DM818 Assignment 3

DNS Algorithm

Dan Sebastian Thrane
<dathr12@student.sdu.dk>
Lars Thomasen
<latho12@student.sdu.dk>

Winter 2015

Table of Contents

Chapter 1

Introduction

This report documents the development of an MPI virtual topology, and the implementation of the DNS algorithm covered in DM818.

The implementation takes some assumptions, such as it expects a cubic number p of processors and assumes that the matrices of size $n * n \geq p^{1/3}$.

1.2 Contributions

Programming was done only using pair programming, and as such an equal contribution of this assignment has been done.

Chapter 2

Implementation

2.1 Design choices

No specific design choices has been made. The implementation consists of two phases:

- Create virtual topology using MPI.
- Follow the DNS algorithm.

Creating the virtual topology is a rather simple task, when a good understanding of how MPI works, and thus little to no design choices will be needed.

Implementation of the DNS algorithm left no room for design choices, as it was simply a matter of following the algorithm presented in the course.

2.2 Virtual topologies

The communication pattern of a set of processes can be represented by a graph. The nodes in this graph is represented by processes and the edges between them is the processes communication between each other. MPI provides message-passing between any pair of processes in a group, often represented in a Communicator.

MPI offers built-in methods such as `MPI_Graph_create` and `MPI_Cart_create`. The latter offers the possibility to describe Cartesian structures in more than two dimensions.

The virtual topology was created in MPI using `MPI_Cart_create`, this method takes the following parameters:

- MPI Communicator to which the Cartesian topology information is attached.
- Number of dimensions.

- Number of processes for each dimension.
- Whether the dimensions are periodic.
- Whether ranking may be reordered, if false the rank from the old communicator is retained.

and outputs a new communicator for the Cartesian topology. This is used to create a 3-dimension hypercube.

In order to communicate between the processes in this hybercube, we can create additional Communicators using `MPI_Cart_sub`, specifying which dimensions we want our new communicator to attach to processes in. This helps simplifying the complexity of the communication code for the remaining part of the project.

As an example the following line specifies that we want all processes in the i'th axis in the new communicator `iComm`.

```
int iDimensions[3] = {1, 0, 0};
MPI_Cart_sub(gridCommunicator, iDimensions, &iComm);
```

Whenever we now want to communicate across all processes on the i'th dimension we can now simply use this communicator.

2.3 Matrix not divisible by $p^{1/3}$

In order to make it possible to run this algorithm on matrix sizes which is not divisible by $p^{1/3}$ we choose to apply padding.

Some background to understand this problem can be shown by thinking of the 3-dimensional cube split across p processes. Each dimension needs and equal amount of processes to split the n elements, if the $\frac{n}{p}! = 0$ this will not be possible.

The simple solution is to pad the original matrix with additional elements with a value of 0. The end result will remain the same, at the cost of slight additional overhead multiplying and adding the 0's. The specific number of additional rows/columns added is calculated as following:

$$\frac{n}{p^{1/3}} - n \mod p^{1/3}$$

This calculates the number of elements per process first, and then subtracts the remainder we have. Thus we get the number of rows/columns we need.

Chapter 3

Performance

3.1 Runtimes

Chapter 4

Isoefficiency of the implementation

Need the results first.

Chapter 5

Testing

Confirming that the final results are correct have been verified in two methods:

- Manually
- Random element check each iteration

The former was done using small numbers of `n`, this was initially done to verify that the algorithm atleast works correctly.

The latter will each iteration calculate the matrix multiplication on the root process, and then pick a random element and compare it against the two matrices. This is done using and assert, which would halt the program if not held. An error tolerance of 0.001 was put in place to allow for rounding errors. This testing is done after saving the `endtime`, thus the overall runtime measurements are not affected.

Chapter 6

Conclusion