

Load Balance Schemes

Overview

This assignment was to test practical application of various load balance schemes. Specifically:

- ARR (Asynchronous Round Robin)
- GRR (Global Round Robin)
- RP (Random Polling)
- NN (Nearest Neighbor)

Implementation is in C, using the MPI parallelization library.

General Implementation

For all load balance schemes in this assignment, the following apply.

Program is run via command line. Running the program and required args are provided in project readme.md file.

By default, the makefile runs the program by detecting the number of processor cores on the given machine, and launching that many unique processes. If desired, the makefile can be manually edited to specify a given number of processes to launch, instead. Note that program has a minimum of 4 processes, and any value under this will default to launching 4 processes.

Process Types

Regardless of how many processors will launch, 1 process will always be dedicated as the “main process”, and then all other processes will be “worker processes”. The “main process” does not actually do any load work. Instead, it manages all program output to console, as well as watches for lack of any response from workers for a set amount of time, which it takes as “program done executing”. When main determines the program execution is complete, it sends out kill signals to all workers.

Meanwhile, worker processes will either “process a load” if they have any work to do, or send requests out to fellow workers for additional work. Every time a worker process finishes a load, it will send a message indicating its current status to the main process. Any workers without work will skip sending status messages to main, in favor of requesting work from fellow workers. Note that by default, the program starts by giving worker #1 all the work, and all other workers are empty of work.

Regardless of if a worker has work or not, each worker will periodically check for pending messages from fellow workers, after nearly every action. This is in an attempt to minimize the amount of time that any worker spends waiting for possible work.

Doing Work

Finally, this program does not do any “real” work. Instead, it simulates “doing a set amount of work” by sleeping for a second at a time. Ideally, the workers themselves would sleep upon processing a load. However, for reasons I couldn’t figure out, having the workers sleep at that point seemed to break communication and cause deadlocks. Yet removing the sleep statement and leaving everything else the same would function as expected.

To get around this, I instead had the main process have a core loop of:

- Check if any status updates are present from worker processes, making sure to check all workers.
- Output status update to console, if any status messages were provided.
- Sleep for one second. Due to blocking operations of the worker status messages, this effectively forced them all to also sleep for one second, every time they updated their status (aka, every time they processed a theoretical load).
- Check if program termination messages should send out and handle appropriately.
- Repeat from the top.

Schema Implementation

ARR

ARR is implemented as expected. Each process has it’s own counter, initialized to the process’ own `process_id` value. Each time the process makes a work request, the counter is incremented, and it loops back down to 1 after requesting from the process with the highest id.

GRR

GRR normally would have a shared memory value accessible to all processes/threads. However, due to the distributed-memory nature of MPI, I decided it’s more fitting to stick to distributed memory, and handle GRR with message passing.

Thus, the “main process” also contains a “grr counter” value. When a given worker needs to contact another worker for work, it first checks with the main process to get the current grr counter value.

Note that this is somewhat similar to a “schedule based” scheme, except that SB would go a step further. Instead of just keeping a counter and sending that value to workers, it would also keep some sort of data structure (likely a queue) to hold which workers are most likely to contain work to split.

RP

RP is implemented as expected. On needing to send a work request, the worker will contact an entirely random other worker.

Code-wise, this is probably the simplest and shortest one to implement.

