# 15-418/618, Fall 2024
# Assignment 4
# Parallel VLSI Wire Routing via MPI

| | |
|---|---|
| Assigned: | Wed., Oct. 9 |
| Due: | Wed., Oct. 30, 11:59pm, 11:59 pm |
| Last day to handin: | Sat., Nov. 2 |

## Overview

Building upon the previous assignment, you will now be using *MPI* to write a *message passing* version of the same application. Before you begin, please take the time to review the course policy on academic integrity at:

```
http://www.cs.cmu.edu/~418/academicintegrity.html
```

Download the Assignment 4 starter code from the course Github using:

```
git clone https://github.com/cmu15418f24/asst4.git
```

## MPI Resources

If you do not have an experience with MPI, we strongly recommend going through the MPI tutorial at https://hpc-tutorials.llnl.gov/mpi/ before you attempt doing the assignment. Learn how to compile and run MPI binaries on the same host and across the nodes of a cluster. Study, compile, and run at least a few of the exercises (Hello World, Array Decomposition, etc.) listed at https://hpc-tutorials.llnl.gov/mpi/exercise_1/. Other MPI resources include the documentation pages on MPI Forum and Open MPI.

To use OpenMPI on the PSC machines, you must first run

```
module load openmpi
```

to load the OpenMPI module. (This loads “`openmpi/4.1.1-gcc8.3.1`” as the default, but you can also choose different compiler versions from the list after running “`module avail openmpi`”.) To run a program with MPI, use the `mpirun` command. The handout also provides an example MPI program `sqrt3` that approximates sqrt(3).

```
// Outer simulated annealing time steps
for (timestep = 0; timestep < N_iters; timestep++) {
   while (there is still work to do in this timestep) {
      each processor grabs a batch of B wires to be routed;
      for (i = 0; i < B; i++) {
         determine the new route for wire i (within the batch);
         if (the route has changed) {
            update local (private) memory with the new route;
            remember this updated route (to propagate it later);
         }
      }
      // Note: this propagation step can be done asynchronously
      propagate updated routes from this batch to other processors;
      // Remember to also process updates from other processors
   }
}
```

Figure 1: Pseudo-code for the outer control structure of a message-passing version of the Across-Wires approach with batching (where $B$ is the batch size).

## Programming Task: Message-Passing Version of VLSI Wire Routing

Please refer to the Assignment 3 handout for a general discussion of the VLSI wire routing problem. Unless we specify otherwise, please use the same problem setup from Assignment 3 (e.g., wire routing constraints, structure of the simulated annealing algorithm, etc.).

Recall that in Assignment 3, we discussed two different parallelization strategies for the VLSI wire-routing problem: *within-wires* and *across-wires*. For this assignment, we are not requiring you to implement both approaches: you can choose either one. (You can also explore both approaches, or even a hybrid approach, but this is not a requirement.) Just keep in mind that because sending messages is relatively expensive, experienced MPI programmers generally try to structure their code to avoid frequent communication. We are interested in seeing good performance from your code, so if your initial approach does not scale well, be sure to explore various options for improving your code.

### Batching Revisited

Recall that in Assignment 3, your *across-wires* strategy included a *batch size* parameter. In this assignment, you will also be using batching, although there are some differences in how this affects the orchestration step, given the differences between the shared-address-space and message-passing programming models. For the message-passing version, batching does not result in delays to updating the local (private) memory with the new route (since no other processors can access a thread's local memory), but instead it creates an opportunity to batch together updates into larger messages. An illustration of this batching approach is shown in Figure 1.

Similar to Assignment 3, the batch size parameter provides a mechanism to make tradeoffs between performance and the quality of the result. Increasing the batch size helps to amortize communication overheads, but it may also have a negative impact on the overall quality of the routing result (i.e. the overall cost may increase). Hence we would like for you to experiment with this batch size parameter, to see how these tradeoffs affect both quality and performance.

