DM818 Assignment 3

DNS Algorithm

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

Christmas 2015
:-)

Table of Contents

1 Introduction **2**
1.2 Contributions 2

2 Implementation **3**
2.1 Design choices 3
2.2 Virtual topologies 3
2.3 The DNS algorithm 4
2.4 Matrix not divisible by $p^{1/3}$ 6

3 Performance **7**
3.1 Runtimes 7
3.1.1 The theoretical serial runtime 8
3.1.2 The plots 8

4 Testing **10**

5 Conclusion **11**

6 Appendix **12**
6.1 Plotting tables 12

Chapter 1

Introduction

This report documents the development of a 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. Due to time constraints the report part of the assignment was done separately whenever time could be found. The quality of the report has suffered for this.

Chapter 2

Implementation

2.1 Design choices

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.

While implementing the DNS algorithm, the design choices revolved around what type of communication calls to use.

Both of these are covered in the following sections.

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 The DNS algorithm

Implementing the DNS algortihm can be divided into a few simple steps:

1. Distribute at k'th dimension.
2. Broadcast at j'th/i'th dimension.
3. Multiply.
4. Reduce k'th dimension to $k = 0$.

Initially the matrix is blocked and distributed among processes at the $k = 0$ dimension. This distributing is done using `MPI_Scatterv` which takes a vector of data and scatters it among the specified processes. Scatter was the correct choice here as each process in the k'th dimension requires each their data (a separate block of the matrix).

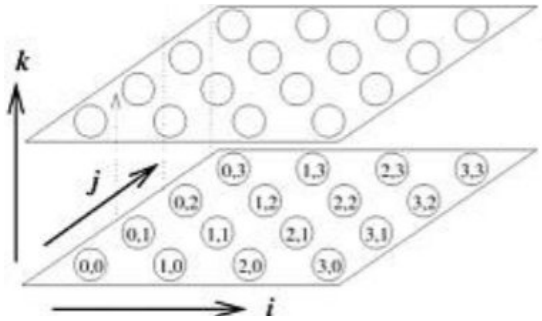

Figure 2.1: Blocking and distribution of sub-matrices at k=0.

Step 1: Distribute at k'th dimension

Following the algorithm from the book, the row `j` is send to the k'th iteration. Such that $k = j$. This is done using Send for processes at $k = 0$ and Receive for the processes at $k > 0$. Using a single send and receive was chosen as this is a one-to-one communication, ie. a single process only has to hand their data to a single process at a higher dimension.

Note that this is the same procedure for the B matrix, just replace `j` with `i`.

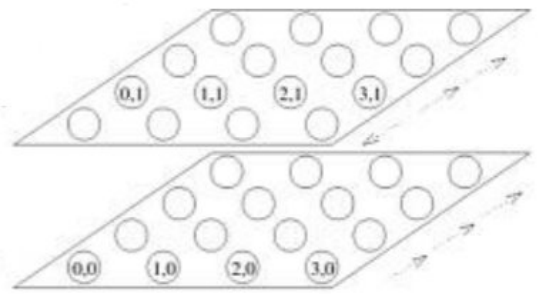

Figure 2.2: Distribution from k=0 to k=j.

Step 2: Broadcast at j'th dimension

The data given to the dimensions at $k > 0$ is then broadcasted to the processes along the j'th dimension. This is done using `MPI_Bcast`, as we have a one-to-many communication, the previous process who received data has to transmit the same data to the remaining processes along j'th dimension.

Note that this is the same for the B matrix, using i'th dimension instead of j'th.

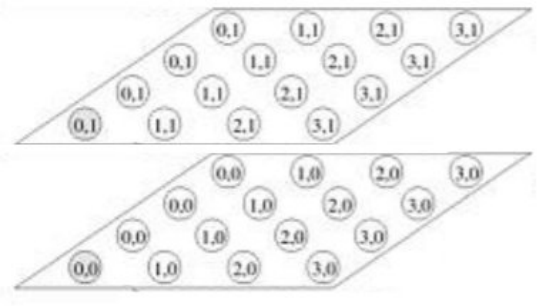

Figure 2.3: Broadcast along j'th dimension

Step 3: Multiply

Simply multiplying matrix `(i,j)` of A with matrix `(j,i)` of matrix B. This is done using the naive implementation if run on Imada, or using the BLAS library if run on hopper. The result is stored in `ResultMatrix` on each process.

Step 4: Reduce k'th dimension to $k = 0$

Reducing along the k'th dimension is done using `MPI_Reduce`, since we are doing a many-to-one communication along the k'th dimension. All processes at $k > 0$ sends their data to $k = 0$.

