

1 Introduction

In this report, I document my findings with exploring four more versions of the `matmul` program: one with 2 processes, one with N processes, one with N threads, and one using OpenMP. I describe my implementation for each of them before finally showing some plots that compare their runtime across the number of cores used. The code and artefacts are available in the associated folder.

2 Implementation

2.1 2 Processes matmul

Firstly, we improve upon the naive sequential implementation of `matmul`. Matrix multiplication as implemented in `mm_seq` is inherently parallelizable since accessing the two matrices to perform the multiplication does not change the matrices, so there won't be a race condition. Thus, we can divide the work in half by invoking a new process and having it do one-half of the total multiplications. The only difficulty is handling shared memory - we first construct a “shared” matrix to store the result in. Then, when we are done, we copy the matrix into our result matrix C , taking care to unmap the memory allocated to the “shared” matrix.

As seen in `mm_proc.c`,

```

1 void mm ( size_t N, NUMTYPE * A, NUMTYPE * B, NUMTYPE * C) {
2     int* T = mmap(NULL, N * N * sizeof (NUMTYPE),
3         PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS,
4         -1, 0);
5     int child_pid = fork();
6     if (child_pid == 0) {
7         mm_helper(0, N/2, N, A, B, T);
8         exit(0);
9     } else {
10        mm_helper(N/2, N, N, A, B, T);
11    }
12    wait(NULL);
13    memcpy(C, T, sizeof(NUMTYPE) * N * N);
14    munmap(T, N * N * sizeof (NUMTYPE));
15 }
```

where `mm_helper` is simply the initial sequential implementation of matrix multiplication abstracted as a helper function.

2.2 N-Processes matmul

When generalizing the above to N -processes, the intuition is the same: allocate memory for a “shared” matrix, perform the multiplication with N processes, then copy it over to the result matrix. However, an important thing to note is that the dimensions of the $n \times n$ matrix may not be perfectly divisible by N . For instance, if we have a 500×500 matrix, with 16 processes, 500 is not perfectly divisible by 16. So we have to account for this by finding the remainder with the modulo infix operator, and then handling that case specially. Other than that, the implementation is similar: in `mm_procn.c`, we have

```

1 void mm ( size_t N, NUMTYPE * A, NUMTYPE * B, NUMTYPE * C) {
2     int* T = mmap(NULL, N * N * sizeof (NUMTYPE),
3         PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS,
4         -1, 0);
5     int i = 0;
6     int slice;
```

```

7  int leftover = N % NUM_PROCESSES;
8  if (leftover == 0) {
9      slice = N/NUM_PROCESSES;
10     while (i < NUM_PROCESSES){
11         int child_pid = fork();
12         if (child_pid == 0){
13             mm_helper(i*slice, (i+1)*slice, N, A,B,T);
14             exit(0);
15         }
16         i++;
17     }
18     while(wait(NULL)>0);
19     memcpy(C, T, sizeof(NUMTYPE) * N * N);
20     munmap(T, N * N * sizeof (NUMTYPE));
21 } else { //account for when N cannot be perfectly divided by NUM_PROCESSES
22     slice = (N-leftover)/NUM_PROCESSES;
23     while (i< NUM_PROCESSES ){
24         int child_pid = fork();
25         if (child_pid == 0){
26             mm_helper(i*slice, (i+1)*slice, N, A,B,T);
27             exit(0);
28         }
29         i++;
30     }
31     while(wait(NULL)>0);
32     mm_helper(i*slice, (i*slice)+leftover,N,A,B,T);
33     memcpy(C, T, sizeof(NUMTYPE) * N * N);
34     munmap(T, N * N * sizeof (NUMTYPE));
35 }

```

2.3 N-Threads matmul

Again, the idea for using N -threads is very similar to using N -processes, except now we don't use shared memory since the threads *do* share memory already. Firstly, we spawn N threads, telling them to perform their share of the matrix multiplication. Then, we join them. As before, special care is taken to account for the remainder since the dimensions of the matrix may not be perfectly divisible by N .

In this case, however, we employ the help of a lock to lock the critical section where we calculate the portion of the matrix multiplication that each thread does. The reason for this is because each thread, running in parallel, decides its portion of the matrix multiplication by reading a global variable `curr_thread`, and then incrementing it for the next thread. Thus, there is the possibility of a data race since the actions are not atomic. Therefore, we lock this critical section, as shown in `mm_threadsn.c`:

```

1  //lock the critical section; might be instances where these values are changed before
   the current thread uses them, affecting correctness of the program
2  pthread_mutex_lock(&lock);
3  int i = curr_thread++;
4  int start = i*N/NUM_THREADS;
5  int end = (i+1) *N/NUM_THREADS;
6  pthread_mutex_unlock(&lock);
7  for (int x = start; x < end ; x++) {
8      for (unsigned int y = 0 ; y < N ; y++) {
9          unsigned int tidx = x + y * N ;
10         C[tidx] = 0;
11         for (unsigned int d = 0 ; d < N ; d++) {
12             C[tidx] += A[d + y * N] * B[x + d * N] ;
13         }
14     }
15 }

```

The rest of the code is fairly straightforward.

2.4 OpenMP matmul

For OpenMP, we just add in a pre-processing directive `pragma` to specify that the compiler should use the OpenMP compiler extension to optimize the code by adding in parallelism. Then, when compiling, we tell the compiler that we are using this extension by adding the flag “-fopenmp” so that it links properly. What’s interesting is that the pragmas are designed so that even if the compiler does not support them, the program will still yield correct behavior, but without any parallelism.

