ECE445 - Parallel & Distributed Computing

– Homework 2 –

submitted by

Aggeliki-Ostralis Vliora, Spyridon Liaskonis, Stelios Alevizos

December 26, 2024

Aggeliki-Ostralis Vliora, Spyridon Liaskonis, Stelios Alevizos
avliora@uth.gr, sliaskonis@uth.gr, astelios@uth.gr
Student ID: 03140,03381, 02837

Contents

1 Exercise 1

1.1 OpenMP functions

The functions used for problem 1 are shown in Table 1

Table 1: OpenMP Runtime Library Functions

Function Name	Description
`omp_get_wtime()`	Returns the elapsed wall clock time.
`omp_get_thread_num()`	Returns the thread ID of the calling thread.
`omp_get_num_procs()`	Returns the number of processors available.
`omp_get_num_threads()`	Returns the number of threads currently in the team.
`omp_get_max_threads()`	Returns the maximum number of threads that can be used.
`omp_in_parallel()`	Indicates if the code is currently running in a parallel region.
`omp_get_dynamic()`	Returns whether dynamic adjustment of the number of threads is enabled.
`omp_get_max_active_levels()`	Returns the maximum number of parallelism levels allowed.

1.2 Execution Example

```
OpenMP version: 201511. Information provided by thread 7 .
Number of processors = 8
Number of threads = 8
Max threads = 8
In parallel? = 1
Dynamic threads enabled? = 0
Nested parallelism levels supported = 0
HELLO. I am the Master thread.  I created all participating threads.
I am thread 6 and worked for 3.947577 msec.
I am thread 5 and worked for 3.999088 msec.
I am thread 4 and worked for 3.950128 msec.
I am thread 2 and worked for 3.950129 msec.
I am thread 1 and worked for 3.947803 msec.
I am thread 3 and worked for 3.949916 msec.
I am thread 0 and worked for 0.026982 msec.
I am thread 7 and worked for 3.977772 msec.
```

Figure 1: Execution Example

2 Exercise 2

2.1 Task

We need to calculate a matrix multiplication $C = AB$
where $A(i,j) = i + j, B(i,j) = i * j$ and A has dimensions M*K and B K*N.

We have been tasked with writing two C programs solving this problem, one in serial form and one in parallel form, using OpenMP.

We aim to analyze the behavior of our parallel matrix multiplication program and investigate the impact of different collapse levels on the distribution of work among threads and the overall program performance.

2.2 A) Thread Distribution

Firstly we will examine the distribution of computational work among threads in this parallel algorithm.

2.2.1 Thread Distribution with Dynamic Scheduling

Collapse = 1: With collapse=1, OpenMP parallelizes only the outermost loop (i). Each iteration of the i loop processes all combinations of j and k serially, with a total of 5 iterations for the i loop. When chunk size is 2, the total number of chunks is 2 full chunks with 1 remaining chunk, which results in an uneven distribution. Thread 0 processes iterations 0 and 1, Thread 1 processes iterations 2 and 3, and Thread 2 handles the remaining iteration (i=4). With chunk size 3, there are 2 chunks, and Threads 0 and 1 each handle a chunk of 3 iterations, leaving Thread 2 and Thread 3 idle. Finally, with chunk size 4, Thread 0 handles most of the iterations (i=0 to i=3), while Thread 1 processes the remaining iteration (i=4), leading to an imbalance in thread utilization. Overall, when the chunk size increases, the imbalance in the workload across threads becomes more pronounced.

thread_id for all k(for each i,j)	j=0	j=1	j=2	j=3
i=0	222	222	222	222
i=1	222	222	222	222
i=2	111	111	111	111
i=3	111	111	111	111
i=4	333	333	333	333

Collapse Level 1 for chunks=2

thread_id for all k(for each i,j)	j=0	j=1	j=2	j=3
i=0	333	333	333	333
i=1	333	333	333	333
i=2	333	333	333	333
i=3	111	111	111	111
i=4	111	111	111	111

Collapse Level 1 for chunks=3

thread_id for all k(for each i,j)	j=0	j=1	j=2	j=3
i=0	222	222	222	222
i=1	222	222	222	222
i=2	222	222	222	222
i=3	222	222	222	222
i=4	111	111	111	111

