Jarred Parr November, 2018

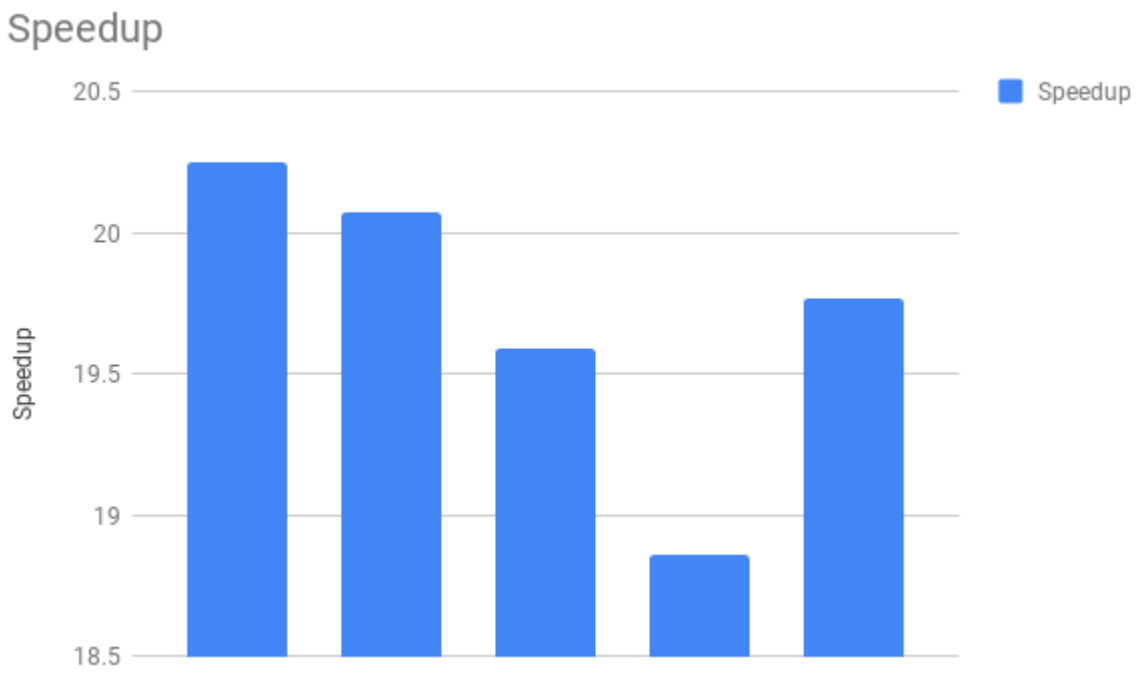
Message-passing Computing

Questions

1. For MPI, the difference in the count parameter for `MPI_Recv()` and `MPI_Get_count()` is very simple. For `MPI_Recv()`, the count parameter is used to indicate the maximum size of the message passed to the receive function. This does not five the actual length of the message though. Instead, the developer must use the `MPI_Get_count()` function to handle getting the actual size of the received message. According to documentation and sources online, Recv simply defines the maximum space for the value, and Get_count gets the actual value.
2. The benefit of non-blocking communication is that it allows for multiple concurrent messages or jobs to be handled at once. They can also be handled upon receipt asynchronously and as long as there is not a need for the output of one task to be used in another, there really isn't a problem. The main problem that can arise is when one of these situations does come up, like in the case of writing to the same memory location for multiple jobs, or the need for all jobs to be complete before continuing onward. This will require a blocking operation in order to handle this task; the addition of all the hardware overhead needed to do parallel computing, we have the potential to see degraded runtime when compared to the sequential implementation. There is a fine line when it comes to blocking vs non blocking depending on the context.

Practical

Upon inspection of the code base on GitHub in search of something to compare, I stumbled upon the built in broadcast message. I noticed that they take a lot of precautionary steps and keep with the general flow of the library. This made me hypothesize in my head what a naive implementation might look like and how this could apply when being compared to the one that exists inside of the actual MPI library. Overall, I found that the naive implementation presented lower overhead overall and allowed for a very consistent speedup to be observed across the board.



The data was distributed in the following:

Speedup	Run Average Size
20.25	10
20.07	100
19.59	1000
18.86	100000
19.77	100000000

```
#include <chrono>
#include <cstring>
#include <iostream>
#include <mpi.h>
#include <sstream>
#include <string>
#include <unistd.h>

class MPIComparison {
  public:
    void compare_broadcast(void* data, int num_elements);
  private:
    const int MASTER = 0;
    const int TAG = 0;
    const int MSGSIZE = 100;
    const int MAX = 25;
```

```
};

void MPIComparison::compare_broadcast(void* data, int num_elements) {
  int rank, num_nodes;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &num_nodes);
  std::string message("");
  std::string host("");
  std::ostringstream mess_buf;

  if (rank == this->MASTER) {
    // If we are the master, send our stuff to everyone
    gethostname(const_cast<char*>(host.c_str()), this->MAX);
    mess_buf << "Process: " << rank << ", host: " << host;
    message = mess_buf.str();

    for (int i = 0; i < num_nodes; ++i) {
      if (i != rank) {
        MPI_Send(data, num_elements, MPI_CHAR, i, TAG, MPI_COMM_WORLD);
      }
    }
  } else {
    char mes[MSGSIZE];
    MPI_Recv(mes, MSGSIZE, MPI_CHAR, MASTER, TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    std::cout << std::string(mes) << std::endl;
  }
}

int main(int argc, char** argv) {
  MPIComparison mpic;
  if (argc < 3) {
    std::cerr << "usage: comp iterations num_items" << std::endl;
    return EXIT_FAILURE;
  }

  // How many times to send broadcast
  int iterations = std::stoi(argv[1]);

  // How many items to send
  int num_items = std::stoi(argv[2]);

  MPI_Init(nullptr, nullptr);

  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  double total_standard_time = 0.0;
  double total_broadcast_time = 0.0;
  int* data = (int*)malloc(sizeof(int) * num_items);

  // Test over number of iterations
  for (int i = 0; i < iterations; ++i) {
    // Syncrhonize before beginning (to be safe)
```

```
    MPI_Barrier(MPI_COMM_WORLD);

    total_broadcast_time -= MPI_Wtime();
    mpic.compare_broadcast(data, num_items);
    MPI_Barrier(MPI_COMM_WORLD);
    total_broadcast_time += MPI_Wtime();

    MPI_Barrier(MPI_COMM_WORLD);
    total_standard_time -= MPI_Wtime();
    MPI_Bcast(data, num_items, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Barrier(MPI_COMM_WORLD);
    total_standard_time += MPI_Wtime();
  }

  std::cout << "Average standard time: " << total_standard_time / num_items <<
std::endl;
  std::cout << "Average broadcast time: " << total_broadcast_time / num_items <<
std::endl;

  MPI_Finalize();

  return EXIT_SUCCESS;
}
```