Parallel MMULT, RK38, & RECT using POSIX Threads

Felipe Gómez Lozada

October 14, 2024

1 Introduction

Here we present a parallel implementation for the matrix multiplication, integration by the left rectangular rule, and the Runge-Kutta 3/8 algorithms. Each method has an advantage in this context since its data structure can be partitioned, allowing its performance to largely benefit from the parallelization of its code.

Matrix multiplication consists of the implementation of the following formula

$$C_{i,j} = \sum_{k=0}^{N} A_{i,k} B_{k,j}. \tag{1}$$

where A, B and C are $N \times N$ matrices. The serial code can be written straightforwardly by using the corresponding `for` loops that iterate along each matrix dimension, as shown in the following

```
for(i=0;i<NROW;i++)
    for(j=0; j<NCOL; j++)
        for(k=0;k<NROW;k++)
            outputArrayC[i][j]+=inputArrayA[i][k]*inputArrayB[k][j];
```

In this implementation we will consider the matrices $A_{i,j} = L * i + j$ and $B_{i,j} = (L + 1) * j$.

In the case of integration, we use the rectangular left rule which approximates an integral by sampling the function in a set of equally-spaced points as follows

$$\int_a^b f(x)dx = \sum_{i=0}^{N} f(x_0 + h * i)h, \qquad h = \frac{x_f - x_0}{N}, \tag{2}$$

where f can be any function, integrated in the domain $[x_0, x_f]$ with N sampling points. The associated serial code for $f(x) = \cos(x)$ corresponds to

```
for(i = 0; i < NITER; i++){
    p_current = i*h;
    f_result = cos(p_current);
    area += f_result*h;
}
```

where we use $x_0 = 0$, $x_f = 10$ and `area` corresponds to the integral of the function.

Finally, the Runge-Kutta methods are a family of time-evolution algorithms for ordinary differential equations with initial conditions of the form

$$\frac{dy}{dt} = f(t, y), \qquad y(0) = y_0. \tag{3}$$

In this case, the 3/8 method has the following formula to evolve a system from state n to $n + 1$ with

a time-step h

$$y_{n+1} = y_n + \frac{1}{8}k_1 + \frac{3}{8}k_2 + \frac{3}{8}k_3 + \frac{1}{8}k_4, \tag{4}$$

$$k_1 = hf(y_n), \tag{5}$$

$$k_2 = hf(y_n + \frac{1}{3}k_1), \tag{6}$$

$$k_3 = hf(y_n + k_2 - \frac{1}{3}k_3), \tag{7}$$

$$k_4 = hf(y_n + k_1 - k_2 + k_3). \tag{8}$$

Here we will consider a linear right-hand side function as

$$f(y) = -Cy + p, \tag{9}$$

where C and p are a matrix and a vector given by $C_{i,j} = i*i+j$ and $p_i = 2i$. Also, the initial condition will be given by $[y_n]_i = i * i$. The serial implementation corresponds to separate `for` loops used to calculate each k as follows

```
    for (i = 0; i < PROBLEM_SIZE; i++){
        yt[i] = 0.0;
        for (j = 0; j < PROBLEM_SIZE; j++)
                yt[i] += c[i][j]*y[j];
        k1[i] = h*(pow[i]-yt[i]);
    }
```

then in the last loop, all k vectors are added to the initial condition to find the next step y_{n+1}.

2 Implementation

To parallelized each algorithm we use the library `pthread` that allows multi-thread programming in a shared-memory local environment. Then, each program corresponds to the initialization of shared variables before the creation of the threads, after that, each thread is forked from the main job with a function in which all the parallel code is included. When this part of the code finishes we free the memory from the threads and post-process data such as the time elapsed during the code execution and tests to check that the code is working correctly. All codes are compiled with the command `gcc -g -Wall -O3 -o prog_par.x prog_par.c -lpthread` where `prog` is either `mmult` (matrix multiplication), `rect` (integration) or `rk3_8` (Runge Kutta). It was checked that the optimization flag did not affect the results of the program due to unwanted reordering of the code.

