CS566 Parallel Processing
Assignment 03

Camillo Lugaresi and Cosmin Stroe

Department of Computer Science
University of Illinois at Chicago

November 8, 2011

1 Algorithm Details and Formulations

For this assignment we tackled two main chanllenges. The first was experimenting with several implementations of matrix multiplication algorithms. In order to do matrix multiplication we implemented Cannon's algorithm and the Dekel, Nassimi, Sahni (DNS) algorithm. The second was the optimization of our LU-decomposition algorithm by eliminating broadcast communication and using pipelined communication instead. The reason optimization of the LU-decomposition algorithm became necessary was that we noticed that it was taking significantly longer than the matrix multiplication algorithms. As part of our optimization we implemented a 1D row partitioning LU-decomposition algorithm in addition to the 2D partitioning we had implemented for Assignment 02. We will be using timing results from both of these algorithms.

In our descriptions, we always refer to the formula

$$C = A \times B$$

where A, B, and C are $n \times n$ matrices. Some of the matrix multiplication algorithms distribute the left and right matrices in a different manner and we will make the distinction between A and B, even though for our problem of computing A^k they contain identical values. Our determinant is computed by running the parallel formulation of the LU decomposition algorithm on the final matrix, A^k.

Although we considered the possibility of using the square-and-multiply algorithm to compute powers of X, we decided against it. The objective of this assignment is to evaluate algorithms for matrix multiplication, and it is more convenient to have the parameter k correspond directly to the number of full matrix multiplications $(k-1)$.

1.1 Cannon's Algorithm

Cannon's algorithm is based on a checkerboard decomposition of the output data: the processors are arranged in a 2D mesh, and each is tasked with computing an $(n/\sqrt{p} \times n/\sqrt{p})$ block of the output matrix C. To do so, each processor needs to access the corresponding row of blocks of matrix A and the corresponding column of blocks of matrix B:

$$C_{i,j} = \sum_{k=1}^{\sqrt{p}} A_{i,k} \cdot B_{k,j}$$

A naive solution would be to have each processor store the entire row and column it needs, but that would be memory-inefficient. Instead, Cannon's algorithm distributes an $(n/\sqrt{p} \times n/\sqrt{p})$ block of A and one of B to the processor owning the block of C with the same coordinates, and relies on an intelligent schedule of communications and computations to ensure each processor gets the data it needs with no waste of memory and no dead times.

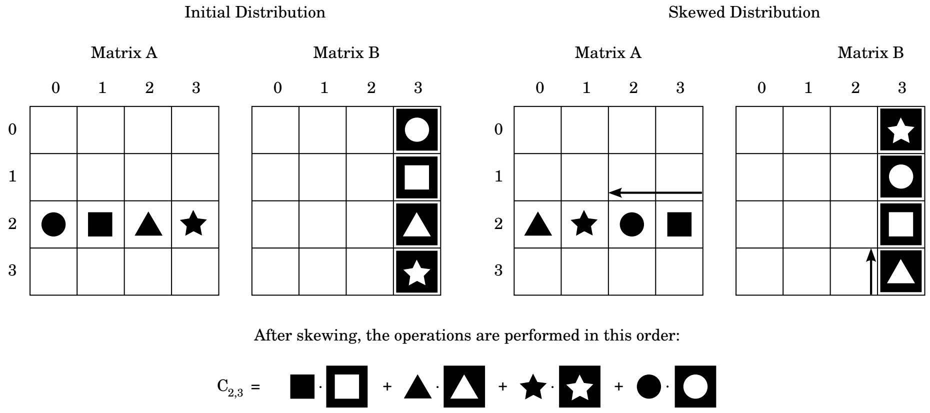

Figure 1: Explanation of the skew operation in Cannon's algorithm.

First, note that, by successively performing ring-1-shifts of the blocks of the A matrix on the row rings, and ring-1-shifts of the blocks of the B matrix on the column rings, each processor gets to access all the blocks it needs. The problem is to ensure that the blocks are lined up so that the processor simultaneously holds the blocks of A and B that must be multiplied together.

As it turns out, this condition can be ensured for all processors at the same time by performing an initial skewing of the A and B matrices (figure **??**). Each row of A is shifted left by $i-1$ steps (where i is the row number), and each column of B is shifted up by $j-1$ steps (where j is the column number). At this point, each processor holds two blocks of A and B that must be multiplied together to form one term of the sum that produces $C_{i,j}$.

Then all rows of A are shifted left by one step, and all columns of B are shifted up by one step; again, each processor ends up with a pair of elements that must be multiplied together and added to the computation of its $C_{i,j}$. These shift and multiply operations are repeated $\sqrt{p}$ times, until each processor has finished computing its block of C. Then the product matrix can be gathered.