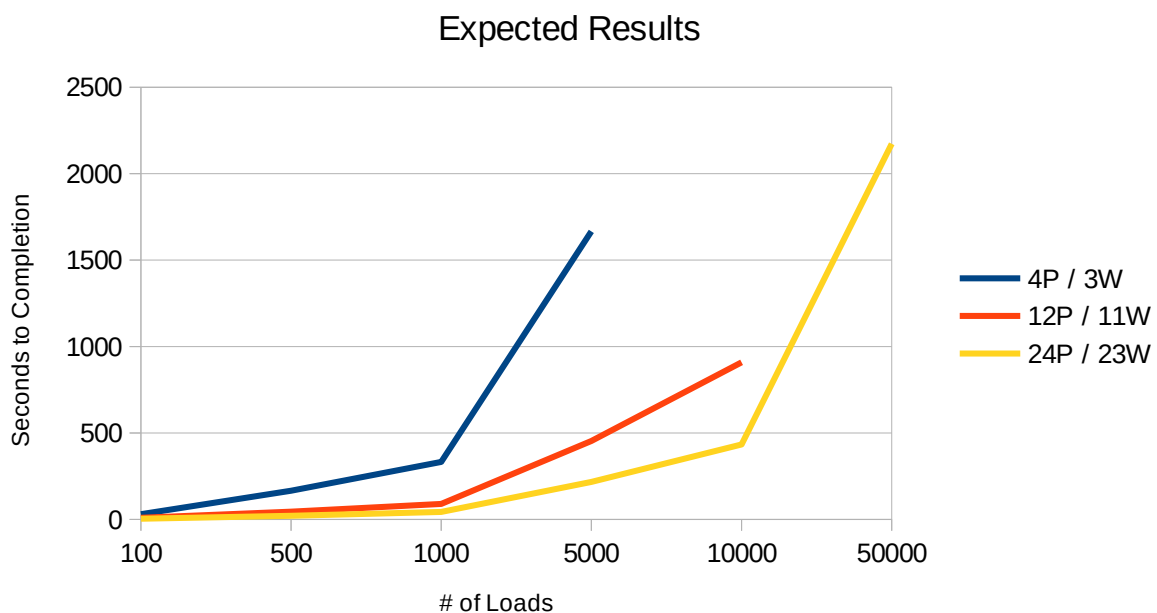
NN

Nearest neighbor is implemented identically to ARR, except that each process’ counter has a limited range. This range is defined by “plus or minus 20% of the number of all workers”. Aka, each counter will stay within the 40% nearest other workers, going based of process id.

For any workers too low or too high, this range will loop around, if necessary.

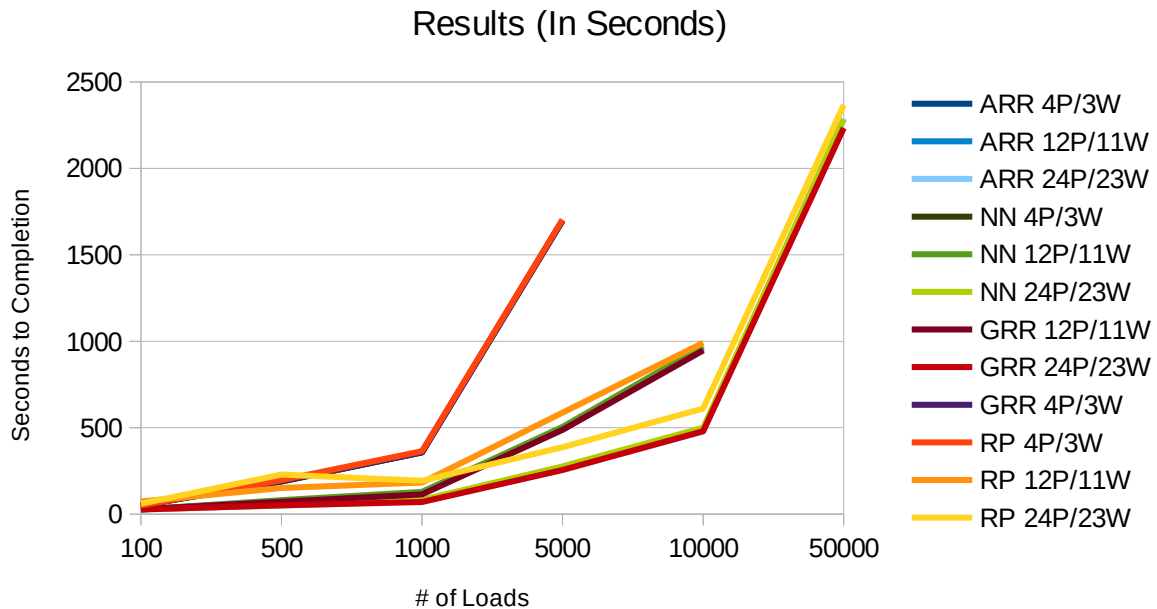
Results

Note: For exact result values, see files in <project_root>/documents/results/



General Color Scheme:

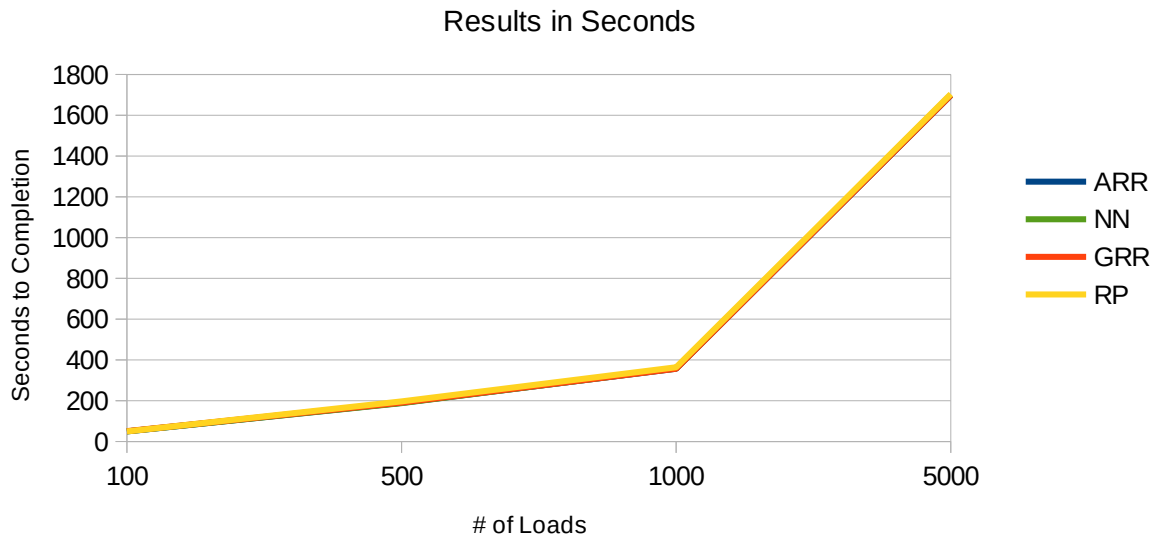
- ARR - Blues
- GRR - Purples/Reds
- RP - Oranges/Yellows
- NN - Greens



At first glance, all of the schemes performed similarly, with the exception of RP that generally seemed to take the longest.

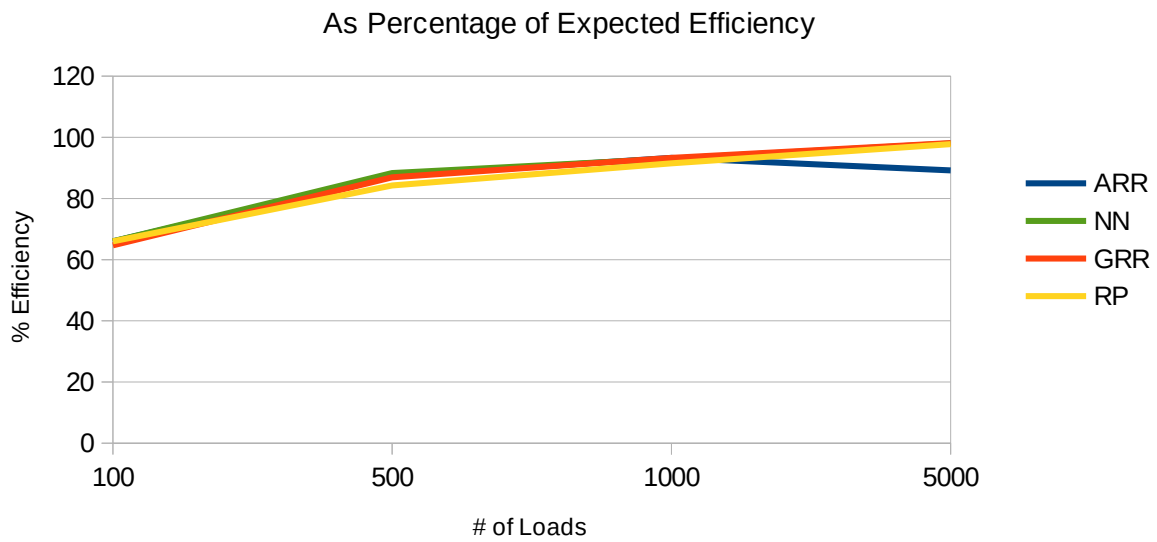
Note that “24 processor RP” actually seemed to take longer than the 12 and 4 processor tests with other schemes.

