Parallel Design Pattern
Coursework Part Two

s1932851

9 Mar, 2023

1 Introduction

This report presents an approach to parallelise the serial code for reactor simulation using geometric decomposition pattern as the algorithm strategy, and SPMD as the implementation strategy.

2 Serial Code

2.1 Code understanding and Profiling

Key data structures There are two main data structures used in the serial program. (1) an array of active neutrons, with each neutron holding its own properties such as position and energy. (2) a reactor core structure composed of different channels with their contents.

Main areas of computation There are two main areas of computation organised around the manipulation of the two key data structures at each time step of the simulation process.

1. Update neutrons: iterate through each active neutron, identify which channel in the reactor core it travels to and update its properties (either slows down or becomes inactive) as a result of interaction with the reactor core channel contents.

2. Update reactor core: iterate through each channel in the reactor core structure and handle its updates based on channel type.

Profiling To understand the computational bottleneck of the serial program and identify which tasks are computationally intensive, we perform code profiling using `gprof` and run the experiment using one CPU core on Cirrus. The configurations used are 10 nanosecond per timestep, 10^4 number of timesteps, 2×10^5 maximum neutron limit, with other settings being default. Some key profiling results are summarised in Table 1.

function name	**% time**	**seconds**	**num calls**
`updateNeutrons`	68.36	32.64	10000
`MeVToVelocity`	20.49	9.78	1999800000
`determineAndHandleIfNeutronFuelCollision`	7.69	3.67	97952457
`getNumberNeutronsFromGenerator`	1.86	0.89	10000
`initialiseNeutron`	0.50	0.24	8708292
`determineAndHandleIfNeutronModeratorCollision`	0.10	0.05	175416
`fissionU236`	0.04	0.02	7105079

Table 1: Profile summary of serial code

From Table 1, we observe that tasks related to updating neutrons contributes to a large portion of program runtime, around 68%. Within those tasks, computation of calculating velocity is performed on every active neutrons, which roughly contributes to 20.49% of total runtime; function to handle neutron and fuel collision takes up 7.69% and this is performed whenever a neutron travels to a fuel assembly channel. On the other hand, tasks related to updating reactor core do not take up much runtime. For example, function to handle fission of U236 only runs in 0.04% of total runtime.

To summarise, since most computationally intensive part of the reactor simulation program is organised around manipulating an array of neutron data, it makes sense to develop a parallel approach driven by decomposing the array of neutrons.

2.2 Finding Concurrency

In this section, we aim to answer the following questions to identify and analyse exploitable concurrency of the simulation program, from the perspective of decomposing the problem by neutrons.

1. what tasks can be executed concurrently?
2. what data are local and global (shared) to each task?
3. what dependencies among tasks need to be handled?

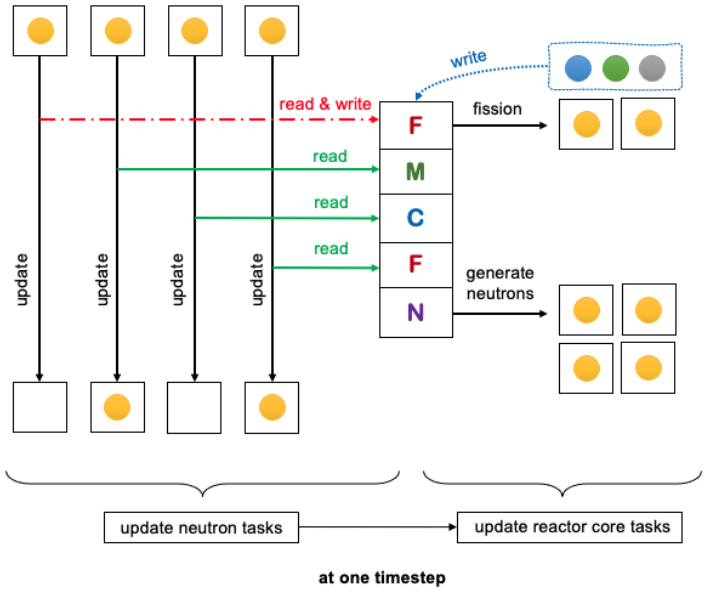

Figure 1: Illustration of neutron update and reactor core update workflow with data dependencies at one timestep of the simulation process.

