Assignment 3: pthreads

The purpose of this assignment is for you to learn more about

- loop schedulers
- task schedulers
- overhead associated with thread management and synchronization

As usual all time measurements are to perform on the cluster.
Link with the pthread library by passing `-lpthread`.
Both schedulers have a first sequential part where you will build a simple application first; and a second part where you will build the scheduler and the parallel application.

1 Loop Schedulers

Loop schedulers are used essentially whenever the code reads like your stereotypical for loop. In the previous assignment, transform would be a stereotypical for loop, but reduce could also be written this way. Here we will look at a variation on the theme of reduction.

1.1 Numerical Integration

Numerical integration is often used when one wants to compute $\int_a^b f(x)dx$ but one does not know how to find a primitive of f. You can use the definition of integration to obtain a simple approximation by computing $\sum_{i=0}^{n-1} f(a + (i + .5) * \frac{b-a}{n})\frac{b-a}{n}$. n is often called the number of point in the approximation. (This is the numerical integration using the rectangle rule. You can learn more at `https://en.wikipedia.org/wiki/Numerical_integration`.)

Note that you do not need to understand numerical integration. The problem is just to evaluate $\sum_{i=0}^{n-1} f(a + (i + .5) * \frac{b-a}{n})\frac{b-a}{n}$ for particular a particular combination of a, b, n, and f.
Question: Implement a sequential code that compute a numerical integration of a function *double* f(*double*) between `argv[1]` and `argv[2]` using `argv[3]` points.

Note that if you pick f to be a known function, you can easily check for correctness. (I suggest you pick $f(x) = 1$.) Also, writing the code so that the integration loop is with integers (over the number of points) will make your life easier in the next part.
Question: Write the code so that you can use different functions f that have different operation intensities `argv[4]`.

(An easy way to do that, is to have f start from a value of 1 and execute `argv[4]` loops of squaring with `pow` and square rooting with `sqrt`.)
Question: Report the sequential time it takes on the cluster to integrate f using different number of points (from 10^1 to 10^8 by multiple of 10) and with different operation intensity (from 1 to 10^3).

Make sure you keep this code around as it is your base for comparisons. Also note that a 10^8 points with an operation intensity of $1,000$ could take an hour to run.

1.2 Writing a dynamic loop scheduler

Loop schedulers are essentially managed by distributing ranges of indices to threads when they request them.

The worker threads execute a code that looks like

```
while (!loop.done()) {
  int begin, end;
  loop.getnext(&begin, &end);
  execute_inner_loop(begin, end);
}
```

The implementation of a loop scheduler boils done to implementing the two functions `done` and `getnext`. They can be easily implemented using mutual exclusion.

The size of the interval `[begin;end]` is called the granularity and is usually a parameter of the scheduler.

Pay attention that managing the memory of the loop scheduler can be a bit tricky as you need to be sure that all threads coordinated by the loop scheduler are done.

Question: Write a loop scheduler to compute numerical integration. Write the program so that you can easily set the number of threads that participate in the computation and the granularity of the loop scheduler.

To compute numerical integration, note that the program needs to make sure that the result is correct, to do so, you can enforce the mutual exclusion in three places:

- Within the most inner loop for each step of the numerical integration,
- By locally storing the value in `execute_inner_loop` and adding them to a shared variable once per call.
- By storing one value per thread, and aggregating to the shared variable once it is `done`.

Question: By adding a parameter to your program, make sure you can run the three cases.

Question: For a particular granularity (pick 1000), and operation intensity (pick 10), compute the speedup that you obtain with 16 threads for the different number of points in the numerical integration for the three ways to enforcing mutual exclusion.

Question: Compute speed up chart for granularity of 10, 1000, 100,000 using 2, 4, 8 and 16 threads for all the cases you reported sequential time on. (So that should be 32 plots with 3 lines per plot. Pick the mutual exclusion mode that gives the best performance from the previous question.)