Collapse Level 1 for chunks=4

Collapse = 2: With collapse=2, OpenMP combines the i and j loops into a single loop, resulting in a total of $M \times N = 5 \times 4 = 20$ iterations. This leads to a more balanced workload distribution compared to collapse=1. For chunk size 2, the total number of chunks is 10, and the work is evenly distributed across all 4 threads. Each thread picks up 2 iterations, and the threads are fully utilized. When using chunk size 3, the total number of chunks is 6 full chunks, plus 1 remaining chunk with 2 iterations. This distribution is slightly uneven, with some threads processing fewer iterations, but the overall workload is still shared fairly. With chunk size 4, there are 5 full chunks, and the threads pick up

iterations in a balanced manner, with each thread handling chunks of 4 iterations. This results in efficient thread utilization and evenly distributed work.

thread_id for all k(for each i,j)	j=0	j=1	j=2	j=3
i=0	222	222	111	111
i=1	333	333	000	000
i=2	333	333	000	000
i=3	222	222	333	333
i=4	111	111	111	111

Collapse Level 2 for chunks=2

thread_id for all k(for each i,j)	j=0	j=1	j=2	j=3
i=0	222	222	222	111
i=1	111	111	333	333
i=2	333	000	000	000
i=3	111	111	111	000
i=4	000	000	333	333

Collapse Level 2 for chunks=3

thread_id for all k(for each i,j)	j=0	j=1	j=2	j=3
i=0	111	111	111	111
i=1	222	222	222	222
i=2	333	333	333	333
i=3	333	333	333	333
i=4	222	222	222	222

Collapse Level 2 for chunks=4

Collapse = 3: With collapse=3, OpenMP combines the i, j, and k loops into a single iteration space, leading to a total of $M \times N \times K = 5 \times 4 \times 3 = 60$ iterations. With chunk size 2, the total number of chunks is 30, and the work is evenly balanced across threads. Threads pick up 2 iterations each, and the work is fairly distributed among the threads. For chunk size 3, there are 20 chunks in total, and the threads pick up chunks of 3 iterations each. This results in an evenly distributed workload, with all threads being utilized efficiently. With chunk size 4, the number of chunks reduces to 15, and each thread processes 4 iterations at a time. The threads share the work equally, with fewer chunks to handle, and all threads remain busy with no idle time.

thread_id for all k(for each i,j)	j=0	j=1	j=2	j=3
i=0	221	133	220	033
i=1	330	022	003	300
i=2	003	322	113	311
i=3	221	133	331	122
i=4	331	111	223	333

Collapse Level 3 for chunks=2

thread_id for all k(for each i,j)	j=0	j=1	j=2	j=3
i=0	111	222	333	333
i=1	111	111	000	000
i=2	222	111	333	111
i=3	222	111	000	222
i=4	000	222	222	222

Collapse Level 3 for chunks=3

thread_id for all k(for each i,j)	j=0	j=1	j=2	j=3
i=0	222	211	113	333
i=1	111	122	222	222
i=2	222	222	223	333
i=3	000	033	330	000
i=4	111	133	331	111

Collapse Level 3 for chunks=4

2.2.2 Thread Distribution with Static Scheduling

Collapse = 1: With collapse=1, OpenMP parallelizes only the outermost loop (i), resulting in a total of 5 iterations for the i loop. In static scheduling, the iterations are divided into equal chunks, and each thread processes one or more chunks of iterations. For chunk size 2, the total number of chunks is 2 full chunks with 1 remaining chunk. This leads to a fairly balanced distribution, but some threads may process fewer iterations. With chunk size 3, the total number of chunks is 2, and Threads 3, 1 handle chunks of 3, 2 iterations. However, Threads 0 and 2 remain idle as there are only 2 chunks. For chunk size 4, the total number of chunks is 2, with Thread 2 processing the first chunk (i=0 to i=3) and Thread 1 processing the remaining iteration (i=4). As chunk size increases, the workload becomes less balanced, and some threads end up with fewer iterations to process or staying idle.

