Multi-threading: Matrix Multiplication

OBJECTIVES

- Learn the basics of multi-threading implementation using *pthreads* in C.
- Execute multi-threaded applications.
- Measure execution time (with 1 us resolution) for different number of threads.

REPOSITORY EXAMPLES

- Refer to the Tutorial: Embedded Intel for the source file used in this Tutorial.

TERASIC DE2I-150 BOARD

- Refer to the board website or the Tutorial: Embedded Intel for User Manuals and Guides.

BOARD SETUP

- Connect the monitor (VGA or HDMI) as well as the keyboard and mouse.
  - ✓ Refer to the *DE2i-150 Quick Start Guide* (page 2) for a useful illustration.

ACTIVITIES

FIRST ACTIVITY: MATRIX MULTIPLICATION (SEQUENTIAL IMPLEMENTATION)

- This is a popular application. It involves many calculations, especially when it comes to matrices of higher order:

$$C_{m\times n} = A_{m\times p} \times B_{p\times n}$$

- A straightforward implementation involves nested loops where the dot product is computed for each element of the output matrix. There are mxn dot products in this operation, where each dot product involves two p-element vectors.

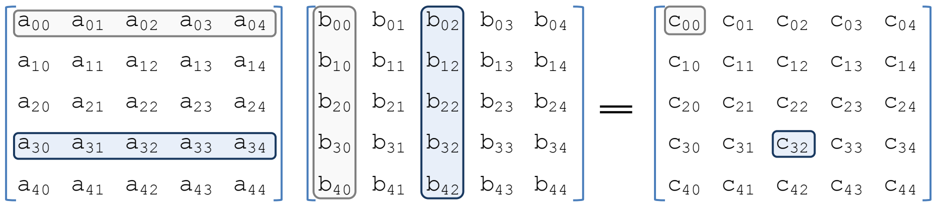


Figure 1. Matrix multiplication for matrices of size 5x5

- Application files: `matrix_mult.c, mat_fun.c, mat_fun.h`
  - ✓ We use a parameter $N = m = n = p = 20$. This parameter can be modified before compilation.
  - ✓ The matrices A and B are initialized with random data.

- Compile this application:

```
make matrix_mult ↵
```

- Execute this application:

```
./matrix_mult ↵
```

- **Processing time**: This is measured using the `gettimeofday()` function in the `sys/time.h` library. Note that only the actual computation time is included. This excludes memory allocation and matrix initialization.
  - ✓ For N=20, the processing time results in 90 us. Feel free to verify this result.

SECOND ACTIVITY: MULTI-THREADED MATRIX MULTIPLICATION

- Using *pthreads* leverages the parallel computing capabilities of the microprocessor. Here, parallelism is achieved by distributing the operations (ideally evenly) among threads than run in parallel.
- This popular application can be easily parallelized.

$$C_{m\times n} = A_{m\times p} \times B_{p\times n}$$

- In this application, we distribute the operations in terms of computed rows.

OPERATION

- To compute an output matrix with n rows, a group of threads is generated, where each thread computes a number of rows. All these threads simultaneously compute the output rows.

- If the number of threads is given by $nthreads$, then the index i represents each thread form 0 to $nthreads\text{-}1$. Thus, each thread i computes rows from: $\left\lfloor \frac{i \times n}{nthreads} \right\rfloor$ to $\left\lfloor \frac{(i+1) \times n}{nthreads} \right\rfloor - 1$.
  - ✓ $nthreads = n$: Each thread i computes row i.
  - ✓ $nthreads = 1$: Here, thread $i = 0$ computes rows from 0 to $n\text{-}1$. There is no parallelization here.
  - ✓ $nthreads > n$: Here, there are $nthreads\text{-}n$ threads where $\left\lfloor \frac{i \times n}{nthreads} \right\rfloor = \left\lfloor \frac{(i+1) \times n}{nthreads} \right\rfloor$. These threads are idle. The remaining n threads compute one thread each. This is suboptimal.
  - ✓ In general, we prefer $nthreads \in [1, n]$

- Examples for $n = 10$.
  - ✓ Fig. 2 depicts the case for 5 threads, each in charge of computing two rows.
  - ✓ Table I shows two cases ($nthreads = 4,6$) where the computations cannot be distributed evenly among the threads.