See: `mm_openmp.c`

```

1 void mm (size_t N, NUMTYPE * A, NUMTYPE * B, NUMTYPE * C) {
2     unsigned int x,y,d;
3 #pragma omp parallel for shared(A,B,C) private(x,y,d)
4
5     for ( x = 0 ; x < N ; x++) {
6         for ( y = 0 ; y < N ; y++) {
7             unsigned int tidx = x + y * N ;
8             NUMTYPE tmp = 0;
9             for ( d = 0 ; d < N ; d++) {
10                tmp += A[d + y * N] * B[x + d * N] ;
11            }
12            C[tidx] = tmp;
13        }
14    }
15 }
```

Essentially, `#pragma omp parallel for` starts a team of threads (whose number is determined at runtime) to delegate portion of the `for`-loop to. `shared(A,B,C)` just informs OpenMP what memory is able to be shared, and `private(x,y,d)` says what is private.

3 Evaluation

To generate the plots for this section, just run `make graphs` from the `matmul_process_and_threads` directory. Please pardon my amateur-ish attempt at automating it using the Makefile and python scripting. The requirements are that you have `python3` and `matplotlib` library for `python` installed (works for `python 2` as well I think but I worked in `python 3`).

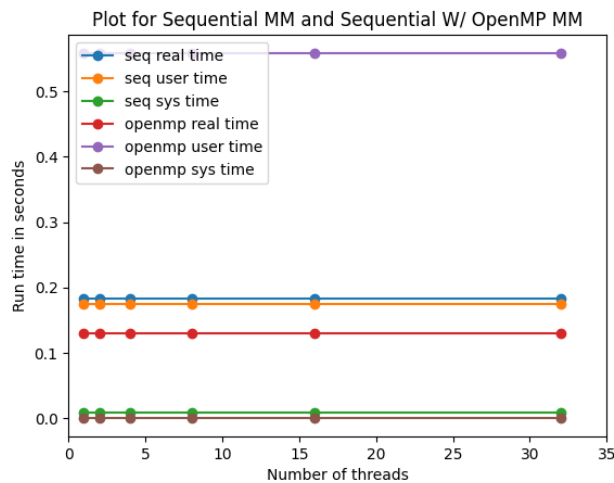
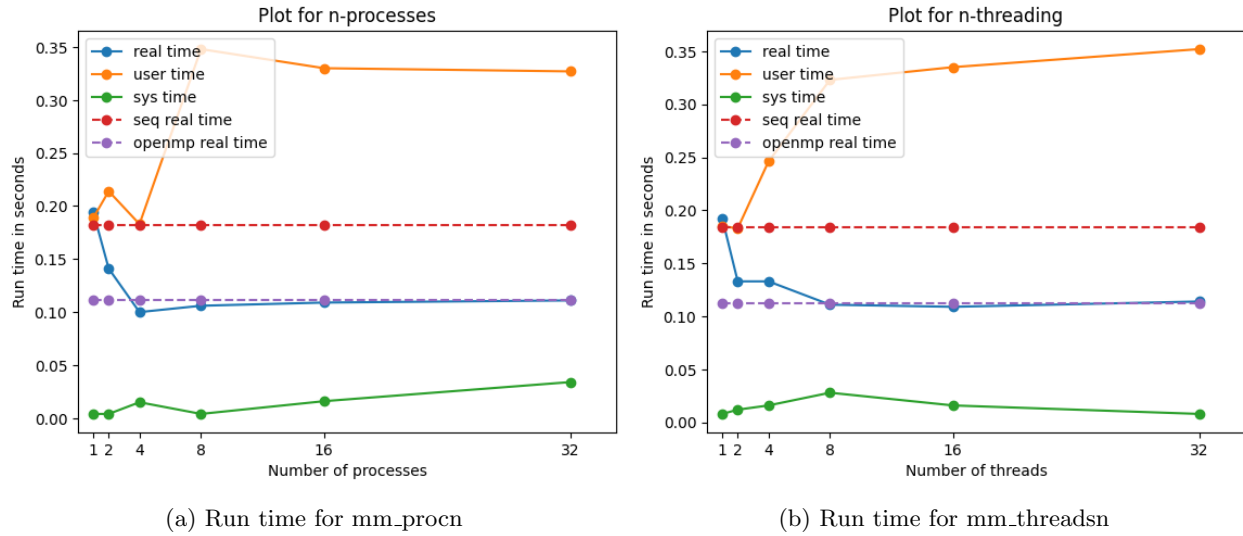


Figure 1: Run time for `mm_seq` and `mm_openmp`

In figure 1, we simply have a horizontal line for the runtime since sequential matrix multiplication is, well,

sequential, and because OpenMP determines at runtime the optimal number of cores, so it is fixed. We see that the OpenMP implementation is significantly faster than the naive sequential one in terms of real time (blue line v.s. red line). However, since OpenMP uses significant multithreading, the user time for OpenMP matrix multiplication (purple line) is very high which signifies the usage of multiple cores and threads.



(a) Run time for mm_procn

(b) Run time for mm_threadsn

For `mm_procn` and `mm_threadsn`, we compare their real, user, and sys times as well. We also overlay the previous sequential and OpenMP real runtimes just for easier comparison. Looking at figures (a) and (b), we see that for both implementations, the real run time (blue line) decreases as more threads are being used (tested on 1,2,4,8,16,32 threads). As one expects, as the number of threads increases, and the real run time decreases, the user time (orange line) increases as the total CPU time is increased since we utilize more of it. We see that the improvements to real runtime stabilizes after around 16 threads, which is probably the upper limit of actual hardware threads available, and after that the threads are just software threads being spawned that don't improve the runtime. We see that both `mm_threadsn` and `mm_procn` perform comparably to OpenMP after 8 threads, which is wonderful because it means my implementation was almost as efficient.