4 Processors / 3 Workers



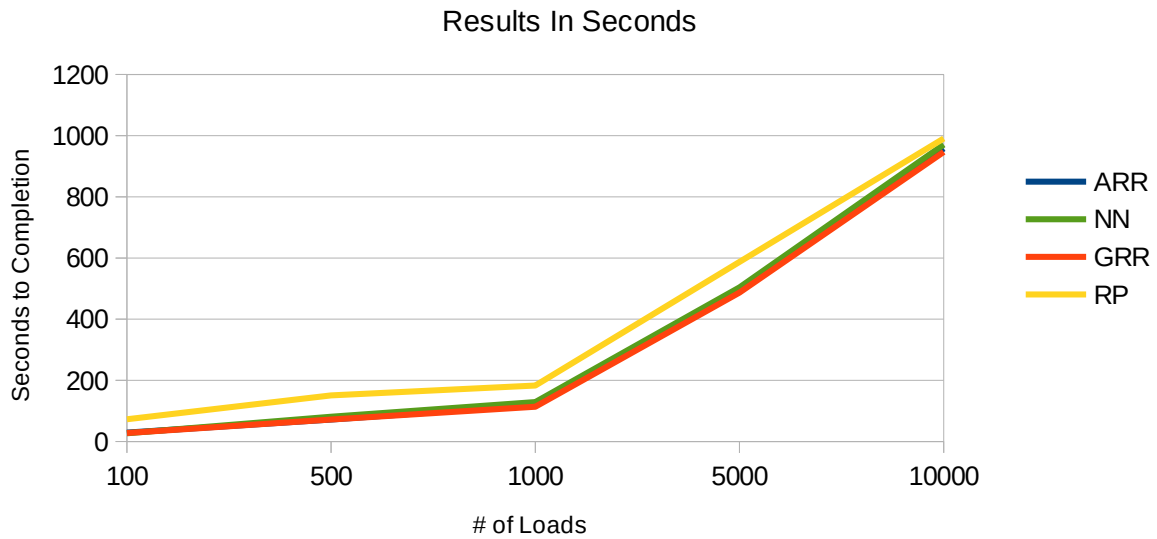
With only 3 workers, all schemes seemed to have essentially identical output.