thread_id for all k(for each i,j)	j=0	j=1	j=2	j=3
i=0	222	222	222	222
i=1	222	222	222	222
i=2	111	111	111	111
i=3	111	111	111	111
i=4	333	333	333	333

Collapse Level 1 for chunks=2

thread_id for all k(for each i,j)	j=0	j=1	j=2	j=3
i=0	333	333	333	333
i=1	333	333	333	333
i=2	333	333	333	333
i=3	111	111	111	111
i=4	111	111	111	111

Collapse Level 1 for chunks=3

thread_id for all k(for each i,j)	j=0	j=1	j=2	j=3
i=0	222	222	222	222
i=1	222	222	222	222
i=2	222	222	222	222
i=3	222	222	222	222
i=4	111	111	111	111

Collapse Level 1 for chunks=4

Collapse = 2: With collapse=2, OpenMP combines the i and j loops into a single iteration space, resulting in a total of $M \times N = 5 \times 4 = 20$ iterations. With chunk size 2, the total number of chunks is 10, and each thread is assigned 2 iterations. Threads 0, 1, 2, and 3 each process 2 iterations, leading to a well-balanced distribution. With chunk size 3, the total number of chunks is 6 full chunks, plus 1 remaining chunk with 2 iterations. Although the workload is distributed evenly, one thread handles fewer iterations. For chunk size 4, the total number of chunks is 5, with each thread handling 4 iterations. Threads are fully utilized, and work is distributed in an efficient manner, with no idle threads.

thread_id for all k(for each i,j)	j=0	j=1	j=2	j=3
i=0	000	000	111	111
i=1	222	222	333	333
i=2	000	000	111	111
i=3	222	222	333	333
i=4	000	000	111	111

Collapse Level 2 for chunks=2

thread_id for all k(for each i,j)	j=0	j=1	j=2	j=3
i=0	000	000	000	111
i=1	111	111	222	222
i=2	222	333	333	333
i=3	000	000	000	111
i=4	111	111	222	222

Collapse Level 2 for chunks=3

thread_id for all k(for each i,j)	j=0	j=1	j=2	j=3
i=0	000	000	000	000
i=1	111	111	111	111
i=2	222	222	222	222
i=3	333	333	333	333
i=4	000	000	000	000

Collapse Level 2 for chunks=4

Collapse = 3: With collapse=3, OpenMP combines all three loops into a single iteration space, leading to a total of $M \times N \times K = 5 \times 4 \times 3 = 60$ iterations. For chunk size 2, the total number of chunks is 30, and each thread picks up 2 iterations at a time. Threads 0, 1, 2, and 3 are all busy, with each thread processing a chunk of 2 iterations. The workload is fairly balanced across the threads. For chunk size 3, the total number of chunks is 20, and each thread processes a chunk of 3 iterations. All threads are utilized efficiently and that the workload is evenly distributed. With chunk size 4, the total number of chunks is 15, and each thread processes 4 iterations at a time. Threads handle 4 iterations each, leading to a well-balanced workload distribution with minimal idle time.

thread_id for all k(for each i,j)	j=0	j=1	j=2	j=3
i=0	001	122	330	011
i=1	223	300	112	233
i=2	001	122	330	011
i=3	223	300	112	233
i=4	001	122	330	011

Collapse Level 3 for chunks=2

thread_id for all k(for each i,j)	j=0	j=1	j=2	j=3
i=0	000	111	222	333
i=1	000	111	222	333
i=2	000	111	222	333
i=3	000	111	222	333
i=4	000	111	222	333

Collapse Level 3 for chunks=3

thread_id for all k(for each i,j)	j=0	j=1	j=2	j=3
i=0	000	011	112	222
i=1	333	300	001	111
i=2	222	233	330	000
i=3	111	122	223	333
i=4	000	011	112	222

Collapse Level 3 for chunks=4

2.3 B) Observations

With dimensions M = 50, K = 10000, N = 20, we observe the following for each collapse level:

