Introducing MPI

Exercises

1. Examine the following code.

```cpp
#include <iostream>
#include <mpi.h>

using std::cout, std::endl, std::flush, std::string;

int main()
{
    MPI_Init(nullptr, nullptr);

    int numProcs;
    MPI_Comm_size(MPI_COMM_WORLD, &numProcs);

    if (numProcs != 2)
    {
        cout << "This example should use 2 processes." << endl;
        return 0;
    }

    int procId;
    MPI_Comm_rank(MPI_COMM_WORLD, &procId);

    MPI_Status stat;
    switch (procId)
    {
    case 0:
        int m = 0;
        MPI_Send(&m, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
        MPI_Recv(&m, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &stat);
        cout << "Process 0 received " << m << " from 1." << endl;
        break;
    case 1:
        int n = 14;
        MPI_Send(&n, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
        MPI_Recv(&n, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &stat);
        cout << "Process 1 received " << n << " from 0." << endl;
        break;
    default:
        cout << "Should not get here!";
    }

    MPI_Finalize();

    return 0;
}
```

(a) Predict what the output of this code will be, assuming you assign two processes to the task.

(b) Run the code on Blue Crystal. What do you see? (You may wish to first run the code on two cores of a single node and then on one core on each of two nodes, just to see if this makes any difference.)

(c) Replace the two calls to `MPI_Send` with equivalent calls to `MPI_Ssend` (synchronous send) and run the code again. What do you see?

(d) Can you explain your results.

2. A commonly used MPI algorithm prototype is the *self-scheduling* or *manager-worker* algorithm. In such an algorithm, the manager process is responsible for coordinating the work of the worker processes, handing out a new task whenever a process finishes an old one. Worker processes receive tasks, perform them as requested and return their results to the manager, after which they wait to receive the next task. When the manager has no more tasks to distribute, it responds to workers that complete a task with a message indicating that the work is done.

   This type of algorithm is useful when the worker processors do not need to coordinate with each other and when the time required for each task is highly variable and difficult to predict. (In other words, in those situations where you would use a dynamic schedule in OpenMP.)

   Recall the Collatz conjecture code from previous exercises.

```cpp
#include <iostream>
#include <vector>
#include <limits>
#include <algorithm>
#include <omp.h>

using integer = long long;  // For easy changing of integer type

integer maxCanInc = (std::numeric_limits<integer>::max() - 1) / 3;

integer collatz(integer x)
{
    int count = 0;
    while (x != 1)
    {
        if (x % 2 == 0)
        {
            x /= 2;
        }
        else if (x <= maxCanInc)
        {
            x = 3 * x + 1;
        }
        else
        {
            std::cerr << "Integer overflow. Exiting" << std::endl;
            exit(EXIT_FAILURE);
        }
        ++count;
    }

    return count;
}
```

```
34
35 int main()
36 {
37     using std::cout, std::endl;
38
39     const integer n = 100000000;
40     std::vector<int> counts(n, 0);
41
42     double startTime = omp_get_wtime();
43     for (integer i = 2; i < n; ++i)
44     {
45         counts[i] = collatz(i);
46     }
47     double timeTaken = omp_get_wtime() - startTime;
48     cout << "Time taken: " << timeTaken << endl;
49
50     int mostSteps = std::max_element(counts.begin(), counts.end())
51                     - counts.begin();
52     cout << "Most steps needed from " << mostSteps << " (";
53     cout << counts[mostSteps] << " steps required.)" << endl;
54
55     return EXIT_SUCCESS;
56 }
```

Running this code might produce the following output.

```
Time taken: 27.129
Most steps needed from 63728127 (949 steps required.)
```

(a) Using MPI, parallelise the algorithm so that it uses a manager-worker approach. Initially create tasks consisting of one iteration of the original `for` loop and time your code. (If your code is taking too long to run, reduce `n`. Your code should be able to handle `n` equal to 10,000,000 in less than 10 seconds.)

(b) Modify the code so that a task consists of $m > 1$ iterations of the loop. Comment on your results.

Hint: To send long long integers, use `MPI_LONG_LONG` rather than `MPI_INT`.