Introduction to Supercomputing

TMA4280 · Problem set 5

Exercise 1. Answer the following questions.

1. On which multi-processor systems is it of interest to use message passing?
2. What are the advantages of using a standardized communication library (or message passing library) such as MPI?
3. A communication library consists of many specific message passing operations. How would you classify these operations into a few main groups, or types of communication patterns?
4. Explain what is wrong with the following code segment. It is written in C but the language is not important.

```
MPI_Comm_rank(comm, &rank);
if (rank == 0) {
  MPI_Recv(recvbuf, count, MPI_DOUBLE, 1,
           tag, comm, &status);
  MPI_Send(sendbuf, count, MPI_DOUBLE, 1, tag, comm);
}
else if (rank == 1) {
  MPI_Recv(recvbuf, count, MPI_DOUBLE, 0,
           tag, comm, &status);
  MPI_Send(sendbuf, count, MPI_DOUBLE, 0, tag, comm);
}
```

Solution.

1. Message passing can be used on all multi-processor systems, but is mostly relevant on distributed memory systems, where it is the only viable way to communicate between different processes. On shared memory systems you can use shared memory between threads as an implicit form of communication.
2. The advantages of using a standardized *anything* is that your code can (should) run on any implementation of MPI that conforms with the MPI

standard. In practice, that means your code will run without changes on your own computer (which runs e.g. OpenMPI or MPICH or a similar library) and on Vilje (which uses Intel's MPI implementation).

3. The main groups of communication routines are: one-to-one, one-to-all (as well as all-to-one) and all-to-all.

4. This is a deadlock situation. Since both processes expect to receive a message before they send one, both of them will block waiting for a message that never appears. The program will stall.

Exercise 2. Assume a distributed memory multiprocessor computer with the following interconnect charateristic: the time $\tau(k)$ it takes to send a message with k bytes can be approximated as

$$\tau(k) = \tau_s + \beta k,$$

where τ_s is a fixed startup time and β is the inverse bandwidth (units of seconds per byte).

For our setup, $\tau_s = 1\,\mu s$ and $\beta = 1.25\,\text{ns}\,\text{B}^{-1}$.

1. How many bytes can we send in a single message before the time to send the message is twice the startup time? To how many (double precision) floating-point numbers does this correspond?

2. How long does it take to send a message with a single floating point number? Is it preferable to send many short messages instead of a single, long one?

Solution.

1. Setting $\tau(k) = 2\tau_s$ and solving for k, we get

$$k = \frac{\tau_s}{\beta} = 800\,\text{B}.$$

2. A (double precision) floating point number is eight bytes, so it takes

$$\tau(8) = 1\,\mu s + 8\,\text{B} \cdot 1.25\,\text{ns}\,\text{B}^{-1} = 1.01\,\mu s$$

This is about half the time for one hundredth of the message length, so it is clearly preferable to send long messages when possible.

Exercise 3. Let $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D}$ be matrices of size $n \times n$, and let the matrix $\boldsymbol{D}$ be constructed as

$$\boldsymbol{D} = 3\boldsymbol{A}\boldsymbol{B} + \boldsymbol{C}.$$

How many floating point operations does it take to construct $\boldsymbol{D}$?

Solution. A scalar (inner) product takes n multiplications and $n-1$ additions, so $2n-1$ operations in total. A matrix-matrix product is n^2 such inner products, so it takes $n^2(2n-1)$ operations. The multiplication with the scalar and the addition of another matrix both require n^2 operations (since they are single operations applied to each element). In total we get

$$n^2(2n-1) + 2n^2$$

operations.

Exercise 4. Implement an MPI-based matrix multiplication program. It should accept one command-line argument, which is the size of the matrix and vector to multiply. Use arbitrary data (e.g. random).

You are free to choose the structure of the matrix and vector, but a typical approach is the following:

- Each process "owns" a certain set of indices.
- Each process only stores the part of the vector that it owns, and the part of the matrix corresponding to the *columns* that it owns.
- The result of the local matrix-vector product is then a full-sized vector, with contributions to the vector chunks of all other processes.

Things to consider:

1. What changes will you have to make if you want to store the matrix on each process by rows instead of columns?
2. For some types of matrices it is possible to minimize communication by cleverly choosing the index sets. What characterizes these matrices?
3. What would a matrix transpose operation look like?

Solution. See the attached file for the source code.

1. If we store the matrix by rows instead of columns (but we keep the vector in the same format), we have to collect the vector *before* the local matrix-vector product, since this product will now expect a "long" vector as input and produce a "short" vector as output. Correspondingly, there will not be a need for communication *after* the local product has been computed, as the result will be immediately usable as the local result vector.
2. The communication in this method occurs when the global matrix-vector product must be assembled on each process after the local producs have been computed. This takes place because the local product on process p may have contributions to elements which are owned by another process $q \neq p$.

However, for many applications the sparsity pattern of the matrix is known in advance, so it is possible to predict exactly which elements must be communicated where: process p must communicate element j to process q (who owns element j) the local matrix $\boldsymbol{A}$ on process p has any nonzero element in row j. If there are only zeros in that row, then clearly no communication is necessary because the contribution will be zero.

In other words, the best possible outcome is if all local matrices on each process only have nonzero elements corresponding to those indices which are owned by that process. Such matrices are *block diagonal*. If the index sets for the processes correspond to the blocks, no communication will be necessary.

In reality this almost never happens, but many matrices are *almost* block diagonal, in which case it is possible to compute matrix vector products with very little communication indeed.

3. For the next exercise!