- **Level 1** With only 50 iterations to distribute across threads, the results reveal that smaller chunk sizes (e.g., chunk = 10) offer the most balanced workload distribution, as reflected in higher efficiency and speedup values. As the chunk size increases to 20 and 40, inefficiencies emerge due to load imbalance: threads finish their assigned chunks at different times, leaving some idle. This is evident in the significant drop in efficiency (Ep) and the slower improvements in speedup (Sp) at higher thread counts. For example, at chunk = 40 and 8 threads, Ep plummets to 0.234958. Additionally, after 5 threads, adding more threads introduces overhead without meaningful gains in execution time, as Tp increases instead of decreasing. This behavior highlights the importance of choosing an appropriate chunk size to ensure even workload distribution, particularly when the total iteration count is small.

Nthreads \ chunk	10	20	40
2	0.029198, 1.482944, 0.741472	0.028098, 1.540999, 0.513666	0.036887, 1.173828, 0.293457
3	0.02033, 2.129808, 1.064904	0.021094, 2.052669, 0.684223	0.037746, 1.147115, 0.286779
4	0.020469, 2.115345, 1.057673	0.020603, 2.101587, 0.700529	0.035974, 1.203619, 0.300905
5	0.019007, 2.278055, 1.139028	0.019886, 2.177361, 0.725787	0.034782, 1.244868, 0.311217
6	0.020459, 2.116379, 1.058190	0.031361, 1.380664, 0.460221	0.039526, 1.095456, 0.273864
7	0.019145, 2.261635, 1.130817	0.02822, 1.534337, 0.511446	0.042356, 1.022264, 0.255566
8	0.031643, 1.368360, 0.684180	0.032903, 1.315959, 0.438653	0.046071, 0.939832, 0.234958

Statistics (Tp, Sp, Ep) for Collapse Level = 1

- **Level 2** In this case, with M*N = 1000 iterations, we observe a general decline in execution time as the number of threads increases. However, there are discrepancies between chunk sizes (10, 20, 40) in execution time, speedup (Sp), and efficiency (Ep). For smaller chunk sizes (e.g., 10), the execution time is slightly higher due to increased scheduling overhead, but the workload distribution is more even, leading to better efficiency, particularly with lower thread counts. As the chunk size increases (e.g., 20 and 40), we see improved execution times for mid-range thread counts, but efficiency drops at higher thread numbers because larger chunks result in uneven workload distribution, leaving some threads idle. Notably, at chunk size 20, the parallel time (Tp) and speedup (Sp) values are more consistent across threads, suggesting a balance between overhead and workload distribution. For chunk size 40, while Tp remains competitive, efficiency decreases significantly due to the limited number of iterations not being evenly divisible among the threads.

- **Level 3** With M*N*K = 10^7 iterations (a significant number of workloads), we observe a clear negative trend in execution times as we increase the thread count

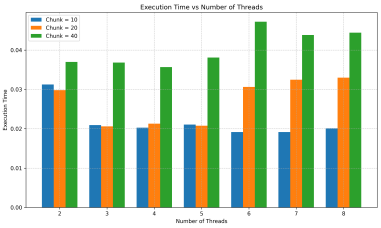
Nthreads \ chunk	10	20	40
2	0.03209, 1.436647, 0.718323	0.026996, 1.707734, 0.569245	0.027253, 1.691630, 0.422908
3	0.027896, 1.652638, 0.826319	0.019241, 2.396029, 0.798676	0.020662, 2.231246, 0.557811
4	0.024638, 1.871175, 0.935587	0.015837, 2.911031, 0.970344	0.015794, 2.918957, 0.729739
5	0.019854, 2.322051, 1.161025	0.014742, 3.127255, 1.042418	0.015139, 3.045247, 0.761312
6	0.019266, 2.392920, 1.196460	0.015132, 3.046656, 1.015552	0.016361, 2.817798, 0.704450
7	0.019583, 2.354185, 1.177092	0.01637, 2.816249, 0.938750	0.014558, 3.166781, 0.791695
8	0.032008, 1.440327, 0.720164	0.016073, 2.868288, 0.956096	0.017092, 2.697285, 0.674321

