For example, if you write a new version of `roll1d20()`, which is wrapped by something like `#ifdef PREPROC` (where the original version of the function should of course be included in the `#else` branch), you can then call the `make` commands with the additional flag `-DPREPROC` to tell the pre-processor that `PREPROC` is defined for this particular compilation.

Once you have altered versions of the dice functions, you need to make sure you actually load up the data in advance of any call to the dice functions, while making sure the the relevant vectors are accessible to the relevant dice functions. You can most simply do this by adding a function to `rng.cpp` which initialises these pre-processed vectors and randomly generates the relevant numbers (integers between 1 and 20) with the correct distribution (uniform, or `roll2d20dl()` for *some* (but not all!) calls to the barbarian). Note here that in the default version of *testSuite* we **do not know a priori how many times `barb_roll1d20()` and `barb_roll2d20dl()` are called**, since whether an enemy makes an attack or causes a saving throw is random.

Note

Although the barbarian class in 5e can roll 2d20dl for strength saving throws, none of the implemented NPCs in our game target the strength statistic. Thus, any `type="spellcaster"` type enemy will use `barb_roll1d20()`, while any `type="regular"` enemy attacking *a barbarian* ***of at least level 2*** will use `barb_roll2d20dl()`. If you want to make sure you don't generate too many random numbers, you can explicitly separate `"regular"` and `"spellcaster"` enemies into two different loops — otherwise, you can of course just generate the maximum possible amount of random numbers for each of the functions.

With the actual pre-processing functions written, you just need to make sure that the random number generation itself is called at the top of your `main` function, and now you have a program where all the random number generation is separated into a pre-processing routine. Great job!

6.3. Vector instructions

For the final bonus exercise, let's try to get some vectorised instructions running in our program. If you haven't done bonus exercise 2, you will probably not be able to get SIMD instructions running in the simulation loop — but there are some good news: There's a different part of the code that does simple integer addition and floating point division!

If you haven't touched the accumulation loops on lines 201 through 218 of *testSuite* (in the current version of the git repo), you can take a look there and see that the only things done are tallying up the total amount of successful attacks, before normalising it to a probability. These `for`-loops are of course embarrassingly parallel across all $2 \times 4 \times 20^2$ iterations, so vector instructions might be applicable here.

Although we might simply try to vectorise these loops with compiler flags (which you can add in the `makefile` by simply appending or modifying the value of `CXXFLAGS`), in order to get a proper understanding of how and if SIMD vectorisation works, I would recommend that you use some explicit directive. Since we're using OpenMP for the main exercise, we'll remind you that OpenMP has a directive for explicit SIMD vectorisation,

```
#pragma omp simd
```

which will apply SIMD instructions to the next `for`-loop in the program. If you've already done bonus exercise 2, you should be able to add this to the simulation loop(s), but if you jumped straight here you should nevertheless be able to add this to the accumulation loop(s), although you are unlikely to see much measurable acceleration in that case.

Best of luck!

7. Closing thoughs

If you've gotten this fair, that means you've either finished the main exercise or maybe even the bonus exercises (or you're reading ahead, in which case, very studious of you! Sadly, there's no secret cheat code down here to get everything working on your first try.). In the former case, great job! If you want to learn more about the practical details of writing optimised, heterogeneous, and parallel code I advise you to take a look at the bonus exercises, which go a lot deeper into the details of writing a vectorised program. In the latter case — amazing! You're (one of) my favourite student(s), and may proudly present that on your CV.

Although this exercise has mostly been about getting any level of parallelism working on completely sequential code, it should have given you an understanding for task- and data-parallelism, and how you might go about applying it to your own problems. Feel free to contact me if you have any thought or questions, if you have ideas for porting your own code to heterogeneous or parallel architectures, or if you happen to need a bass-baritone singer for your choir or musical theatre troupe. And if not, I'm happy you joined this exercise session and did this exercise, and I hope that you've learned something you might actually use in your own work or for any personal hobby projects you have on the side. Than you so much for attending, and happy coding! :-)

Contact:
513/1-014, CERN, Meyrin Site
zenny.wettersten@gmail.com
zeniheisser