TABLE I. ROWS COMPUTED PER EACH THREAD FOR DIFFERENT NUMBER OF THREADS. OUTPUT MATRIX IS OF SIZE 10X10.

$nthreads = 5$

i	rows computed
0	0 to 1
1	2 to 3
2	4 to 5
3	6 to 7
4	8 to 9

$nthreads = 6$

i	rows computed
0	0 to 0
1	1 to 2
2	3 to 4
3	5 to 5
4	6 to 7
5	8 to 9

$nthreads = 4$

i	rows computed
0	0 to 1
1	2 to 4
2	5 to 6
3	7 to 9

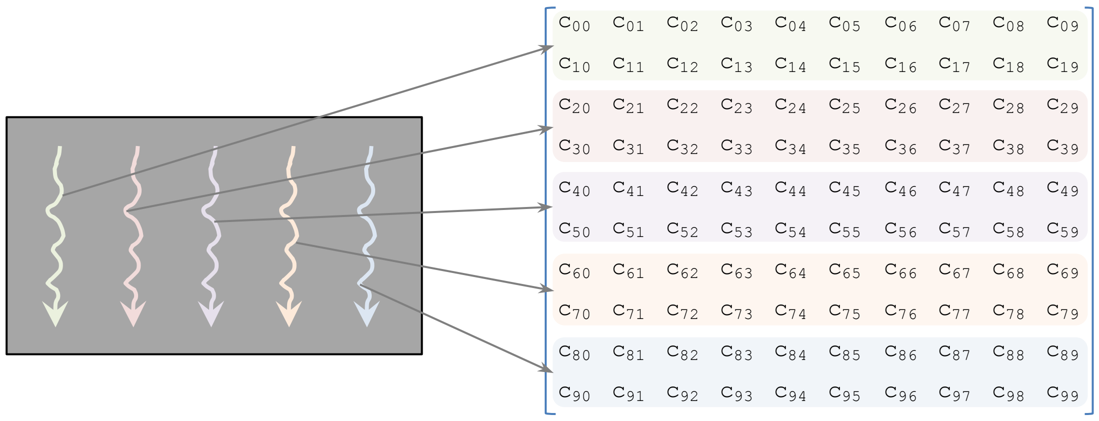
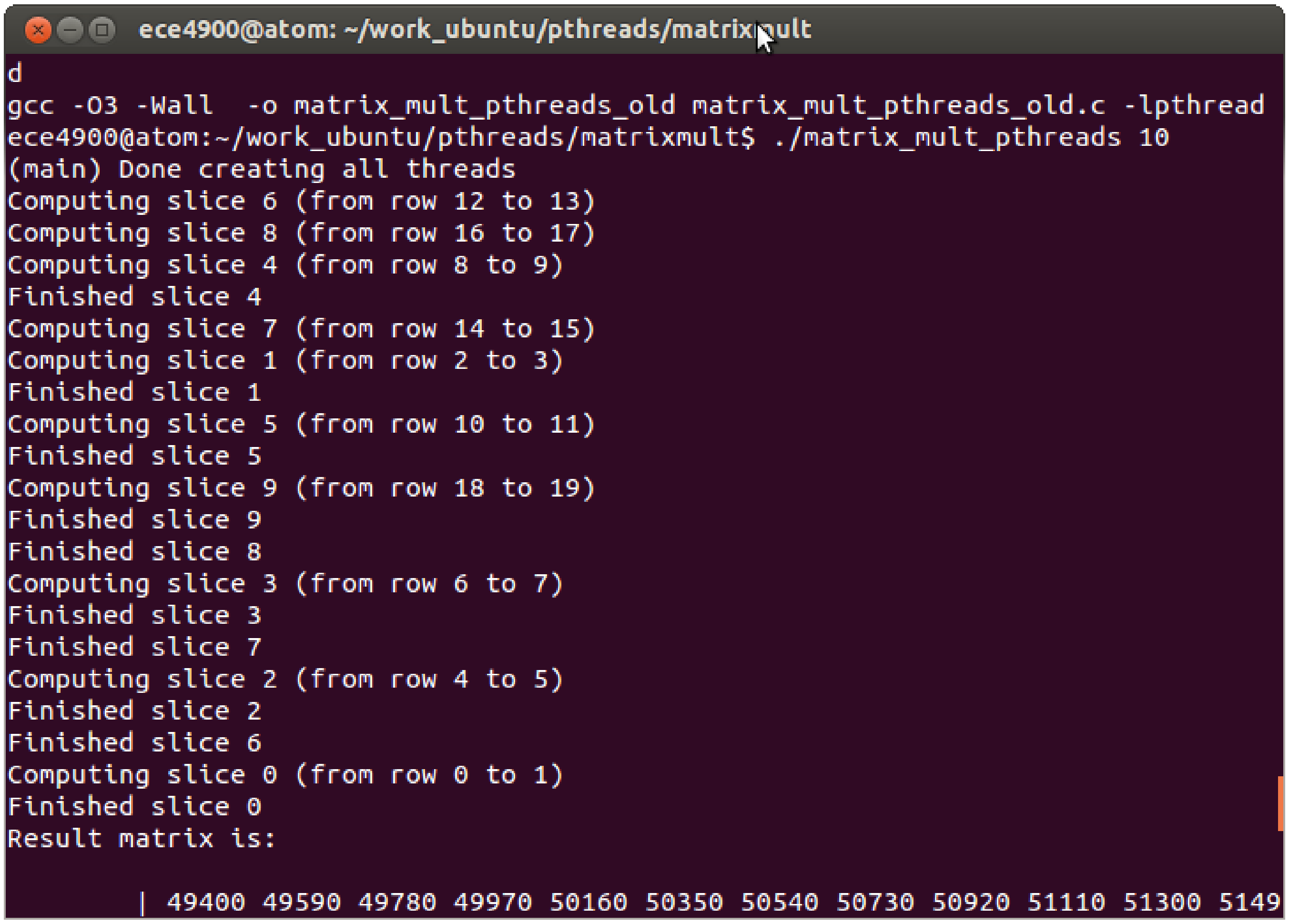