Statistics (Tp, Sp, Ep) for Collapse Level = 2

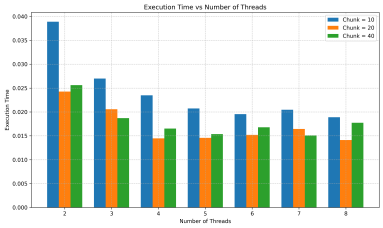
and chunk size. This is because when we parallelize the third loop as well, we need to add an atomic add operation to the instruction executed within the nested loops. This is required to prevent multiple threads from accessing and modifying the same memory location simultaneously. However, the atomic add introduces overhead, which reduces the overall performance (Ep) and speedup (Sp), as observed in the table below, and increases the total execution time, as shown in the following diagrams.

Nthreads \ chunk	10	20	40
2	0.177648, 0.277813, 0.138907	0.19573, 0.252148, 0.084049	0.189829, 0.259987, 0.064997
3	0.204906, 0.240857, 0.120428	0.213019, 0.231684, 0.077228	0.221856, 0.222455, 0.055614
4	0.204392, 0.241462, 0.120731	0.212898, 0.231815, 0.077272	0.211131, 0.233755, 0.058439
5	0.207004, 0.238416, 0.119208	0.202584, 0.243617, 0.081206	0.201071, 0.245451, 0.061363
6	0.196379, 0.251315, 0.125658	0.200268, 0.246435, 0.082145	0.190447, 0.259143, 0.064786
7	0.199456, 0.247438, 0.123719	0.198161, 0.249055, 0.083018	0.196567, 0.251075, 0.062769
8	0.195983, 0.251823, 0.125911	0.19516, 0.252885, 0.084295	0.2074, 0.237960, 0.059490

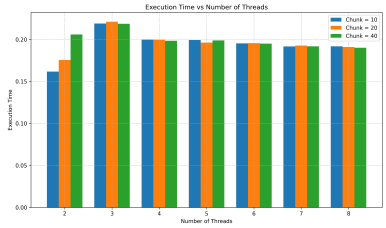
Statistics (Tp, Sp, Ep) for Collapse Level = 3



Execution Time VS Number of Threads, Collapse Level = 1



Execution Time VS Number of Threads, Collapse Level = 2



Execution Time VS Number of Threads, Collapse Level = 3

3 Exercise 3

3.1 Task

Implement the Jacobi Method for determining the solutions of a strictly diagonally dominant system of linear equations using OpenMP (form $Ax = b$). The solution needs to be implemented in a separate function containing orphaned OpenMP directives inside a seperate file (jacobi_par.c).

3.2 Formula and Pseudocode

The Jacobi iterative method, in which we calculate a new estimation for the solution x, of system $Ax = b$ has a formula for each iteration as shown in 1. The pseudocode ?? (next page) shows our rough implementation of the method.

$$x_i^{(k+1)} = \frac{b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}}{a_{ii}}, \; i = 1, 2, ..., n \tag{1}$$

3.3 OpenMP Implementation

As instructed we have implemented the Jacobi method in a separate routine (jacobi_par.c), containing orphaned directives. This routine is called by our main program (ask3.c) in its parallel region.

```
#pragma omp parallel num_threads(N_THREADS) default(shared)
{
    iter = jacobi(A, b, N, maxIter, tol, x);
}
```

Inside the **jacobi** function (in jacobi_par.c), we have multiple 2-deep nested loops first for calculating the solution, then the residual. They look like this:

```
#pragma omp for private(sum)
    for(int i=0; i<N; i++) {
        sum = A[i][i]*x[i];

        for(int j=0; j<N; j++) {
            sum -= A[i][j]*x[j];
        }
        x_k[i] = (b[i] + sum)/A[i][i];
    }
```

In the end of the iteration, a single directive prints the values needed, all threads increment their iteration variables seperately and last they all meet in a barrier to synchronize before the next iteration:

```
#pragma omp single
    printf("iter = %d, residual = %.6f, difference %.6f\n", iter,
           maxnorm_res, maxnorm_calc);

iter++;

#pragma omp barrier
if(tol > maxnorm_res)
    break;
```