In the case of matrix multiplication, the parallelized function corresponds to the following:

```
void *Pth_mmult(void *pid){
    long my_pid = (long) pid;

    int i,j,k;
    int my_first_row = partition_index(NROW, my_pid);
    int my_last_row = partition_index(NROW, my_pid+1);

    for (i = my_first_row; i < my_last_row; i++)
        for(j = 0; j < NCOL; j++)
            for(k = 0; k < NROW; k++)
                outputArrayC[i][j] += inputArrayA[i][k]*inputArrayB[k][j];

    return NULL;
}
```

The main idea of this code is to partition the set of rows from the resulting matrix along the different threads. To obtain appropriate load-balanced domains for each thread we use the next function to obtain corresponding bounds:

```
int partition_index(long N, long pid){
    int ratio = N/nproc;
    int reminder = N%nproc;

    if (pid < reminder){
        return (ratio+1)*pid;
    } else if (pid < nproc) {
        return ratio*pid + reminder;
    } else {
        return N;
    }
}
```

This receives the number of elements N from the set to partition and a thread id *pid*, then it returns the index of the first element of its chunk of data. It takes into account that if N is not divisible by *nproc*, the total number of threads, we need to distribute the remainder of elements evenly among the threads.

Apart from this last function, the code for parallel matrix multiplication closely resembles the serial program, since each thread can work independently on different parts of the matrix. On the other hand, for the integration problem, all of the samples have to be added to the same memory address, which raises an additional level of complexity for this parallel implementation. In this case, the function that each thread runs is given by

```
void *Pth_rect(void *pid){
    long my_pid = (long) pid;

    double h = (double)(P_END-P_START)/NSTEPS;

    int i;
    int my_first_item = partition_index(NSTEPS, my_pid);
    int my_last_item = partition_index(NSTEPS, my_pid+1);

    double local_area=0.0;
    for (i = my_first_item; i < my_last_item; i++)
        local_area += cos(P_START+(i+0.5)*h)*h;

    pthread_mutex_lock(&barrier_mutex);
    area += local_area;
    pthread_mutex_unlock(&barrier_mutex);

    return NULL;
}
```

Here use again the function `partion_index` and inside each partition the corresponding thread calculates the function in the sampling points and adds them all together in the `local_area` which as its name implies is a local variable to the thread. After this, the objective is to accumulate the local variables in the total `area`. For that, we invoke a mutex called `barrier_mutex` that enforces a critical section for the calculation of the total area. This way we can guarantee that threads won't interfere with each other. This mutex has to be initialized beforehand as a global variable to be included in this function, and its memory must be freed before the code ends.

Finally, we will use a combination of the previous two techniques for the Runge-Kutta method. This is because calculating each k corresponds to a matrix-vector multiplication, where an analogous code to the first program can be used, as shown in the following

```
void *Pth_rk(void *pid){
    long my_pid = (long) pid;

    int i, j;
    double local_sum = 0.0;
    int my_first_row = partition_index(PROBLEM_SIZE, my_pid);
    int my_last_row = partition_index(PROBLEM_SIZE, my_pid+1);

    for (i = my_first_row; i < my_last_row; i++){
        k1[i] = h*poww[i];
```

```
        for (j = 0; j < PROBLEM_SIZE; j++)
            k1[i] -= h*c[i][j]*y[j];
    }

    barrier();
```

Here we first partition the rows along the threads and calculate the corresponding multiplication after initializing the result with the static part of the vector b from the original linear function. Before calculating k_2 all of the threads must finish, since in its calculation a term of the form Ck_1 appears which enforces that each threads need the information of k_1 from the others. To enforce this we implement the function `barrier()` which acts as a thread barrier.

```
void barrier(){
    pthread_mutex_lock(&barrier_mutex);
    barrier_counter++;
    if (barrier_counter == nproc){
        barrier_counter = 0;
        pthread_cond_broadcast(&barrier_condvar);
    } else {
        while (pthread_cond_wait(&barrier_condvar, &barrier_mutex) != 0);
    }
    pthread_mutex_unlock(&barrier_mutex);
}
```

In this case, two structures are needed, a global mutex and a conditional variable. The first one enforces all threads to stop at this function while the second one evaluates the condition that all threads have arrived to release them when this is fulfilled. This way, we analogously calculate each k_i and place a barrier to synchronize the threads between them. At the end of the calculation, we want to sum up all of the resulting vector elements to benchmark the program's results. Then, we use an analog code as the integral implementation.

```
    pthread_mutex_lock(&barrier_mutex);
        totalSum += local_sum;
    pthread_mutex_unlock(&barrier_mutex);
```

where the local sum is calculated on the last `for` loop of the resulting vector.

3 Experimental evaluation

A series of tests were performed for each method with different numbers of threads and system sizes to study their parallel speedup and efficiency. All of them were run on a personal laptop with 6 cores and 12 threads, because of the latter in the next plots a vertical dashed line indicates the maximum number of cores. For each data point, 10 tests were sampled and the minimum time was taken as the result since it is more probable that outside effects will slow down the program than to enhance its performance.