Here, we define the task of updating a neutron being an iteration of the loop over the array of neutrons at one timestep of the simulation process, which includes locating the neutron's position, checking its interaction with reactor core channel, and updating its properties. For each task, the neutron data (its energy, current 3D position, activeness) remains local to the task and no information about other neutrons is required. However, every task needs read and write access to the reactor core data structure. More specifically, it needs to read channel contents data to check interaction and based on which to update the neutron's properties; Write will only be made to some specific fuel assembly chemical quantities, i.e. decrements U235 or Pu239 and increments U236 or Pu240, when neutron absorption happens. Theoretically, such writes could possibly affect the absorption probability for the next neutron because the amount of U235 or Pu239 atoms present

in a particular pellet of certain fuel assembly channel, where the neutron arrives at, is used to calculate the probability. However, this dependency among neutron update tasks will only come into effect, if, at one timestep, more than one neutron travel to the exact same pellet of a certain fuel assembly channel and at least one such neutron has gotten absorbed (because if no absorption has ever happened, no writes are made, so does not matter). Whilst this situation is possible, empirically it will not happen that often[1]. Therefore in our approach, we make the assumption that such dependency observed among neutron update tasks is safe to ignore. Since there will be no dependency between neutron update tasks given this assumption, we can conclude that neutron update tasks can be executed concurrently, with the only shared data being the reactor core.

Next we look at the task of updating reactor core. This task can be further broken down into two types of tasks: (1) task to update every fuel assembly channel. This involves fission of all present U236 and Pu240 atoms that have been generated during neutron update tasks and updating the chemical quantities of all byproducts of the fission process; it also creates new active neutrons that will be processed in the next simulation timestep. Data-access wise, this task needs to read from and write into the reactor core data structure, and also writes into the array of active neutrons. Dependency wise, it relies on how many neutrons have been absorbed and hence how many fissionable atoms, i.e. U236 and Pu240, have been produced during the current timestep; (2) task to generate new neutrons by generator. This task will generate a fixed amount of neutrons (the exact amount is deterministic and is based on the generator configuration), and does not require access to other data structures. Plus, this task does not have any dependencies with others.

To conclude, neutron update tasks can be executed concurrently with a need to access the global reactor core data and the local neutron data owned by each task. Reactor core update tasks are dependent on the changes made in the reactor core data from earlier neutron update tasks. Figure 1 provides a conceptual image of such concurrency and dependency. This would serve as the blueprint to design our parallisation approach based on Geometric Decomposition pattern.

3 Geometric Decomposition

3.1 Solution Overview

We decompose the 1D array of neutrons into chunks of 1D sub-arrays to be our problem subdomains, and each chunk is owned by a UE. This is valid since each neutron can be operated on independently as we discussed in section. To get the size of the sub-array, we divide the size of the original array by the number of UEs. If this doesn't divide evenly, we handle it by adding an extra element to certain UEs, i.e

```
/* UE_rank is the rank of current UE */
int subarray_size = original_array_size / num_UEs;
if (UE_rank < original_array_size - subarray_size * num_UEs)
{
   subarray_size++;
}
```

The size of a sub-array is calculated using the size of the original array specified in the configuration (i.e. `MAX_NEUTRONS`) and an array of neutrons of this size will be initialised on each UE at the start of the program. For more detailed implementation strategy, see following sections.

Since the two main tasks in our problem - updating neutrons and updating reactor core needs access to the reactor core data, we replicate this data on each UE in a distributed-memory environment. During one timestep of simulation process (*before* moving on to the next), no exchange interaction between subdomains is required. This is because updating neutron tasks are local to those neutrons on each UE (now that

[1]However, this is subject to the number of pellets in a fuel assembly channel, the number of total fuel assembly channels in a reactor core, and potentially the nanosecond defined for each timestep. The more pellets and fuel assembly channels we have, the longer a neutron travels, the less likely an *actual* dependency situation will happen.

reactor core is replicated on every UE) and subsequent updating reactor core tasks are only responsive to those changes (generated fissionable atoms) made by the local neutrons. Therefore, the two main tasks also remain local within each timestep.