1.1.1 Local memory usage

Each processor holds a block of the operand matrices A and B and of the result matrix C. Therefore,

$$M_p = O(n^2/p)$$

However, the root node is an exception since it holds the initial matrix and gathers the final product matrix, so its local memory usage is

$$M_p = O(n^2)$$

1.1.2 Parallel Computation Time (Local Computations)

In the analysis of matrix multiplication it is usually assumed that the cost of multiplication instructions dominates that of addition instructions; this assumption is reflected in the reality of current computer systems.

For each invocation of the algorithm, each processor performs $\sqrt{p}$ naive matrix multiplications of submatrices of size $n/\sqrt{p}$. Naive matrix multiplication performs $O(n^3)$ multiplications, so the number of operations per processor is $O(\sqrt{p}(n/\sqrt{p})^3)$. Then the computation time is:

$$\begin{aligned} T_{comp} &= t_{mult} \cdot \sqrt{p} \cdot (n/\sqrt{p})^3 \\ &= t_{mult} \cdot n^3/p \end{aligned}$$

1.1.3 Parallel Communication Time

The algorithm involves the following communication steps:

- scatter the X matrix into blocks of size $O(n^2/p)$
- skew matrix A (B); each processor performs a ring_shift operation on the row (column) ring with distance = row (column) number
- $\sqrt{p}$ times: shift matrix A (B): ring_shift with distance 1
- gather the product matrix

The message size m is always n^2/p. In general, we can assume that t_s is dominated by $t_w \cdot m$.

- Scatter on a mesh takes:

  $$T_{scatter} = t_s \log p + t_w \cdot m \cdot (p-1) = t_s \log p + t_w \cdot (n^2/p) \cdot (p-1)$$

  Gather has the same complexity as scatter.

- Ring q-shift involves $\min(q, p-q)$ neighbor-to-neighbor communications:

  $$T_{shift-q} = \min(q, p-q) \cdot (t_s + t_w \cdot m)$$

- The highest value for the skew shift will be:

  $$T_{skew} = \sqrt{p}/2 \cdot (t_s + t_w \cdot n^2/p)$$

- For all of the $\sqrt{p}$ shifts collectively, we have

  $$T_{shift} = \sqrt{p} \cdot (t_s + t_w \cdot n^2/p)$$

Then the overall communication time is:

$$\begin{aligned}
T_{comm} &= T_{scatter} + T_{skew} + T_{shift} + T_{gather} \\
&= 2 \cdot T_{scatter} + T_{skew} + T_{shift} \\
&= 2 \cdot t_s \cdot \log p + 2 \cdot t_w \cdot \frac{n^2}{p}(p-1) + \frac{\sqrt{p}}{2} \cdot (t_s + t_w \cdot n^2/p) + \sqrt{p} \cdot (t_s + t_w \cdot \frac{n^2}{p}) \\
&= 2 \cdot t_s \cdot \log p + 2 \cdot t_w \cdot \frac{n^2}{p}(p-1) + \frac{3}{2}\sqrt{p} \cdot \left(t_s + t_w \cdot \frac{n^2}{p}\right) \\
&= t_s \cdot \left(2 \log p + \frac{3}{2}\sqrt{p}\right) + t_w \cdot \left(2\frac{n^2}{p}(p-1) + \frac{n^2}{p}\right) \\
&= t_s \cdot \left(2 \log p + \frac{3}{2}\sqrt{p}\right) + t_w \cdot \frac{n^2}{p}(2(p-1)+1) \\
&= t_s \cdot \left(2 \log p + \frac{3}{2}\sqrt{p}\right) + t_w \cdot \frac{n^2}{p}(2p-1)
\end{aligned}$$

1.1.4 Parallel Run Time

$$\begin{aligned}
T_p &= T_{comp} + T_{comm} \\
&= t_{mult} \cdot \frac{n^3}{p} + t_s \cdot \left(2 \log p + \frac{3}{2}\sqrt{p}\right) + t_w \cdot \frac{n^2}{p}(2p-1)
\end{aligned}$$

1.1.5 Speedup

For the reference serial algorithm, we will use the naive multiplication algorithm here.

$$\begin{aligned}
S &= \frac{T_s}{T_p} \\
&= \frac{t_{mult} \cdot n^3}{t_{mult} \cdot \frac{n^3}{p} + t_s \cdot \left(2 \log p + \frac{3}{2}\sqrt{p}\right) + t_w \cdot \frac{n^2}{p}(2p-1)} \\
&= \frac{n^3}{\frac{n^3}{p} + \frac{t_w}{t_{mult}} \cdot \frac{n^2}{p}(2p-1) + \frac{t_s}{t_{mult}} \cdot \left(2 \log p + \frac{3}{2}\sqrt{p}\right)} \\
&= \frac{p}{1 + \frac{t_w}{t_{mult}} \cdot \frac{2p-1}{n} + \frac{t_s}{t_{mult}} \cdot \frac{p\left(2 \log p + \frac{3}{2}\sqrt{p}\right)}{n^3}}
\end{aligned}$$

If n grows much larger than p, the denominator approaches 1, and so the system approaches linear speedup.