2 Task Schedulers

2.1 Strassen's algorithm

Strasssen algorithm is used to multiply matrices. You can find a complete description on Wikipedia at `https://en.wikipedia.org/wiki/Strassen_algorithm`. But for the purpose of this assignment, we will only implement a mock up of Strassen algorithm by following its structure as given in Figure 1.

Note that you do not need to understand how Strassen's algorithm work. One just need to implement the graph from Figure 1. Ignore tasks A, A_{ij}, B, B_{ij}, and C. Implement all other tasks using the function f from the numerical integration problem. We will pick a lower operation intensity for the additions (tasks M_i, N_i, T_i and C_{ij}) and a higher operation intensity for multiplication (tasks P_i).

Question: Implement a sequential mock up Strassen's algorithm as described above. Make sure you keep operation intensity easy to parameterize.

Question: Find proper operation intensity so that the algorithm takes a few seconds. Make the multiplication five times more intense than additions.

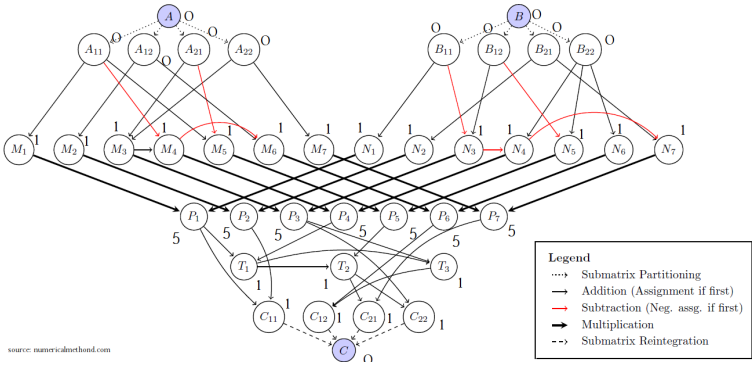

Figure 1: Structure of Strassen's algorithm

2.2 Task schedulers

There are many ways to implement a task based runtime system. Here we will implement a simple task scheduler. A task is defined as :

- an identifier
- a function that will execute it
- a pointer to parameters (that will be given to the function once it is executed)
- a set of identifier of successor tasks. (P_1 is a successor of M_1)
- the number of tasks that precedes it. (P_1 has 2 predecessors)

Threads will essentially wait until a task becomes ready (that is to say, its number of predecessors is zero). Then the thread will take the task and execute it. Once the task's execution is completed, the thread will access all its successor tasks and decrease the number of predecessor by 1.

The task are usually initialized by the master thread before the scheduling begins.

Question: Implement this simple task scheduler.
Question: Implement the mock-up Strassen algorithm in this task scheduler.
Question: Plot a speedup chart for the operation intensities you found in the sequential part.
Question: Vary the operation intensity of both types of tasks. How does the speedup change with 16 computational threads? (That is to say keep a ratio of 1:5, but vary the absolute intensity.)
Question: Vary the ratio of intensity. How does the speedup change with 16 computational threads?

3 Extra credit

3.1 Gantt Chart

Question: Instrument the task scheduler to extract for each task its start time and completion time. Use that to print a Gantt chart of the execution of the Strassen algorithm using different number of threads.

3.2 Thread Pool

A common overhead is the time it takes the system to create threads. Often, runtime systems will create the threads only once and re-use them from one parallel construct to the next.
Question: Implement a thread pool in the loop scheduler and perform multiple numerical integration. What is the impact on speedup for small computation?

3.3 Strassen algorithm

Question: Really implement Strassen algorithm rather than the mock up described above.
Question: Compute speedup charts on matrices of size 1024x1024 and of size 32768x32768.

3.4 Merge Sort

Question: Can you implement merge sort using either schedulers ?
Question: Report speedup for arrays of size ranging from 10^4 to 10^{10}.