Algorithm 1 Jacobi iterative method

Input: 2D array A of size $n \times n$, vectors b and x of size n, max number of iterations $maxIter$, error tolerance tol
Output: Iterations reached $iter$, estimated solution x
$iter \leftarrow 1$
while $iter \leq maxIter$ **do**
    $x_{k+1}[n] \leftarrow \{0, ..., 0\}$ ▷ Calculate x_{k+1}
    for each row index i from 0 to $n-1$ **do**
        $sum \leftarrow 0$
        for each column index j from 0 to $i-1$ and $i+1$ to $n-1$ **do**
            $sum \leftarrow sum - A[i][j] * x[j]$
        end for
        $x_{k+1}[i] \leftarrow (b[i] + sum)/A[i][i]$
    end for
    $res[n] \leftarrow \{0, ..., 0\}$ ▷ Calculate residual $b - Ax_{k+1}$
    for each row index i from 0 to $n-1$ **do**
        $res[i] \leftarrow b[i]$
        for each column index j from 0 to $n-1$ **do**
            $res[i] \leftarrow res[i] - A[i][j] * x_{k+1}[j]$
        end for
    end for
    $norm \leftarrow maxnorm(x_{k+1})$ ▷ Calculate max norm and check against tol
    if tol >norm **then**
        break
    end if
    $x \leftarrow x_{k+1}$
    $iter \leftarrow iter + 1$
end while
Return $iter, x$

3.4 Testing Results

The system used for testing is the following:

$$\begin{bmatrix} 2 & -1 & & \\ -1 & \ddots & \ddots & \\ & \ddots & \ddots & -1 \\ & & -1 & 2 \end{bmatrix} x = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ N+1 \end{bmatrix}$$

Testing of the application occured with parameters:

- **N** = 10000
- **maxIter** = 200
- **tol** = 0.00005

And the results, as presented below, show that in **simple** parallelism, the number of threads correlates positively to a decrease in execution time. In the case of **nested** parallelism however, the # of threads increase initially improves the time, but afterwards worsens it significantly. This happens because the **nested** parallelism adds significant overhead to the execution. In every iteration a team of threads is created specifically to compute the new solution and is subsequently destroyed, only to be created again in the next iteration.

# of threads	Simple	Nested
1	38.37	38.37
2	19.55	20.08
4	10.21	20.51
5	9.57	22.35
8	8.01	31.04
10	8.31	39.16
12	8.00	49.15
16	7.67	72.68
20	7.52	93.88
32	7.48	167.98

Table 2: Execution Times (sec) for: N = 10000, maxIter = 200, tol = 0.00005

A System Configuration

This section includes all the details about the hardware and software configuration used to run the experiments.

A.1 Experiment 1 - Exercise 2

A.1.1 Hardware Details

The system used for running the experiments is described below:

- Processor: Intel® Core™ i7-8550U × 8
- RAM: 8 GB

The CPU topology is shown in figure 2:

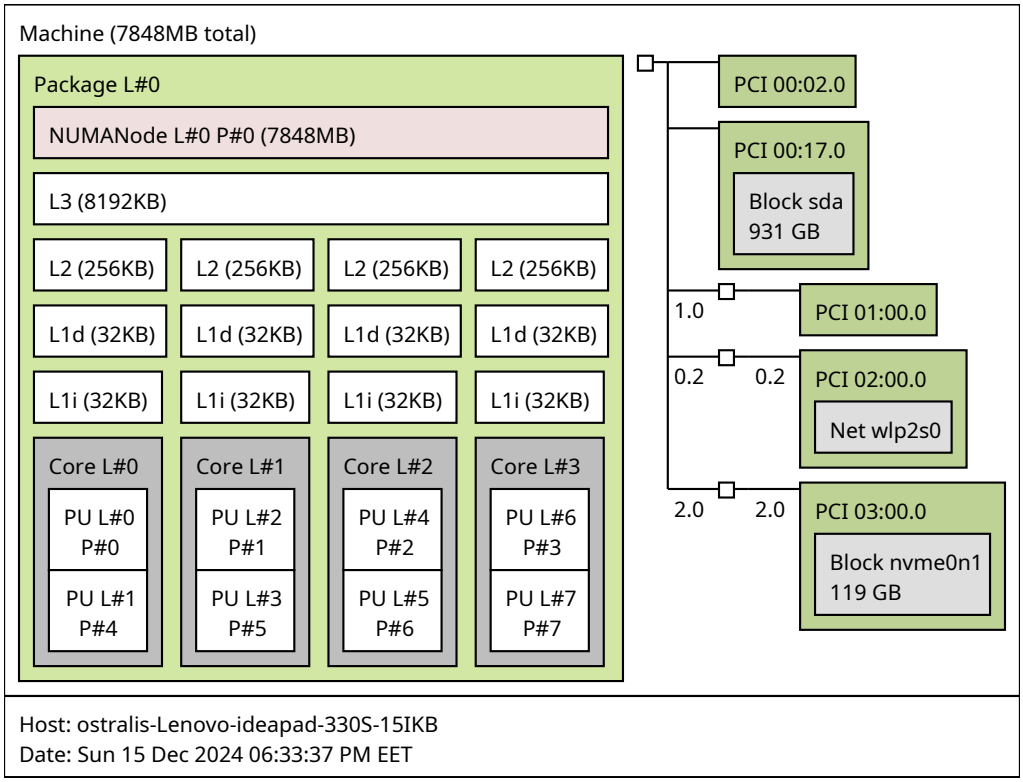


Figure 2: ALenovo Lenovo ideapad 330S-15IKB Topological Map

A.1.2 Software Details

The software environment is outlined below:

- Operating System: Ubuntu 24.04.1 LTS
- Compiler: gcc version 13.2.0 (Ubuntu 13.2.0-23ubuntu4)

A.2 Experiment 2 - Exercise 3

A.2.1 Hardware Details

The system used for running the experiments is described below:

- Processor: Apple M1, 8 cores, 3.20 Ghz
- RAM: 8 GB LPDDR4

The CPU topology is shown in figure 3:

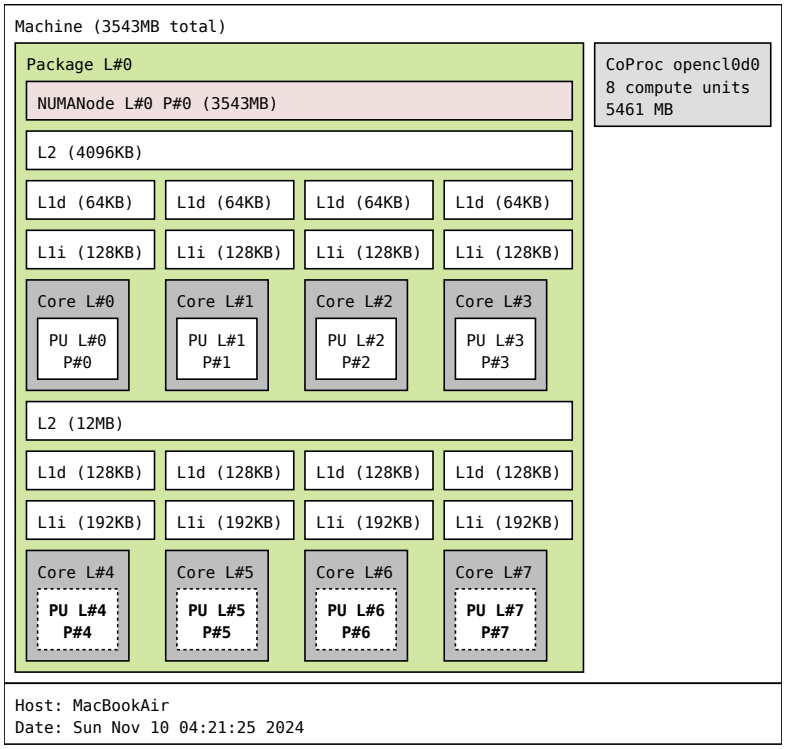


Figure 3: Apple M1 Topological Map

A.2.2 Software Details

The software environment is outlined below:

- Operating System: MacOS Sequoia 15.1 (24B83)
- Compiler: Apple clang version 12.0.5 (clang-1205.0.22.9)