2.4 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:

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

Chapter 3

Performance

3.1 Runtimes

Test-runs for the program has been run with 4 different settings, using the number of processes as following:

- # Processes:= 1
- # Processes:= 125
- # Processes:= 512
- # Processes:= 1000

With the number of processes at 1, we can get a baseline of the best possible efficiency that can be achieved, given that zero overhead for the communication is present. The remaining numbers will then show how the increasing amount of overhead for communication affects different sizes of matrices.

For matrices the different sizes run are: {25, 100, 400, 800, 1000, 10000, 15000, 20000}. This should give a good indication at which sizes the overhead stops dominating the calculations, thus giving a good efficiency and speed-up.

Efficiency shows us just how good the algorithm performs, ie. how much time is spent doing calculations versus communication overhead. The bigger the number, the more time is spent doing actual calculations. The efficiency is calculated by dividing the speed-up with the number of processes used.

Speedup shows us how much faster the problem is solved. Thus any number above 1 will be a faster solution than the theoretical serial solution. The speed-up is calculated by dividing the serial runtime with the parallel runtime.

3.1.1 The theoretical serial runtime

The assignment required that we did not use the $p = 1$ for our serial comparison, but instead inferred a theoretical maximal achievable. This max is calculated using Hoppers theoretical max of 8.4Gflop/s divided by the complexity of the problem. Multiplying two n-by-n matrices takes $O(n^3$ calculations, but in order to be fair we have to include the summation of the reduce step, thus the number on calculations performed is actuall $n^3 * 2$. Thus the best possible serial solution would be:

$$\frac{n^3 * 2}{8400000000}$$

3.1.2 The plots

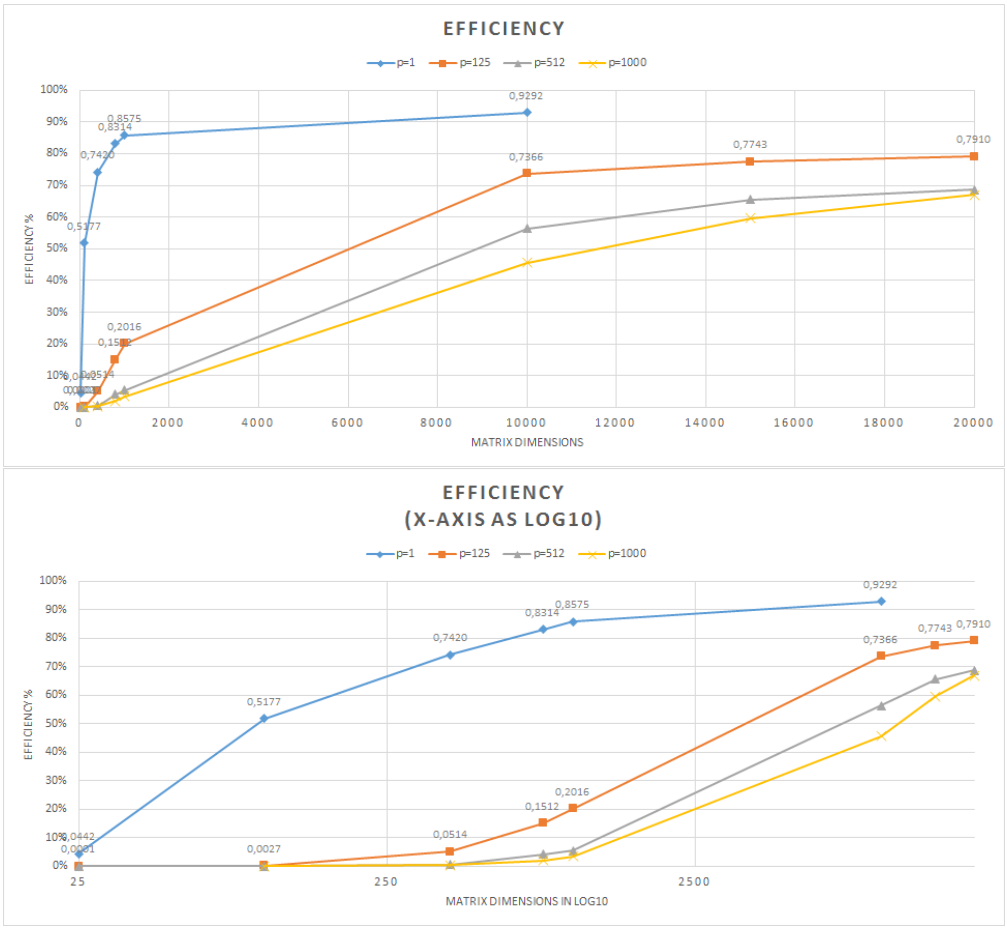

Figure 3.1: The efficiency for different values of number of processes and matrix dimensions. Table can be seen in appendix.