c00 c01 c02 c03 c04 c05 c06 c07 c08 c09
c10 c11 c12 c13 c14 c15 c16 c17 c18 c19
c20 c21 c22 c23 c24 c25 c26 c27 c28 c29
c30 c31 c32 c33 c34 c35 c36 c37 c38 c39
c40 c41 c42 c43 c44 c45 c46 c47 c48 c49
c50 c51 c52 c53 c54 c55 c56 c57 c58 c59
c60 c61 c62 c63 c64 c65 c66 c67 c68 c69
c70 c71 c72 c73 c74 c75 c76 c77 c78 c79
c80 c81 c82 c83 c84 c85 c86 c87 c88 c89
c90 c91 c92 c93 c94 c95 c96 c97 c98 c99

Figure 2. Task allocation for n=10, nthreads = 5.

- Application files: `matrix_mult_pthreads.c, mat_fun.c, mat_fun.h`
  - ✓ We use a parameter $\mathrm{N} = m = n = p = 20$. This parameter can be modified before compilation.
  - ✓ The matrices A and B are initialized with random data.
  - ✓ The number of threads ($nthreads$) is an argument of the program.
  - ✓ There is a parameter MTIME set to 0. It will print out execution messages for each thread. When measuring time, you need to set MTIME to 1 to avoid including `printf` instructions in the computation times.
  - ✓ Code structure:
    - ▫ Thread generation and initialization of arguments.
    - ▫ Initialization of input matrices A and B.
    - ▫ If $nthreads > 1$: Create $nthreads$ threads, where each thread i computes threads $\left\lfloor \frac{i \times n}{nthreads} \right\rfloor$ to $\left\lfloor \frac{(i+1) \times n}{nthreads} \right\rfloor - 1$.
    - ▫ Wait until threads complete, merge all the results.
    - ▫ Display output results.