## General Advice

**Implications of private address spaces:** This is our first assignment where you only have private address spaces. This requires a change of thinking when you design your algorithm, and it is important to wrap your head around the implications of having only private memory. For example, do not be tempted to try to mimic a shared address space by either sending frequent broadcast messages or treating a single processor as the the memory module, since these approaches do not scale well. Go back and review the message-passing version of the Grid Solver program (in Lecture 5), and think about how it differed from the other versions of the Grid Solver.

**Be careful about using MPI collective communication primitives:** MPI includes a set of primitives for collective communication, including broadcast (`MPI_Bcast`). While these primitives may be convenient for programmers, it is important to realize that the underlying implementations of these primitives is fundamentally expensive. Hence experienced MPI programmers use these primitives sparingly: e.g., to distribute input parameters to all of the processors at the start of the program, to communicate scalar variables at time step boundaries (as we saw with the Grid Solver example in Lecture 5), etc. If you find yourself tempted to use the collective communication primitives frequently to communicate large amounts of data, be forewarned that this may become a significant performance bottleneck for your program (especially at higher thread counts). It is okay to use these operations judiciously, especially if you perform measurements to confirm that they are not hurting scalability.

**Every processor counts:** Sometimes novice MPI programmers are tempted to set aside one processor (e.g., processor 0) as a "manager" that distributes work to all of the other processors (the "workers"). While this approach may simplify some aspects of writing the code, it has some unfortunate scalablity problems. First, when we calculate speedup when running on $P$ processors, this includes *all* of the $P$ processors, not just the $P-1$ workers. Therefore if you set one processor aside as a manager and have only $P-1$ processors doing parallel work, it immediately hurts your scalablity (especially with smaller thread counts). A second performance problem with this manager/worker approach occurs with higher thread counts: communication with the manager tends to become a bottleneck in the interconnection network, since the $P-1$ workers are all trying to send and receive messages from it at the same time (causing it to become a hot spot, often incurring significant queueing delays).

Hence experienced MPI programmers prefer to use all $P$ processors to perform parallel computation in a fairly symmetric fashion with peer-to-peer communication. One example of such a communication structure is a software-managed ring, which enables processors to use point-to-point send/receive messages to communicate effectively. Another advantage of the ring is that when a processor receives a message on the ring that it originally initiated, it knows that the message has been seen by all other processors. Beyond rings, there are other options for structuring your communication so that it is relatively symmetric.

**Think carefully about the design of your messages:** In the Grid Solver example from Lecture 5, most of the messages that were sent were simply boundary rows. For the assigment, you may need to put more thought into the structure of your messages. There are opportunities for reducing communication overhead if you think about this carefully. At the same time, be careful about the functionality of propagating updates asynchronously (you don't want to end up with corrupted occupancy arrays).

**Differences between sending messages within and across chips:** Within a processor chip, message passing can be implemented (under the covers) using memory copy operations. Across separate processor chips, however, messages are passed across interconnection networks. Therefore the latency and bandwidth properties of sending messages in MPI can change noticeably once the computation scales beyond individual processor chip boundaries (i.e. at higher thread counts).

## Logistical Details

Unless specified otherwise, please refer to the details in the Assignment 3 handout regarding command line flags, the formats of the input and output files, and strategies for measuring performance.

**Input files:** Same as Assignment 3 - refer to `/code/inputs/timeinput`.

**How to run:** Use the `mpirun` command to run (both on GHC and PSC machines). Refer to the mpi man page for further details. Here is an example that runs the `easy_4096.txt` test with `<numprocs>` number of process.

```
    mpirun -np <numprocs> ./wireroute -f ./inputs/timeinput/easy_4096.txt
-p 0.5 -i 5 -b 8 -m A
```

**Guidelines on performance and cost targets:** To give you a sense of whether your code is in the right ballpark in terms of its performance and the quality of the routing results (i.e. the cost metrics), we have added a table at the end of this handout.