1.1.6 Efficiency

$$\begin{aligned} E &= \frac{T_s}{pT_p} = \frac{S}{p} = \\ &= \frac{1}{1 + \frac{t_w}{t_{mult}} \cdot \frac{2p-1}{n} + \frac{t_s}{t_{mult}} \cdot \frac{p\left(2\log p + \frac{3}{2}\sqrt{p}\right)}{n^3}} \end{aligned}$$

If n grows much larger than p, the system approaches ideal efficiency.

1.1.7 Cost-optimality

Cost-optimality is attained when efficiency remains constant as n and p change. First, let us note that the impact of the startup time of communication is negligible when the message size is large; since our message size is n^2/p, this condition is satisfied. Therefore we can simplify the expression of efficiency as follows:

$$E = \frac{1}{1 + \frac{t_w}{t_{mult}} \cdot \frac{2p-1}{n}}$$

The condition for cost-optimality is then

$$\begin{aligned} O(2p-1) &= O(n) \\ O(p) &= O(n) \\ p &= \Theta(n) \end{aligned}$$

Since our workload W is n^3, the isoefficiency function can be obtained as

$$\begin{aligned} n &= \Theta(p) \\ n^3 &= \Theta(p^3) \\ W &= k \cdot p^3 \end{aligned}$$

1.2 Cannon's algorithm with local multiplication using Strassen

This is a variant of the previous algorithm, where each processor computes the local products using Strassen's multiplication algorithm instead of the naive multiplication algorithm. This increases memory usage (although Cannon's decomposition remains memory-efficient), but reduces the computation time for large matrices.

1.2.1 Local memory usage

In addition to the normal memory usage for Cannon's algorithm, we must consider the temporary storage used for Strassen. Strassen's algorithm is run locally by each processor on a square block with size $n/\sqrt{p}$. Let us assume that $n/\sqrt{p}$ is a power of 2.

Each invocation of Strassen on a matrix of size s allocates storage for $9 \cdot (s/2)^2$ elements (the 7 M matrices and two temporary matrices for computing intermediate sums and products). Then it recursively invokes Strassen 7 times on blocks of size $s/2$. Therefore its memory usage is:

$$\begin{aligned}
f(s) &= 9/4 \cdot s^2 + 7 \cdot f(s/2) \\
&= 9/4 \cdot s^2 + 7 \cdot 9/4 \cdot (s/2)^2 + 7 \cdot 7 \cdot f(s/4) \\
&= 9/4 \cdot (s^2 + 7/4 \cdot s^2) + 7 \cdot 7 \cdot f(s/4) \\
&= 9/4 \cdot (s^2 + 7/4 \cdot s^2 + (7/4)^2 \cdot s^2) + 7^3 \cdot f(s/2^3)... \\
&= s^2 \cdot 9/4 \cdot (7/4 + (7/4)^2 + \ldots) \\
&= s^2 \cdot 9/4 \cdot \sum_{i=1}^{log_2 s} (7/4)^i \\
&= s^2 \cdot 9/4 \cdot log_2 \left(2^{\sum_{i=1}^{log_2 s} (7/4)^i} \right) \\
&= s^2 \cdot 9/4 \cdot log_2 \left(\prod_{i=1}^{log_2 s} 2^{(7/4)^i} \right) \\
&= s^2 \cdot 9/4 \cdot log_2 \left(\prod_{i=1}^{log_2 s} (7/4)^{2^i} \right) \\
&= s^2 \cdot 9/4 \cdot log_2(7/4) \cdot log_{7/4} \left(\prod_{i=1}^{log_2 s} (7/4)^{2^i} \right) \\
&= s^2 \cdot 9/4 \cdot log_2(7/4) \cdot \left(\sum_{i=1}^{log_2 s} log_{7/4}(7/4)^{2^i} \right) \\
&= s^2 \cdot 9/4 \cdot log_2(7/4) \cdot \left(\sum_{i=1}^{log_2 s} 2^i \right) \\
&= s^2 \cdot 9/4 \cdot log_2(7/4) \cdot (2^{log_2 s+1} - 1) \\
&= s^2 \cdot 9/4 \cdot log_2(7/4) \cdot (2s - 1) \\
&= s^2 \cdot (2s - 1) \cdot 9/4 \cdot log_2(7/4) \\
&= (2s^3 - s^2) \cdot 9/4 \cdot log_2(7/4)
\end{aligned}$$