However, the accumulative changes made in each reactor core by all neutrons across UEs need to be aggregated and synchronised before the beginning of the next timestep of simulation. This is because, the reactor core states at the start of one timestep is derived as a result of performing the two main tasks with respect to *all* active neutrons from previous timestep. In our approach, we conduct an all reduce of changes made in the reactor core across all UEs at the end of each simulation timestep. For reporting purpose, the number of fissions happened in every reactor core also needs to be aggregated and synchronised. For more details on how this all reduce operation is handled efficiently in our implementation, see following section.

Regarding task scheduling, we map the two main tasks to every UE so that the simulation process associated with the local neutrons remains local to each UE as well.

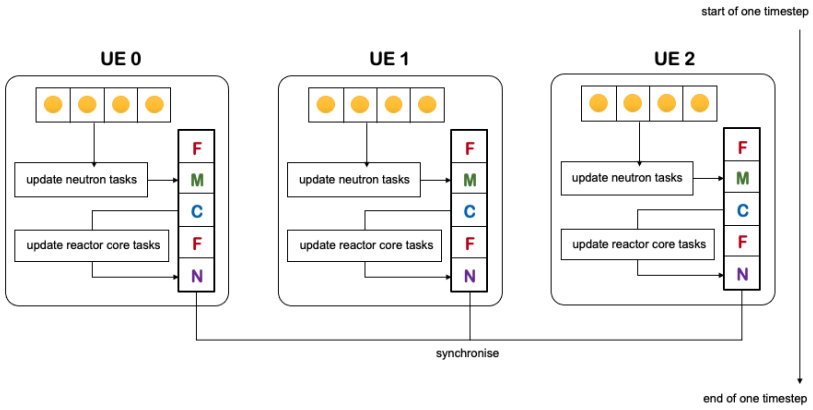

Figure 2

To summarise, an overview of the geometric decomposition algorithm strategy is shown in Figure 2, and the pseudocode[2] of the parallelised simulation program is provided below.

```
/* Parallelised simulation using Geometric Decomposition pattern */
main()
{
    parseConfiguration();
    initialiseReactorCore();
    initialiseLocalNeutrons(); // new in parallel version
    for (int i = 0; i < num_timesteps; i++)
    {
        updateNeutrons();
        updateReactorCore();
        synchReactorCore(); // new in parallel version
        synchNumOfFissions(); // new in parallel version
    }
}
```

[2]Function names used here are for demonstrating algorithm logic purpose, they do not necessarily match actual names used in the submitted code.

3.2 Local Neutrons Initialisation

The function `updateNeutrons(configuration)` in the serial program uses the number of maximum neutrons, i.e. `MAX_NEUTRONS` specified in configuration to initialise an array of neutrons. To reuse this function in the parallel code, we create a new function to modify the configuration by calculating the local sub-array size and substituting that in the configuration structure, i.e.

```
/* Substitute local neutron size in configuration */
void modifyConfig(config, int num_processors, int myrank)
{
   int local_neutron_size = calculateSubarraySize(num_processors, myrank);
   config -> max_neutron = local_neutron_size;
}
```

In this way, all the functions provided in the serial code that manage the dynamics of neutrons, such as using a smart indexing mechanism to keep track of inactive and active neutrons, can be preserved and reused as is in the parallelised program.

3.3 Reactor Core Synchronisation

The reactor core synchronisation is where most communications take place in our parallel program, and hence its implementation is crucial to how much communication overheads and workload will occur in the program. The key considerations in our implementation are

1. Reduce the number of communications needed as much as possible
2. Organise the data to be communicated in a large single data structure if possible
3. Synchronise the communication timing as much as possible

Before we present our approach, we will start with reviewing how exactly changes are made in the reactor core, more specifically fuel assembly channels, from a data structure perspective to motivate the design of the data structure used for communication. Figure 3 shows an example of how overwrite in the fuel assembly channel happens when a neutron gets absorbed.