**Using the GHC and PSC machines.** Compared with Assignment 3, we will be doing more extensive performance analysis on the PSC machines for this assignment since they have more cores and faster interconnection networks. However, it is still the case that you should start on the GHC machines to do all of your initial debugging and testing of your code. (Host names for these machines are `ghc`$X$`.ghc.andrew.cmu.edu`, where $X$ is between 26 and 86.) Once your code is stable and scaling well on the GHC machines, please begin running experiments on the PSC Bridges-2 Regular Memory (RM) machines. Here is a tutorial on accessing the PSC machines. You may also find it helpful to refer to the Bridges-2 User Guide. **IMPORTANT:** To help us preserve our limited allocation of time on the PSC machines, please follow these rules regarding using the PSC machines:

- Do not perform any debugging (or signficant editing) on the PSC machines. We are charged the same for debugging time as we are for collecting performance data, and this time is precious. You should debug on GHC, and only transfer code to the PSC machines when it is stable.

- Collect a full set of performance results on the GHC machines before you start running experiments on the PSC machines. Once you are happy with your performance debugging on GHC, then start doing your PSC experiments.
- Submit no more than 30 sets of experiments per day on the PSC machines. (A "set" of experiments is the data necessary for one speedup curve: i.e. a particular configuration on 1, 2, 4, 8, 16, 32, 64, and 128 threads.) There is a signficant lag between when machine time is used up and when the instructors are notified about it. In previous years, we have had problems where a few groups performed large numbers of experiments, using up the bulk of the time allocation for the class (resulting in their PSC accounts being suspended). So please be careful to stay within your 30-experiments-per-day budget. Thanks!

## Your Mission

In this assignment, you will be implementing, evaluating, and iteratively improving a message-passing version of the wire routing problem using the MPI programming model. The goal of this assignment is for you to think carefully about how real-world effects in the machine are limiting your speedup, and how you can improve your program to get better performance. If your performance is disappointing, then it is likely that you can restructure your code to make things better.

In your write-up, please include the following items:

1. **[40 points] Design and performance debugging journey for your message-passing approach:** Provide a detailed discussion of the thought process that went into designing your program, and how it evolved over time based upon your experiments. (See the Piazza post entitled "Performance Debugging"). Specifically, try to address the following questions:
   - What approaches did you take to parallelize the algorithm (including not only your final design, but any other designs along the way)?
   - Include your reasoning for your final implementation choices, including any graphs or tables that helped you make your decisions.
   - Where is the synchronization in your solution? Did you do anything to limit the overhead of synchronization?
   - Why do you think your code is unable to achieve perfect speedup? (Is it workload imbalance? communication/synchronization? data movement?)
   - At high thread counts, do you observe a drop-off in performance? If so (and you may not), why do you think this might be the case?
2. **[15 points] Experimental results from the GHC machines:** For each of these experiments, please collect and present data for 1, 2, 4, 8, and 16 threads for each experiment while running on the GHC cluster (`ghc`$X$, where $X$ is between 26 and 86).
   (a) **Speedup graphs:** Show a plot of the *Total Speedup* and *Computation Speedup* vs. *Number of Processors* ($N_{procs}$). Discuss these results, including any non-ideal behaviors.

(b) **Cache misses:**

i. **Total cache misses:** Show a plot of the total number of *cache misses* for the entire program vs. *Number of Processors* ($N_{procs}$).

ii. **Per-thread cache misses:** Show plot of the arithmetic mean of *per-thread* cache misses (from perf stat -e cache-misses $PROGRAM) vs. *Number of Processors* ($N_{procs}$).

iii. **Discussion:** Discuss the trends that you see in these cache miss plots, including any surprises, and how these numbers relate to your speedup graphs.

(c) **Sensitivity studies:** If your program uses a batch size parameter, then please perform the batch size sensitivity study. Otherwise, please perform the problem size sensitivity study.

i. **Sensitivity to batch size:** Show plots of how varying the per-processor *batch size* ($B$) between 1, 16, and 128 affects both performance and the overall cost metric. Please run these experiments on 8 threads. Discuss the impact of varying $B$ on performance and overall cost, explaining any effects that you see.

ii. **Sensitivity to the problem size:** Show a plot of the *Computation Speedup* on 8 threads with respect to 1 thread where the input problem size is varied using the different input files in `/code/inputs/problemsize` directory. In particular, we are focusing on differences in grid sizes and numbers of wires. Please discuss the impact of varying the problem size on performance, explaining any effects that you see.