4 Processors / 3 Workers



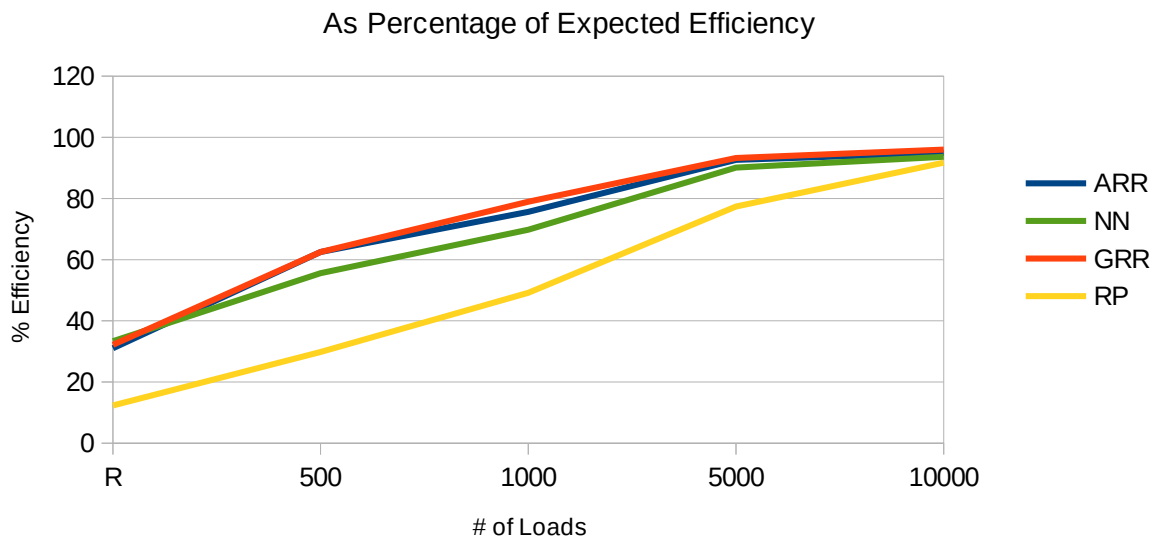
Even when examining efficiency percentages, all schemes perform similarly well with small amounts of workers to split the work between.

12 Processors / 11 Workers



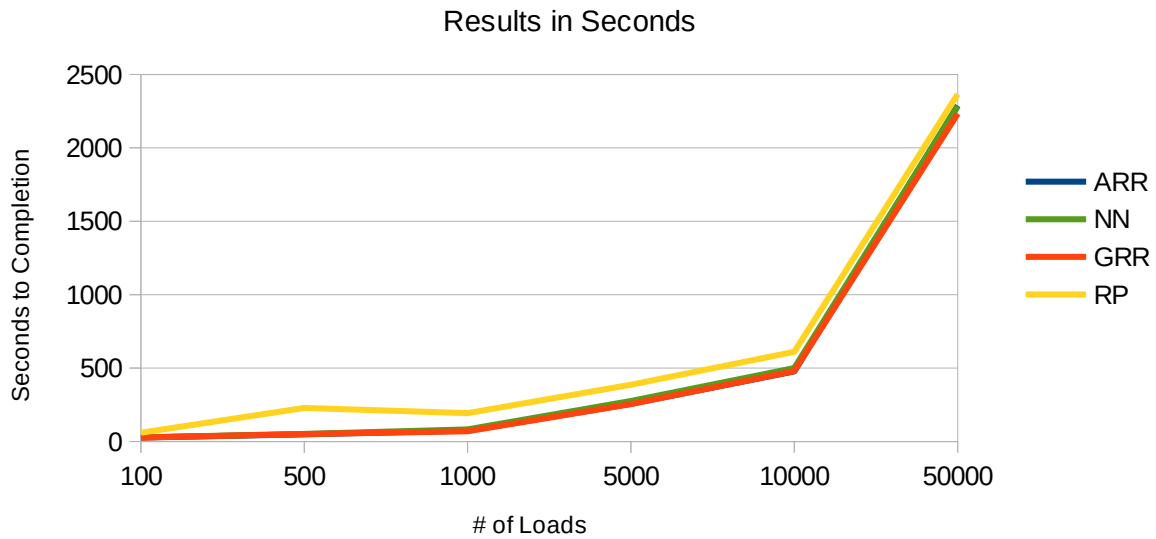
At 11 workers, we start seeing deviation between the different load schemes.

