Introduction to Supercomputing

TMA4280 · Problem set 3

Exercise 1. Go to `http://top500.org`, a website cataloguing the top 500 supercomputers in the world. Study the top 10, in particular their technical specifications. What is meant by the LINPACK benchmark performance?

Solution. Self study!

Exercise 2. Note that some of this was not covered in detail in the lecture. Please have a look at the lecture notes.

1. What limits the scalability of a bus-based interconnect?
2. How are the individual processors connected using a crossbar?
3. How are the individual processors connected using a mesh?
4. What is the difference between a shared-memory and a distributed memory architecture?
5. What characterizes the memory access in an SMP?
6. What is the difference between a NUMA and a ccNUMA architecture?

Exercise 3.

1. The limitation of a bus-based system lies in the fact that the total bandwith is fixed, and given by the bus. We can easily add more processors, but they must share the same bus, so the total amount of bandwidth available to each processor *on average* is reduced.

   Note that caches can reduce bandwidth demand, since if a value can be found in the cache there's no need to use the bus. This introduces the problem of cache consistency.

2. In a crossbar configuration, there are multiple paths between each processor and memory unit. Here, the bandwith available to each processor remains constant as more processors are added. However, each added processor is associated with much higher cost than in the bus case.

3. In a mesh topology, each processor is connected to the directly neighboring processors according to a fixed regular pattern, resembling a

structured mesh in d dimensions. Each processor in the interior is connected to $2d$ other processors, two for each dimension (negative and positive direction). A variation on this structure is a toroidal system in which the same holds for boundary processors.

4. In a shared-memory system, all processors have access to all the available memory in the same address space. In a distributed memory system, each processor only has local memory access, and data stored in other units can only be retrieved with explicit message passing.

5. In an SMP, the memory access time is (nearly) constant, independent of processor and memory unit.

6. ccNUMA stands for *cache-coherent* NUMA. A ccNUMA system has some form of mechanism for ensuring that the caches of each processor remain consistent with the others and the main memory units.

Exercise 4. How many bytes are sent in each of the three messages listed below? Here given in C, but that's not important.

```
MPI_Send(buf1, 80, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
MPI_Send(buf2, 1024, MPI_INT, dest, tag, MPI_COMM_WORLD);
MPI_Send(buf3, 1024, MPI_DOUBLE, dest, tag, MPI_COMM_WORLD);
```

Solution.

1. Each char uses one byte, so a message of 80 chars uses 80 bytes of memory.

2. On most modern systems today, a C int is four bytes, so such a message would require $1024 \cdot 4 = 4096$ bytes of memory. Note that the C standard only requires an int to be at least two bytes.

3. A double is eight bytes, so this message requires $1014 \cdot 8 = 8192$ bytes of memory.

Exercise 5. Is it true that a unique tag must be specified each time `MPI_Recv` is called?

Solution. No, for example it is possible to receive messages with any tag by using `MPI_ANY_TAG` in `MPI_Recv`.

Exercise 6. Implement the program from the previous exercise using MPI. It should run on any number of processes.

Hint: Partition the matrix in column strips.

Solution. In C:

```c
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <time.h>
#include "mpi.h"

double random_double()
{
    return (double)rand() / RAND_MAX;
}

// Converts a local index to a global one
size_t loc_to_glob(size_t loc, size_t rank, size_t nprocs)
{
    return rank + loc * nprocs;
}

int main(int argc, char **argv)
{
    MPI_Init(&argc, &argv);

    int nprocs, rank;
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (argc < 3) {
        // Only one process needs to print usage output
        if (rank == 0) {
            printf("Usage: ex2.5 n g\n");
            printf("  n = vector/matrix size\n");
            printf("  g = constant gamma\n");
        }

        MPI_Finalize();
        return 1;
    }

    // All processes can read arguments
    int n = atoi(argv[1]);
    double gamma = atof(argv[2]);

    // The total number of elements associated to this process
    // It will own elements rank, rank+nprocs, rank+2*nprocs, ...
    int np = 0;
```

```c
for (int i = rank; i < n; i += nprocs)
    np++;

// Allocate vectors and matrices (this is valid C99)
// We only allocate the elements owned by this process
double a[np], b[np], x[np], A[n][np];

// Fill a, b and A with random data in [0,1]
// We do this on each process separately
srand(time(NULL) + rank);
for (size_t i = 0; i < np; i++) {
    a[i] = random_double();
    b[i] = random_double();
    for (size_t j = 0; j < n; j++)
        A[j][i] = random_double();
}

// Compute x
// -----------------------------------------------------------

// There is no need to communicate x to other processes
for (size_t i = 0; i < np; i++)
    x[i] = a[i] + gamma * b[i];

// Compute y
// -----------------------------------------------------------

// Each process contributes data to all elements in y,
// so we need to allocate n elements
double y_temp[n];
for (size_t i = 0; i < n; i++) {
    y_temp[i] = 0.0;
    for (size_t j = 0; j < np; j++)
        y_temp[i] += A[i][j] * b[j];
}

// Add in the contribution from the local a
for (size_t i = 0; i < np; i++)
    y_temp[loc_to_glob(i, rank, nprocs)] += a[i];

// Sum over all processes to get the actual y
// Since each process needs it in the next part,
// we have to use MPI_Allreduce
double y[n];
```

```c
    MPI_Allreduce(y_temp, y, n, MPI_DOUBLE,
                  MPI_SUM, MPI_COMM_WORLD);

    // Compute alpha
    // ----------------------------------------------------------

    double alpha_temp = 0.0;
    for (size_t i = 0; i < np; i++)
        alpha_temp += x[i] * y[loc_to_glob(i, rank, nprocs)];

    // Sum over all processes
    // This time it's only important that root gets the answer
    double alpha;
    MPI_Reduce(&alpha_temp, &alpha, 1, MPI_DOUBLE,
               MPI_SUM, 0, MPI_COMM_WORLD);

    // Final output
    // ----------------------------------------------------------

    if (rank == 0)
        printf("alpha = %f\n", alpha);

    MPI_Finalize();
    return 0;
}
```