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 across N threads. 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, with the `.tex` file of this report available in the "report" 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`,

```
void mm ( size_t N, NUMTYPE * A, NUMTYPE * B, NUMTYPE * C) {
  int* T = mmap(NULL, N * N * sizeof (NUMTYPE),
    PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS,
    -1, 0);
  int child_pid = fork();
  if (child_pid == 0) {
    mm_helper(0, N/2, N, A, B, T);
    exit(0);
  } else {
    mm_helper(N/2, N, N, A, B, T);
  }
  wait(NULL);
  memcpy(C, T, sizeof(NUMTYPE) * N * N);
  munmap(T, N * N * sizeof (NUMTYPE));
}
```

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

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

```
  int slice;
  int leftover = N % NUM_PROCESSES;
  if (leftover  == 0) {
    slice = N/NUM_PROCESSES;
    while (i < NUM_PROCESSES){
      int child_pid = fork();
      if (child_pid == 0){
  mm_helper(i*slice, (i+1)*slice, N, A,B,T);
  exit(0);
      }
      i++;
    }
    while(wait(NULL)>0);
    memcpy(C, T, sizeof(NUMTYPE) * N * N);
    munmap(T, N * N * sizeof (NUMTYPE));
  } else {//account for when N cannot be perfectly divided by NUM_PROCESSES
    slice = (N-leftover)/NUM_PROCESSES;
    while (i< NUM_PROCESSES ){
      int child_pid = fork();
      if (child_pid == 0){
  mm_helper(i*slice, (i+1)*slice, N, A,B,T);
  exit(0);
      }
      i++;
    }
    while(wait(NULL)>0);
    mm_helper(i*slice, (i*slice)+leftover,N,A,B,T);
    memcpy(C, T, sizeof(NUMTYPE) * N * N);
    munmap(T, N * N * sizeof (NUMTYPE));
  }
```

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`:

```
    //lock the critical section; might be instances where these values are changed before
    the current thread uses them, affecting correctness of the program
  pthread_mutex_lock(&lock);
  int i = curr_thread++;
  int start = i*N/NUM_THREADS;
  int end = (i+1) *N/NUM_THREADS;
  pthread_mutex_unlock(&lock);
  for (int x = start; x < end ; x++) {
    for (unsigned int y = 0 ; y < N ; y++) {
      unsigned int tidx = x + y * N ;
      C[tidx] = 0;
      for (unsigned int d = 0 ; d < N ; d++) {
  C[tidx] += A[d + y * N] * B[x + d * N] ;
      }
    }
  }
```

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`

```
  void mm (size_t N, NUMTYPE * A, NUMTYPE * B, NUMTYPE * C) {
  unsigned int x,y,d;
#pragma omp parallel for shared(A,B,C) private(x,y,d)

  for ( x = 0 ; x < N ; x++) {
    for ( y = 0 ; y < N ; y++) {
      unsigned int tidx = x + y * N ;
      NUMTYPE tmp = 0;
      for ( d = 0 ; d < N ; d++) {
    tmp += A[d + y * N] * B[x + d * N] ;
      }
      C[tidx] = tmp;
    }
  }
}
```

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.

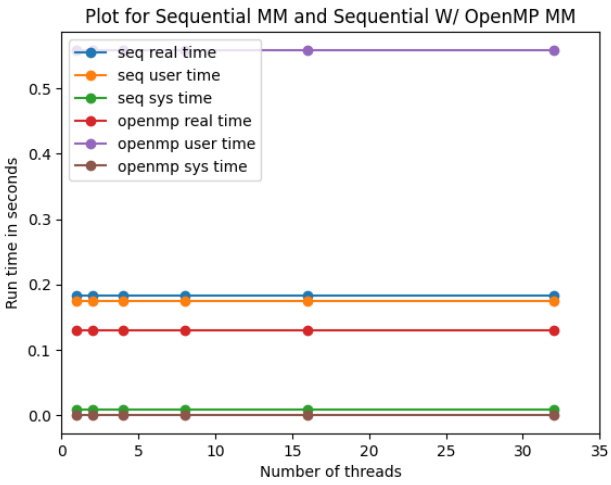
To determine the number of threads that `OpenMP` has access to, we add a flag `OMP_NUM_THREADS=x` where x is the number of threads desired before executing the program.