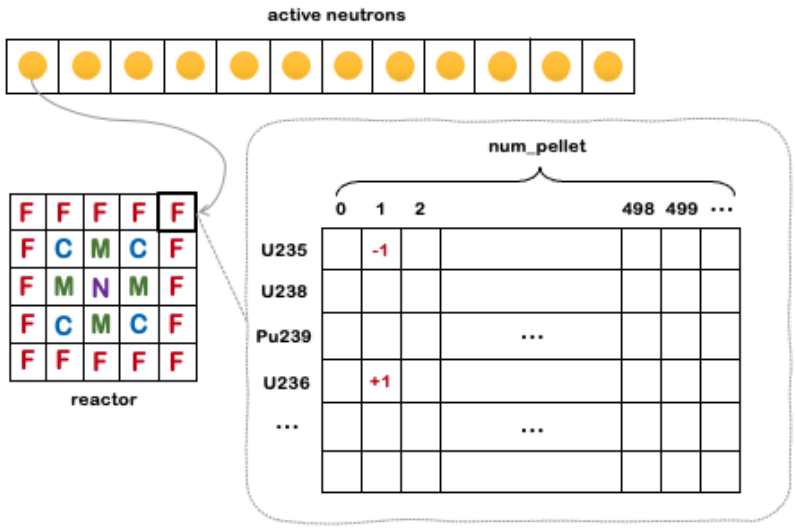

Figure 3: Changes made to the fuel assembly channel contents when a neutron collides with a U235 atom at pellet 1 and gets absorbed.

As can be seen from Figure 3, all fuel assembly channels keep track of a 2D array of chemical quantities, indexed by chemical id and pellet id. Whenever a change happens, it writes into the corresponding entry in this array. Therefore, if we can have a single big array that concatenates these 2D chemical quantities arrays of all fuel assembly channels and keeps track of the *delta* of all entries, let's call it *delta array*, then we will meet the second consideration stated above. Figure 4 shows a conceptual[3] image of this delta array.

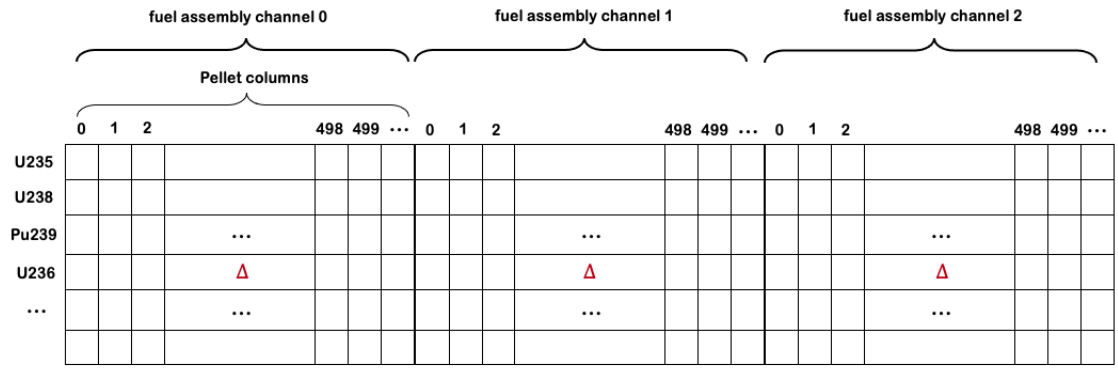

Figure 4: Delta array data structure

Since the delta array records the quantity delta of each chemical in each pellet of each fuel assembly channel on every UE, now synchronisation of reactor cores across UEs becomes a matter of reducing (allreduce) all delta arrays from all UEs, obtaining a reduced delta array that holds the total quantities changes of each entry[4] on every UE, and applying this change to the reactor core. This entire process is illustrated in Figure 5.