In figure 3.1 we can see how the efficiency is for the different sizes of the matrices. The blue line showing the max achievable amount with 1 process. As the sizes of the matrix grows the amount the overhead dominates declines rapidly, this clearly show us that using a high amount of processes is not always the best choice, but should rather be dependent on the size of the problem. As the matrix sizes get sufficiently large, the overhead will only be a fraction of the overall cost. Had the computation time been calculated, this is what we would have seen.

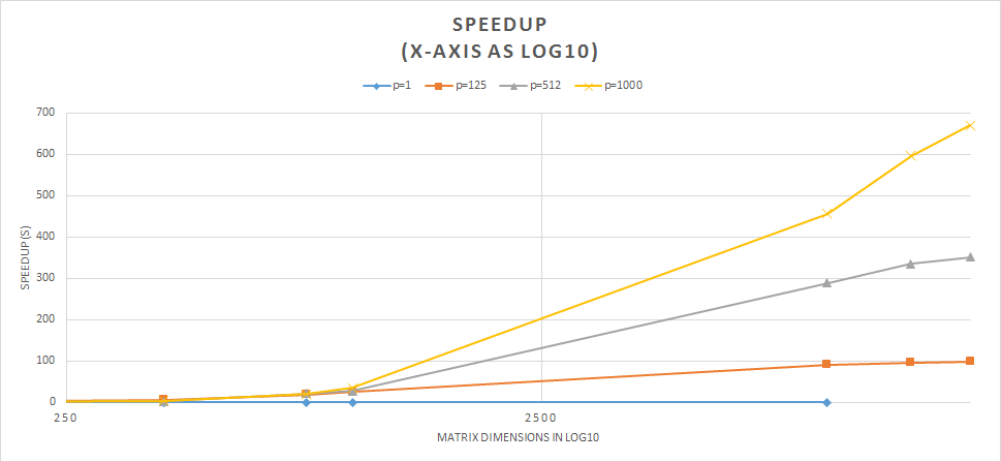

Figure 3.2: The speed-up for different values of number of processes and matrix dimensions. Table can be seen in appendix.

Efficiency only show us how well we utilize computation time versus communication, but what is even more interesting is how much faster we solve the problem. To illustrate this the speed-up is plotted for the same data-set. While the increased amount of communications makes it slower at smaller problem sizes < 400 for `n`, the increased computation power quickly increases how fast the solution is found.

Chapter 4

Testing

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

- Manually calculation
- 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 5

Conclusion

Compared to the previous assignment, less time was spent on understanding how MPI works. Instead the challenge was understanding how to build Cartesian structures. This builds nicely on-top of what we previous learned and in the end gave a fundamentally understanding how to work within a parallel environment using MPI.

The DNS algorithm itself was rather simple, the biggest issue was getting ones head around how to do the communication between the processes correctly.

A working program was somewhat quickly composed (compared to the previous assignment) after spending a fair amount of time discussing how to attack the problem correctly. Due to severe lack of time, the code has not been adhered to different coding standards, and should be re-factored if ever worked with again.

Some requirements were not done, such as plotting computation time and calculating the iso-efficiency function. This was due to time constraints and having to prioritise other assignments.

Chapter 6

Appendix

6.1 Plotting tables

Efficiency				
n	p=1	p=125	p=512	p=1000
25	0,0442	0,0001	0,0000	
100	0,5177	0,0027	0,0002	0,0001
400	0,7420	0,0514	0,0061	0,0042
800	0,8314	0,1512	0,0419	0,0203
1000	0,8575	0,2016	0,0543	0,0356
10000	0,9292	0,7366	0,5643	0,4559
15000		0,7743	0,6559	0,5968
20000		0,7910	0,6868	0,6701

Figure 6.1: Table showing the efficiency for different values.

Speedup				
n	p=1	p=125	p=512	p=1000
25	0,0442	0,0070	0,0014	
100	0,5177	0,3409	1,2558	0,1381
400	0,7420	6,4191	3,1428	4,2430
800	0,8314	18,8968	21,4270	20,2777
1000	0,8575	25,1949	27,8180	35,6001
10000	0,9292	92,0701	288,9037	455,8706
15000		96,7904	335,8288	596,7544
20000		99,1207	351,6361	670,1347

Figure 6.2: Table showing the speed-up for different values.