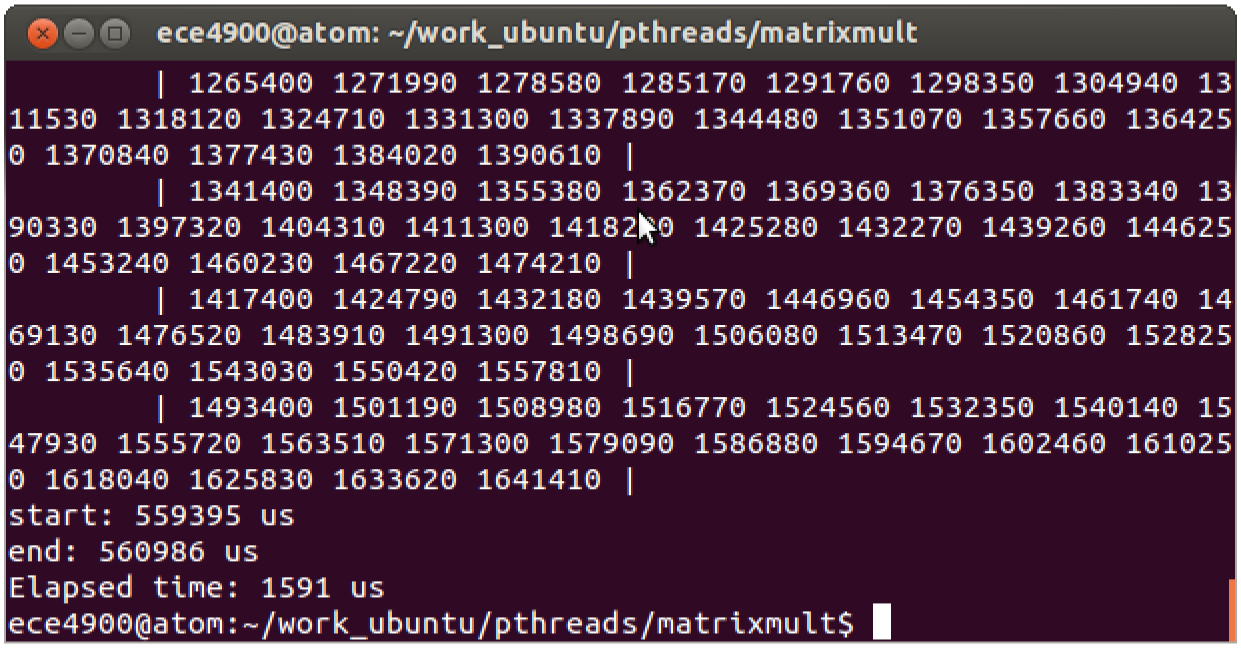
- Compile this application: `make matrix_mult_pthreads` ↵
- Execute this application: `./matrix_mult_pthreads <# of threads>` ↵
- Example: `./matrix_mult_pthreads 10`
  - ✓ Fig. 3 displays the execution messages (output matrix of size 20x20). Note that threads are not necessarily created consecutively. Some threads even finish before others are created (this varies every time the code is run), which means that some are not executed simultaneously. This happens because the processing load of each thread is small.
  - ✓ Fig. 4 displays the execution time. For best accuracy, the messages generated by the threads ("computing slice i", "finishing slice i") are not included in the measurements.



```
d
gcc -O3 -Wall  -o matrix_mult_pthreads_old matrix_mult_pthreads_old.c -lpthread
ece4900@atom:~/work_ubuntu/pthreads/matrixmult$ ./matrix_mult_pthreads 10
(main) Done creating all threads
Computing slice 6 (from row 12 to 13)
Computing slice 8 (from row 16 to 17)
Computing slice 4 (from row 8 to 9)
Finished slice 4
Computing slice 7 (from row 14 to 15)
Computing slice 1 (from row 2 to 3)
Finished slice 1
Computing slice 5 (from row 10 to 11)
Finished slice 5
Computing slice 9 (from row 18 to 19)
Finished slice 9
Finished slice 8
Computing slice 3 (from row 6 to 7)
Finished slice 3
Finished slice 7
Computing slice 2 (from row 4 to 5)
Finished slice 2
Finished slice 6
Computing slice 0 (from row 0 to 1)
Finished slice 0
Result matrix is:

       | 49400 49590 49780 49970 50160 50350 50540 50730 50920 51110 51300 5149
```