To provide more implementation details, as is illustrated in Figure 5: We first take a snapshot (make a copy) of the reactor core at the start of one simulation timestep, here since only fuel assembly channels need to be synchronised, we only copy the contents of all fuel assembly channels in this snapshot; This snapshot copy is used for calculating the change for all (chemical, pellet, channel) entry once both updating neutron tasks and updating reactor core tasks finish. The calculation is done by taking the difference of the contents of the snapshot reactor and the actual reactor which gets overwritten during simulation. The difference is then recorded in the delta array. Note that since this delta array is conceptually 2D (thus can easily locate chemical and pellet), we need an extra parameter to identify the id of a fuel assembly channel. This is done by having a separate data structure to store the mapping between the lcoation of a fuel assembly channel in the reactor core and its artificially assigned id. (see Figure 6 for further clarification) The delta array across all UEs are all reduced and the reduced result is applied to the snapshot copy, which will be then copied into the *actual* reactor. In the end, the delta array will be reset to 0 to be used for next timestep.

The pseudocode of the parallel program can now be refined to the following.

```
/* Parallelised simulation using Geometric Decomposition pattern */
main()
{
    parseConfiguration();
    modifyConfig(config, int num_processors, int myrank)
    initialiseReactorCore();
    initialiseNeutrons();
    initialiseReactorCopy(); // new data strcuture
    initialiseChemicalDelta(); // new data strcuture
    setFuelAssemblyIndex(); // new data strcuture
```

[3]In the submitted code, this array is implemented in 1D, with indexing logic to calculate the index of a corresponding (chemical id, pellet id, fuel channel id) triplet.

[4]By entry, we mean a triplet of (chemical id, pellet id, fuel assembly channel)

```
    for (int i = 0; i < num_timesteps; i++)
    {
        updateNeutrons();
        updateReactorCore();
        calculateChemicalDelta(); // this writes into fuel_assembly_deltas
        MPI_Allreduce(fuel_assembly_deltas); // all reduce

        applyDeltaToReactor();
        copyFuelAssembly();
        resetChemicalDelta();
        /* .. some work to synch number of fission .. */
    }
}
```

As can be observed from the pseudocode, this synchronisation approach has successfully meet our initial considerations. Communications to synchronise fuel assembly are only carried out once at each timestep on every UEs, through MPI Allreduce. The timing of communication is at the end of all simulation tasks, which synchronises well among UEs. (We will see a profiling for this in the profiling section) Moreover, the code related to reactor core synchronisation does not interfere with the serial code logic, which makes the implementation very decomposable.