3. **[20 points] Experimental results from the PSC machines:** Measure both the performance and the quality of the output for your parallel code running on the PSC Bridges-2 Regular Memory machines. Please collect and present data for 1, 2, 4, 8, 16, 32, 64, and 128 threads.

(a) **Speedup graphs:** Show a plot of the *Total Speedup* and *Computation Speedup* vs. *Number of Processors* ($N_{procs}$). Discuss these results, including any non-ideal behaviors.

(b) **Comparison with GHC results:** Discuss how these results compare with the speedup curves that you measured on the GHC machines.

(c) **Sensitivity studies:** If your program uses a batch size parameter, then please perform the batch size sensitivity study. Otherwise, please perform the problem size sensitivity study.

i. **Sensitivity to batch size:** Show plots of how varying the per-processor *batch size* ($B$) between 16 and 128 affects both performance and the overall cost metric. Please run these experiments on 32 threads. Discuss the impact of varying $B$ on performance and overall cost, explaining any effects that you see.

ii. **Sensitivity to the problem size:** Show a plot of the *Computation Speedup* on 32 threads with respect to 1 thread where the input problem size is varied using the different input files in `/code/inputs/problemsize` directory. In particular, we are focusing on differences in grid sizes and numbers of wires. Please discuss the impact of varying the problem size on performance, explaining any effects that you see.

(d) **Scaling across multiple nodes:** Evaluate the performance of scaling beyond a single node to multiple nodes. Please collect and present data for 2, 8, and 64 threads on a single node versus the same number of threads (per node) running on 2 nodes (i.e. a total of 4, 16, and 128 threads in the latter case). Discuss your results, and any lessons that you have learned from your analysis.

## Hand-in Instructions

You will submit your code via Autolab and your report via Gradescope. For the code, you will be submitting all C++ header and source files in the `code` folder.

1. Submitting your code:
   (a) If you are working with a partner, form a group on Autolab. Do this before submitting your assignment. One submission per group is sufficient.
   (b) Make sure all of your code is compilable and runnable. We should be able to simply run make. Please remove excessive print statements, if they were added.
   (c) Run the command "make handin.tar." This will run "make clean" and then create an archive of any C++ source code in `code`. If you find it absolutely necessary, you may modify the Makefile to include other files you have added.
   (d) Submit the file handin.tar to Autolab.
2. Submitting your writeup:
   (a) Please upload your report as file report.pdf to Gradescope, one submission per team, and select the appropriate pages for each part of the assignment. After submitting, you will be able to add your teammate using the add group members button on the top right of your submission.

## Performance Charts

For each table, the column header is the number of procs, and the time in the table is the **computation time** (in seconds). Using command `mpirun -np <nprocs> ./wireroute -f ./inputs/timeinput/<filename> -p 0.1 -i 5 -b 8 -m A`

**easy_4096.txt**
Gates Machine, total cost = 122050

| 1 | 2 | 4 | 8 |
|---|---|---|---|
| 0.49 | 0.31 | 0.19 | 0.13 |

**medium_4096.txt**
Gates Machine, total cost = 601213

| 1 | 2 | 4 | 8 |
|---|---|---|---|
| 9.52 | 6.12 | 4.04 | 3.41 |

**hard_4096.txt**
Gates Machine, total cost = 1032198

| 1 | 2 | 4 | 8 |
|---|---|---|---|
| 11.15 | 7.08 | 4.1 | 3.1 |

**extreme_4096.txt**

Gates Machine, total cost = 23613045

| 1 | 2 | 4 | 8 |
|---|---|---|---|
| 46.81 | 28.95 | 15.2 | 11 |