Figure 3. Program execution with nthreads = 10 and matrix size 20x20. Each thread generates message ("computing slice i", "Finished slice i"). We do not show the execution time here, as it would include the printing of the messages by each thread. Note how threads can be created and executed in a non-consecutive fashion



```
       | 1265400 1271990 1278580 1285170 1291760 1298350 1304940 13
11530 1318120 1324710 1331300 1337890 1344480 1351070 1357660 136425
0 1370840 1377430 1384020 1390610 |
       | 1341400 1348390 1355380 1362370 1369360 1376350 1383340 13
90330 1397320 1404310 1411300 1418290 1425280 1432270 1439260 144625
0 1453240 1460230 1467220 1474210 |
       | 1417400 1424790 1432180 1439570 1446960 1454350 1461740 14
69130 1476520 1483910 1491300 1498690 1506080 1513470 1520860 152825
0 1535640 1543030 1550420 1557810 |
       | 1493400 1501190 1508980 1516770 1524560 1532350 1540140 15
47930 1555720 1563510 1571300 1579090 1586880 1594670 1602460 161025
0 1618040 1625830 1633620 1641410 |
start: 559395 us
end: 560986 us
Elapsed time: 1591 us
ece4900@atom:~/work_ubuntu/pthreads/matrixmult$
```

Figure 4. Program execution with nthreads = 10 and matrix size 20x20. The execution time here does not include the printing of the messages by the individual threads.

- **Processing time**: This is measured using the `gettimeofday()` function in the `sys/time.h` library. Note that only the actual computation time is included. This excludes memory allocation, matrix initialization, and threads arguments generation.
  - ✓ Table II shows different cases. For N=20, the use of more threads increases the computation time. This also occurs for N=50, though the increase is not as pronounced (in some instances, the computation time is decreased). However, for N=100, 200, the multi-threading approach does reduce the computation time. Both thread initialization and argument generation incur in overhead time, and large values of N are needed to offset these overhead times.
  - ✓ Table III shows more cases for large matrices. Note that increasing the number of threads does not lead to performance improvements, and actually increases the processing time.
  - ✓ The case *nthreads=1* does not use *pthreads* (it is a normal matrix multiplication).
  - ✓ Due to variation (sometimes large) in processing time per run, we take an average over 10 consecutive runs. Feel free to verify these processing times on your own.

TABLE II. COMPUTATION TIMES (US) FOR DIFFERENT # OF THREADS AND DIFFERENT MATRIX SIZES. TERASIC DE2I-150 BOARD.

N	# of threads									
	1	2	3	4	5	6	7	8	9	10
20	90	508	586	647	781	946	982	1252	1304	1591
50	1361	1351	1277	1166	1533	1440	1641	1631	1717	1711
100	12331	6454	7152	5882	6841	4996	6691	6507	5152	4830
200	144835	92508	75932	74977	75448	77430	76338	80051	79652	72048
500	4,484270	2,264484	1,670049	1,639815	1,614753	1,565853	1,534638	1,525751	1,505418	1,487816

TABLE III. COMPUTATION TIMES (US) FOR DIFFERENT # OF THREADS AND LARGE MATRIX SIZES. TERASIC DE2I-150 BOARD.