12 Processors / 11 Workers



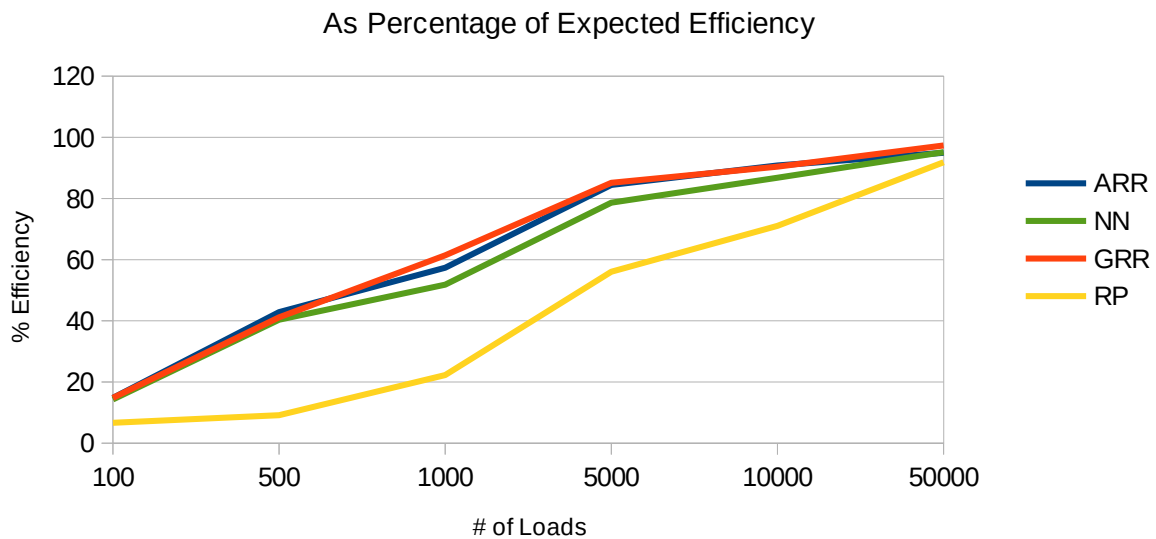
Examining efficiency percentages makes it much clearer which schemes are performing better. Here, rp is noticeably below the others. ARR and GRR perform similarly, while NN is close but starting to slip.

24 Processors / 23 Workers



With 23 workers, it's hard to see much additional change when comparing results by seconds.

24 Processors / 23 Workers



However, examining efficiency percentages tells a different story. Here, RP is far behind in most loads. How far behind also tends to vary, due to the random nature of the scheme.

Surprisingly, as the number of loads increase, RP seems to catch up. I attribute this to higher load counts giving more chances in one run, and so random nature of workload queries tend to average and spread, so to speak.

Conclusion

I was surprised to see indication that the chosen load balance scheme didn't matter quite as much as I had expected. Before seeing these results, I would have anticipated that "SB would be by far the most efficient, with GRR following closely. NN and ARR would both be middle-ground, with ARR generally being the lower one, and then RP being well below the all other schemes".

It turns out that, particularly with low amounts of workers to split work to, the load scheme doesn't seem to impact much at all. They all tend to perform similarly, and the main limitation is the sheer lack of workers to do work. Conversely, as the number of potential workers increases, the chosen load scheme matters more and more, due to the long waits to get initial work when the program starts.

In fact, from my manual observation, that seemed to be the main difference between all schemes. Any scheme that managed to get initial work to all workers the fastest seemed to perform best overall. Once workers all had been given work, finding additional work seemed much easier, even for the less efficient schemes.

On that note, that seemed to be why RP did the worst. Generally, the biggest indicator of an RP run having low efficiency seemed to boil down to "how long does it take for a first worker to randomly choose the right counter so that work can split the first time." As soon as work was split at least once, then RP workers seemed to have a much easier time getting additional work. But until that point, sometimes they really struggled. This explains my "500 Load" RP run actually took longer than my "1000 Load" RP run.

Further Examination

Ideally, this kind of testing should probably be done on an actual problem, instead of with imaginary work. I can't help but wonder if results may be different when workers are actively swapping in and out of memory, actively trying to do work, and competing with each other for processor space. Instead, my implementation had each worker do negligible work (mostly communication), and then sleep.

Furthermore, due to time constraints, I updated loads in uneven increments, and stopped further tests once values started hitting roughly 90% efficiency, or started taking longer than half an hour to run. I also only ran each test type exactly once.

An ideal test would be more thorough, using consistent load increments, and would probably run a minimum of 10 runs per test. These runs should probably record the best time, worst time, and average time, and then compare those values.