3 Evaluation

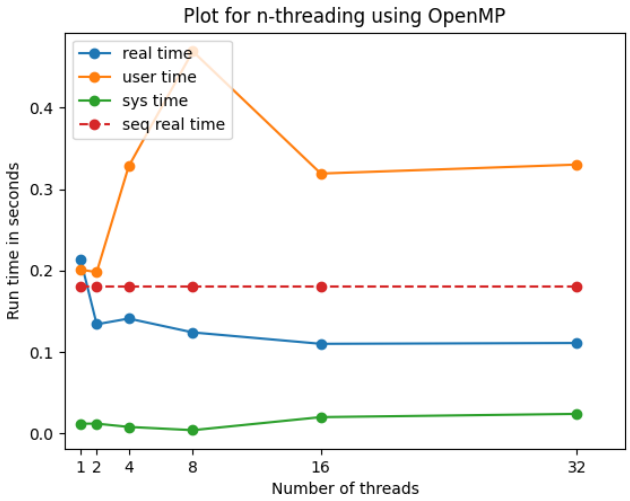
To generate the plots for this section, just run `make graphs` from the `matmul_process_and_threads` directory. Please pardon my amateur-ish attempt at automating it using the Makefile and python scripting. The requirements are that you have `python3` and `matplotlib` library for `python` installed (works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

In figure (a), 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.

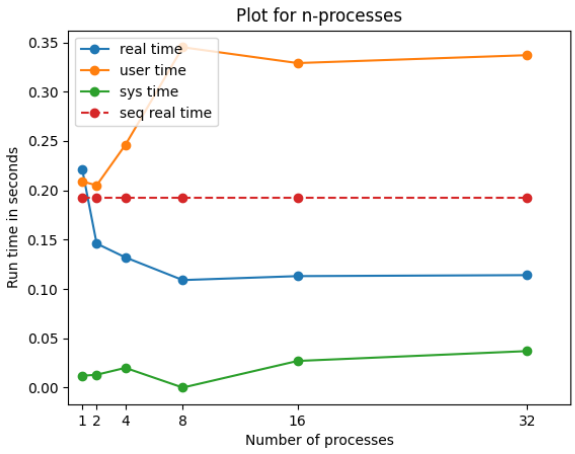
For `mm_procn` and `mm_threadsn`, we compare their real, user, and sys times as well. Furthermore, since we can also change the number of threads that `OpenMP` uses, we also attach the results of that change here. They are represented by figures (b),(c), and (d) respectively. We also overlay the previous sequential real runtimes just for easier comparison. Looking at figures (b),(c), and (d), we see that for all three 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 in my computer, 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.



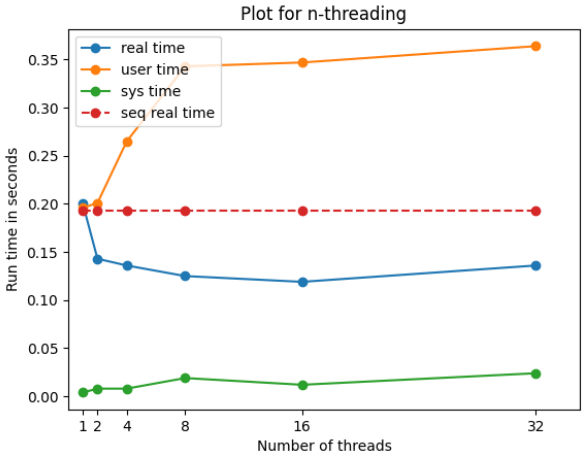
(a) Run time for mm_seq and mm_openmp



(b) Run time for mm_openmp

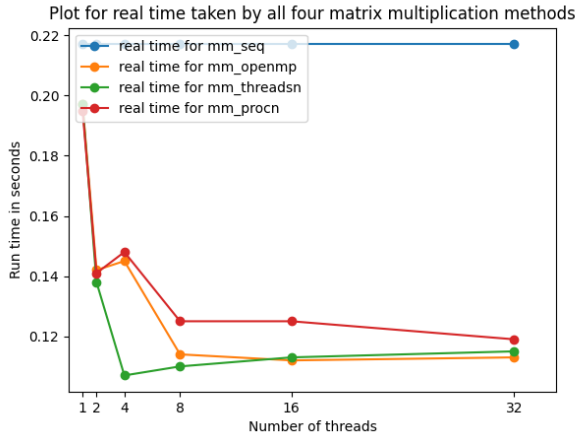


(c) Run time for mm_procn



(d) Run time for mm_threadsn

Finally, we compare the real runtime of all four matrix multiplication implementations:



And we can conclude that `mm_openmp` appears to perform tangentially better than the other two multi-threaded/core implementations, probably because they have other optimizations put in place whereas our implementations of n-threading or processes were just naive implementations.

4 Concluding remarks

In conclusion, in this report we have explored four different implementations of matrix multiplication: sequential, with n-threads, with n-processes, and with the OpenMP framework. In the evaluations of the implementation, we see that the OpenMP implementation performs slightly better than ours, probably because of some micro-optimizations that they perform like accounting for cache-misses or doing loop-unrolling. Nevertheless, it demonstrates that even a naive implementation of concurrency/parallelism can speed up the efficiency of a program by a large factor.