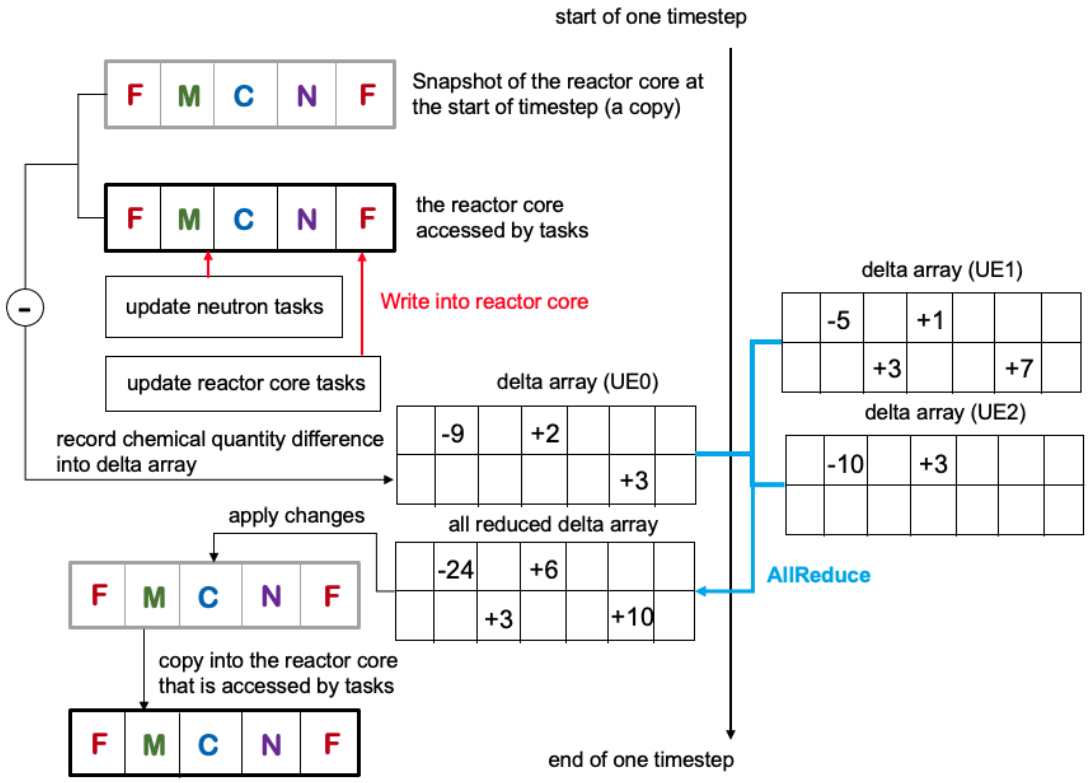

Figure 5: Synchronisation of reactor core

3.4 Number of Fission Synchronisation

Synchronisation of number of fission are relatively easier. We implement this by manipulation configuration and conducting an all reduce of the number across UEs.

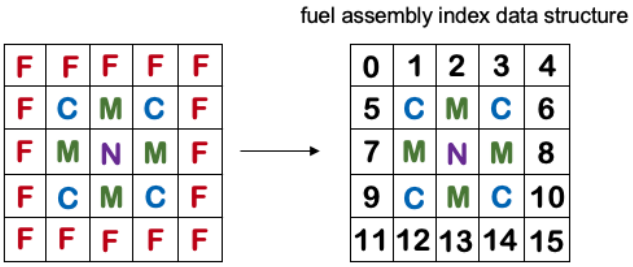

Figure 6: A data structure to index fuel assembly channels

3.5 Neutron Generator

Note that neutron generator only operates once during reactor core update tasks at each simulation timestep, and the amount of neutrons generated is a constant value. We divide this constant by the number of UEs so that the total number of these neutrons generated concurrently across all UEs at each timestep is the same as in the serial code. To be more concrete, we modify the weights of neutron generator specified in the configuration to achieve the aforementioned calculation result, and in this way we do not need to modify existing neutron generation code from the serial program.

3.6 Correctness

Since we reuse all the functions from the serial program to perform the core reactor simulation (benefit of decomposability!), we only need to make sure the calculation and synchronisation of delta array are correctly implemented. We provide some printing functions in the submitted code which we used for verifying intermediate results to ensure correctness.

4 Strong Scalability Test

In this section, we explore the performance and scaling properties of our parallelised solution. For experiment setup, all scaling tests are performed on a tier-2 national HPC facility Cirrus.

parameter	**value**
NS per timestep	10
total number of timesteps	10^4
display progress freq	100
maximum neutron limit	10^7
collision probability multiplier	1000
write reactor freq	1000

Table 2: Strong scaling problem size

We use the following configuration for the strong scaling test, as shown in Table 2. We run our parallel program using 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048 CPU cores (UEs) and record the program total execution time respectively. In Figure 7, we summarised the trend of speed-up and parallel efficiency by increasing the number of UEs. We observe from Figure 7 that, with the above configuration, the speed-up gained from parallelisation reaches its peak around using 1024 UEs, achieving a speed-up of 66.48 from the serial code. The efficiency graph on the right of Figure 7 shows that the more UEs we have, the more communication occur and hence the resultant overhead would reduce the parallel efficiency.