N	# of threads			
	10	20	50	100
100	4830	5217	10094	14450
200	72048	73341	75921	77345
500	1,487816	1,475492	1,465813	1,482250

THIRD ACTIVITY: MULTI-THREADED FIBONACCI NUMBER COMPUTATION

- Compute F_n for $n > 1$:

$$F_n = F_{n-1} + F_{n-2}, F_0 = 0, F_1 = 1$$

STRATEGY

- This operation appears intrinsically sequential, however, note that it is possible to compute F_{n-1} and F_{n-2} simultaneously and without interference. Thus, two threads are used.
- Note that here we only attempt to compute F_n, not the entire sequence.
- Fig. 5 depicts the computation of F_5. Essentially, the computations of F_4 and F_3 can be performed simultaneously.
  - ✓ Note how some elements need to be computed several times, e.g.: F_2 is computed thrice, F_3 is computed twice. It looks as if: i) we could wait until the first F_2 is computed and re-use it in further computations, and ii) we could wait until the first F_3 is computed to use in the final summation. But this would essentially be a sequential implementation as we would need to find a way to signal when each element is available for further computations.

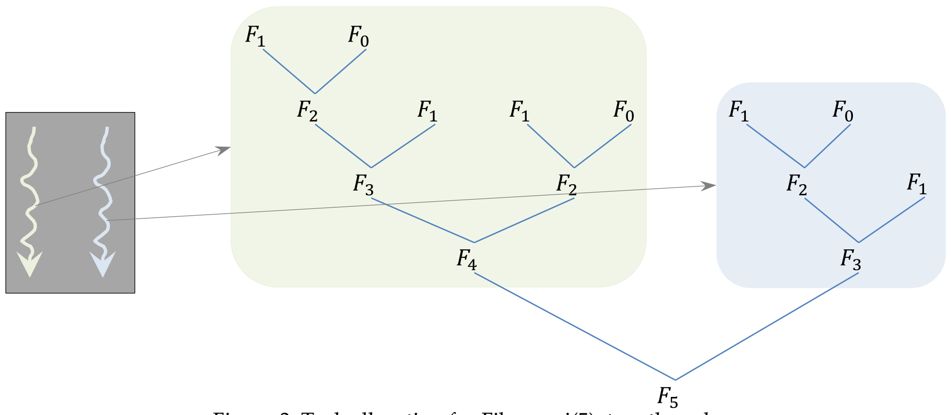


Figure 3. Task allocation for Fibonacci(5): two threads

- Application files: `fibonum.c, fibonum_fun.c, fibonum_fun.h, Makefile`
  - ✓ Code structure:
    - ▫ Initialization of arguments.

- Generate thread that computes F_{n-1}
- Have `main()` compute F_{n-2}.
- Wait until generated thread completes.
- Add the results of F_{n-2} and F_{n-1}.
- Display output result.

✓ We use `size_t` for the computation of F_n. This allows us to use the largest range of unsigned integers for our results.

- Compile this application: `make all` ↵
- Execute this application: `./fibonum <n: # Fib number n> <t: serial or parallel>` ↵
  - ✓ The code allows for parallel implementation (`t=2`) and serial implementation (`t=1`, this is for comparison purposes).
- Example: `./fibonum 30 2` (compute F_{30} using a parallel implementation)

- **Processing time:** By allowing simultaneous execution, we can save processing time as shown in the Table IV for different values of n.
  - ✓ Table IV shows different cases. Fibonnaci code gets about 1.5 speedup for computing F_n (for relatively large n)

TABLE IV. COMPUTATION TIMES (US) FOR DIFFERENT FIBONACCI NUMBERS.

n	Implementation	
	Sequential	Parallel (2 threads)
10	4	268
20	128	450
25	1330	1077
30	11,222	6,653
35	96,989	69,225
40	986,713	634,446
45	11,289,350	7,023,567
50	124,114,729	76,200165