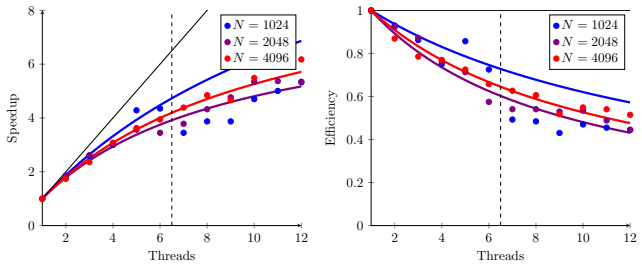

Figure 1: Matrix multiplication for $N \times N$ matrices.

In all three cases (Figs. 1,2,3) we observe an increase in speedup as we raise the number of threads until 6, the maximum number of cores on the computer. After this limit, a drop in both the speedup

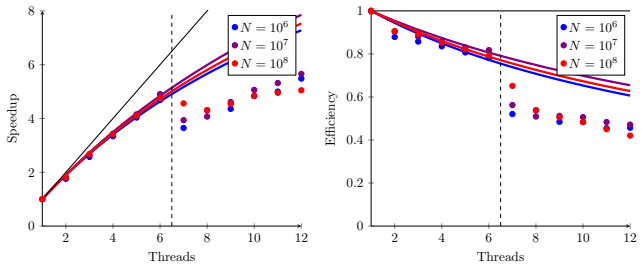

Figure 2: Integration with left rectangle rule with N sample points.

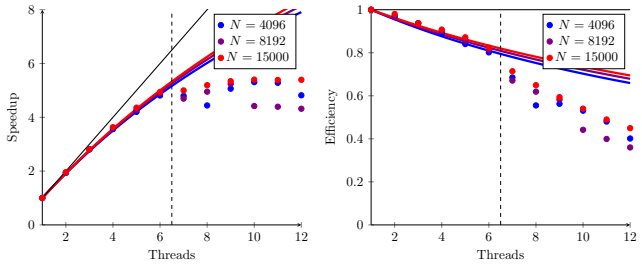

Figure 3: Runge-Kutta 3/8 step with a vector size N.

and efficiency is observed, as expected since now multiple threads have to share the same physical resources. Nevertheless, after this drop, the speedup keeps increasing showing the enhancement of current multi-threading technology in the performance of the programs. Likewise, as we increase the system size in each case an increase in efficiency is evident especially in the region above the limit number of cores, taking advantage of the multithreaded methods.

Moreover, in the left region of each graph, we denote a behavior reminiscent of Amdahl's law given by

$$S(p) = \frac{n}{p + (1-p)n}, \tag{10}$$

where S is the speedup, n is the number of threads and p the percentage of parallelized code. Because of this, we apply a linear fit to $n(S-1)$ vs $(n-1)S$ since the slope corresponds to an estimation of p, then we plotted Amdahl's law with each corresponding color. The latter shows to agree in the expected region for Figs. 2, 3 exhibiting a parallelized percentage around 95%. On the other hand, for Fig. 1 the behavior differs more with lower percentages near 90%.

It makes sense that both Figs. 2, 3 closely resemble Amdahl's law with high p values since both of them only have small critical sections, both at the end of the program to accumulate a global sum and the second one to synchronize threads in between `for` loops. This is not the case for matrix multiplication, as this program does not contain critical parts of code. Nevertheless, even if we isolate a set of rows for the resulting matrix to each thread, due to the operation's structure all threads have to access multiple regions of both operand matrices, which is prone to false sharing between coherent cache memories across different threads. Other implementations such as in GPUs or TPUs can avoid this problem since by having a much larger number of threads they can localize the information of each in the matrices, avoiding overlap between their data domains and removing this cache-related effect.

4 Conclusions

We performed an analysis of the matrix multiplication, integration by left rectangular rule, and Runge-Kutta 3/8 algorithms' parallel implementation, using the speedup and efficiencies compared with the serial code. A principal result corresponds to the high speedups obtained for the last two methods, where we identified the main bottleneck as the critical section necessary for the accumulation of a global sum in each case. We acknowledge that there are more specialized algorithms for this task that can improve this efficiency, especially for a larger number of threads, nevertheless, for the scale of this

study, we found the method used to be enough. On the other hand, the limit of cores in the system showed the advantages of multi-threading technologies, which allow speedups to increase even after surpassing this threshold. We expect this behavior to be more evident for larger problem sizes. Finally, the implementation of matrix multiplication was shown to be the less efficient of the three methods, we associated this with multiple false sharing events that decreased the performance of the program and suggested that algorithms written for GPU or TPU architectures would be more beneficial for this task.