Parallelization - Exercises

May 9, 2018

Abstract

In these exercises, you will apply the various techniques and paradigms for code parallelization that were previously introduced in the lectures. For this purpose, you are given several serial codes to be parallized. You will also find intermediate solutions for each exercise in the `solutions` folder provided to you. While these can be used to allow you to progress within the exercise when getting stuck at a part of it, we highly recommend to only use them as a last resort as the attempt of finding the solution yourself is a significant part of the learning process. All codes can be found in `/users/jschewts/HPC_lectures2018/Day2` on SCIAMA. Copy the whole folder into your own file space.

1 All my threads

Compile and run the source files in `Exercises/all_threads/`. They are running by default on as many processors as available on the node. Try to restrict them to only 4 threads using all of the available techniques (Directive Clause, Environment Variable, Runtime Routine).

2 Reduced Race

Compile and run the two source files in the `Exercises/reduced_race`. They are both implementing a simple vector dot-product. One of them contains a race condition. Can you find and correct it?

3 A "boss"/"unpaid worker" model / functional decomposition

Many programs, especially those in charge of time-critical analysis, use a master/slave model to distribute work away from the main thread to keep it responsive for new input.

(A) **Lonely Boss** - Our story starts with a lonely boss, who runs a company, that receives orders, which she has to process all by herself:

```
void work(int order) {
  sleep(4);
  printf("Order #%d processed!\n", order);
```

```
}

void scan(char* buffer) {
   printf("\nPress <enter> for new order, <q>+<enter> for quit:\n");
   *buffer = getchar();
}

int main(int argc, char** argv) {
  char input;
  int order=0;
  while (1) {
    scan(&input);

    if (input == 'q') {
      exit(0);
    } else {
      order++;
    }

    printf("<Boss> Confirm new order: #%d!\n", order);
    work(order);
  }
}
```

Compile the serial source file that you can find in the `master_slave` folder with the compiler of your choice. Run it and observe, how order requests queue up while the boss is busy processing a previous one.

(B) Help the boss by parallelising the code with OpenMP tasks (hint: let only a single thread create new tasks). Run your optimized version and notice how the boss remains responsive while his "unpaid workers" take over the processing.

(C) Write a parallelization with POSIX Threads.

4 To Order!

In the folder `to_order`, you can find the source code file for a serial routine (`serial.c`) that creates an array of random variables and then runs through a loop to check if these variables are ordered or not. It reapeats that process twice and combines the results.

There is also an unfinished mpi program (`mpi.c`) where each process creates a single random variable. Try to complete the program by adding the communication part to have each process (rank i) compare its variable with those of the neighbouring ranks (i-1, i+1) to determine if the generated variables are ordered. Gather/Reduce the results at the end to have a definite answer on rank 0.

Also try out both blocking and non-blocking communication if possible.