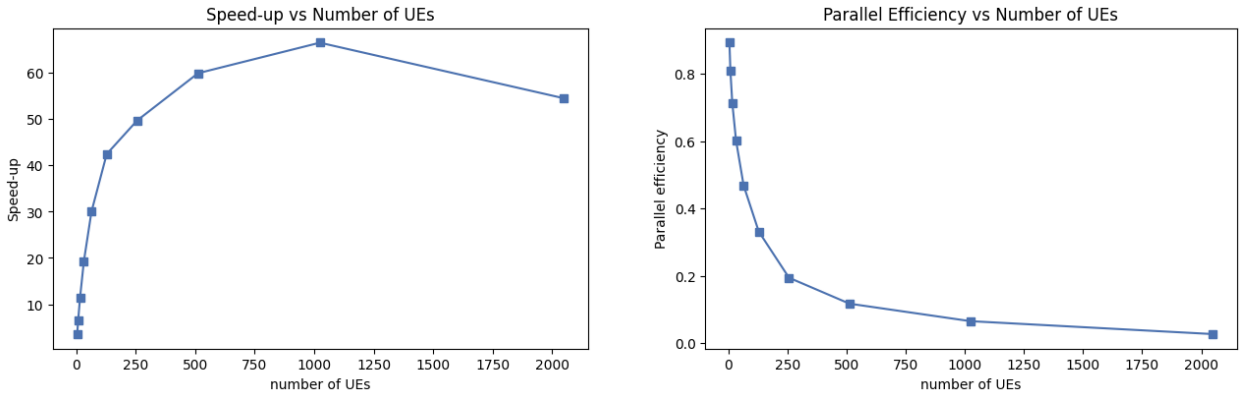

Figure 7: Speed-up and Parallel efficiency vs number of UEs

In Figure 8, we provide a program runtime comparison by increasing the number of UEs. Table 3 summarises the metric values from all plots we have.

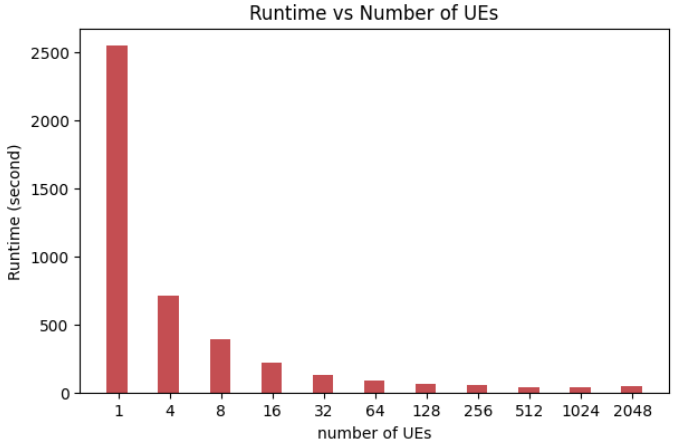

Figure 8: Runtime vs number of UEs

Num UEs	Speed-up	Efficiency	Runtime (seconds)
1	NA	NA	2548.86
4	3.57	0.89	713.54
8	6.48	0.81	392.97
16	11.43	0.71	222.97
32	19.26	0.60	132.32
64	29.99	0.46	84.98
128	42.43	0.33	60.07
256	49.68	0.19	51.30
512	59.88	0.11	42.56
1024	66.48	0.06	38.34
2048	54.49	0.02	46.77

Table 3: Summary of strong scaling statistics

5 Load Balance Profile

In this section, we provide a profiling of our parallel program to demonstrate the load balance and communication synchronisation properties. The program is executed using 32 UEs and maximum number of neutrons is set to 2×10^5. The left table in Figure 9 provides a break-down of percentage of time used of communication and the right graph of Figure 9 visualises the communications. As can be observed from both figures, the communication workload on each UE are roughly about the same, and the timing of communication is synchronised well across all UEs. This result is expected from how we design the communication scheme.

	value 0	value 10	value 19	value 20	value 31	value 32
THREAD 1.1.1	64.26 %	29.73 %	0.01 %	5.95 %	0.00 %	0.05 %
THREAD 1.2.1	63.38 %	30.58 %	0.01 %	5.95 %	0.03 %	0.05 %
THREAD 1.3.1	65.38 %	28.58 %	0.01 %	5.95 %	0.03 %	0.05 %
THREAD 1.4.1	64.50 %	29.46 %	0.01 %	5.95 %	0.03 %	0.05 %
THREAD 1.5.1	65.24 %	28.72 %	0.01 %	5.95 %	0.03 %	0.05 %
THREAD 1.6.1	65.38 %	28.58 %	0.01 %	5.95 %	0.03 %	0.05 %
THREAD 1.7.1	65.82 %	28.14 %	0.01 %	5.95 %	0.03 %	0.05 %
THREAD 1.8.1	65.77 %	28.19 %	0.01 %	5.95 %	0.03 %	0.05 %
THREAD 1.9.1	61.12 %	32.84 %	0.01 %	5.95 %	0.03 %	0.05 %
THREAD 1.10.1	62.15 %	31.81 %	0.01 %	5.95 %	0.03 %	0.05 %
THREAD 1.11.1	62.12 %	31.84 %	0.01 %	5.95 %	0.03 %	0.05 %
THREAD 1.12.1	62.77 %	31.19 %	0.01 %	5.95 %	0.03 %	0.05 %
THREAD 1.13.1	62.82 %	31.14 %	0.01 %	5.95 %	0.03 %	0.05 %
THREAD 1.14.1	62.51 %	31.45 %	0.01 %	5.95 %	0.03 %	0.05 %
THREAD 1.15.1	62.70 %	31.26 %	0.01 %	5.95 %	0.03 %	0.05 %
THREAD 1.16.1	63.13 %	30.83 %	0.01 %	5.95 %	0.03 %	0.05 %
THREAD 1.17.1	64.75 %	29.26 %	0.00 %	5.92 %	0.03 %	0.05 %
THREAD 1.18.1	65.49 %	28.51 %	0.00 %	5.92 %	0.03 %	0.05 %
THREAD 1.19.1	64.57 %	29.43 %	0.00 %	5.92 %	0.03 %	0.05 %
THREAD 1.20.1	64.14 %	29.86 %	0.00 %	5.92 %	0.03 %	0.05 %
THREAD 1.21.1	63.55 %	30.45 %	0.00 %	5.92 %	0.03 %	0.05 %
THREAD 1.22.1	63.57 %	30.43 %	0.00 %	5.92 %	0.03 %	0.05 %
THREAD 1.23.1	64.29 %	29.71 %	0.00 %	5.92 %	0.03 %	0.05 %
THREAD 1.24.1	64.33 %	29.67 %	0.00 %	5.92 %	0.03 %	0.05 %
THREAD 1.25.1	67.93 %	26.07 %	0.00 %	5.92 %	0.03 %	0.05 %
THREAD 1.26.1	66.58 %	27.43 %	0.00 %	5.92 %	0.03 %	0.05 %
THREAD 1.27.1	63.92 %	30.07 %	0.00 %	5.92 %	0.03 %	0.05 %
THREAD 1.28.1	66.27 %	27.73 %	0.00 %	5.92 %	0.03 %	0.05 %
THREAD 1.29.1	68.10 %	25.90 %	0.00 %	5.92 %	0.03 %	0.05 %
THREAD 1.30.1	67.09 %	26.92 %	0.00 %	5.92 %	0.03 %	0.05 %
THREAD 1.31.1	67.33 %	26.67 %	0.00 %	5.92 %	0.03 %	0.04 %
THREAD 1.32.1	68.08 %	25.92 %	0.00 %	5.92 %	0.03 %	0.05 %

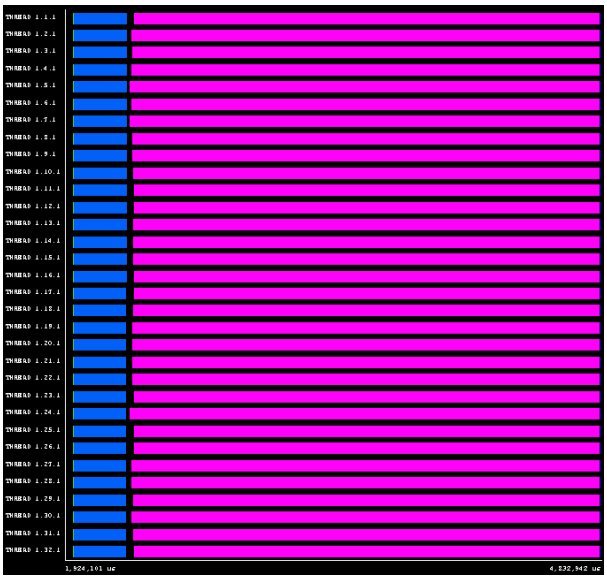

Figure 9: Profiling of the parallel solution