Hence

$$f(s) = O(s^3)$$

$$f(n/\sqrt{p}) = O\left(\frac{n^3}{p^{3/2}}\right)$$

Thus the space complexity becomes:

$$Mp = O\left(\frac{n^3}{p^{3/2}} + \frac{n^2}{p}\right) = O\left(\frac{n^2}{p} \cdot \left(\frac{n}{\sqrt{p}} + 1\right)\right) = O\left(\frac{n^3}{p^{3/2}}\right)$$

For the root node, it becomes:

$$Mp = O\left(\frac{n^3}{p^{3/2}} + n^2\right) = O\left(n^2 \cdot \left(\frac{n}{p^{3/2}} + 1\right)\right) = O\left(\frac{n^3}{p^{3/2}}\right)$$

1.2.2 Parallel computation time

For each invocation of the algorithm, each processor performs $\sqrt{p}$ Strassen matrix multiplications of submatrices of size $n/\sqrt{p}$. Strassen matrix multiplication performs $\approx n^{2.807}$ multiplications, so the number of operations per processor is $\sqrt{p} \cdot (n/\sqrt{p})^{2.807}$. Then the computation time is:

$$\begin{aligned} T_{comp} &= t_{mult} \cdot \sqrt{p} \cdot (n/\sqrt{p})^{2.807} \\ &= t_{mult} \cdot n^{2.807} \cdot \sqrt{p} \cdot p^{-2.807/2} \\ &= t_{mult} \cdot n^{2.807} \cdot p^{(1-2.807)/2} \\ &= t_{mult} \cdot n^{2.807} \cdot p^{-0.9035} \end{aligned}$$

1.2.3 Parallel communication time

This is unchanged from the regular Cannon's algorithm:

$$\begin{aligned} T_{comm} &= T_{scatter} + T_{skew} + T_{shift} + T_{gather} \\ &= t_s \cdot \left(2\log p + \frac{3}{2}\sqrt{p}\right) + t_w \cdot \frac{n^2}{p}(2p-1) \end{aligned}$$

1.2.4 Parallel run-time

$$\begin{aligned} T_p &= T_{comp} + T_{comm} \\ &= t_{mult} \cdot n^{2.807} \cdot p^{-0.9035} + t_s \cdot \left(2\log p + \frac{3}{2}\sqrt{p}\right) + t_w \cdot \frac{n^2}{p}(2p-1) \end{aligned}$$

1.2.5 Speedup

In this case, we will use Strassen as the reference serial algorithm.

$$\begin{aligned}
S &= \frac{T_s}{T_p} \\
&= \frac{t_{mult} \cdot n^{2.807}}{t_{mult} \cdot n^{2.807} \cdot p^{-0.9035} + t_s \cdot \left(2\log p + \frac{3}{2}\sqrt{p}\right) + t_w \cdot \frac{n^2}{p}(2p-1)} \\
&= \frac{n^{2.807}}{n^{2.807} \cdot p^{-0.9035} + \frac{t_s}{t_{mult}} \cdot \left(2\log p + \frac{3}{2}\sqrt{p}\right) + \frac{t_w}{t_{mult}} \cdot \frac{n^2}{p}(2p-1)} \\
&= \frac{1}{p^{-0.9035} + \frac{t_w}{t_{mult}} \cdot \frac{n^2}{pn^{2.807}}(2p-1) + \frac{t_s}{t_{mult}} \cdot \frac{\left(2\log p+\frac{3}{2}\sqrt{p}\right)}{n^{2.807}}} \\
&= \frac{p^{0.9035}}{1 + \frac{t_w}{t_{mult}} \cdot \frac{p^{0.9035}n^2}{pn^{2.807}}(2p-1) + \frac{t_s}{t_{mult}} \cdot \frac{p^{0.9035}\left(2\log p+\frac{3}{2}\sqrt{p}\right)}{n^{2.807}}} \\
&= \frac{p^{0.9035}}{1 + \frac{t_w}{t_{mult}} \cdot \frac{2p-1}{p^{0.0965}n^{0.807}} + \frac{t_s}{t_{mult}} \cdot \frac{p^{0.9035}\left(2\log p+\frac{3}{2}\sqrt{p}\right)}{n^{2.807}}}
\end{aligned}$$

It may be interesting to compare this with the expression we obtained for the speedup with the standard Cannon's algorithm:

$$S_{cannon} = \frac{p}{1 + \frac{t_w}{t_{mult}} \cdot \frac{2p-1}{n} + \frac{t_s}{t_{mult}} \cdot \frac{p\left(2\log p+\frac{3}{2}\sqrt{p}\right)}{n^3}}$$

1.2.6 Efficiency

$$\begin{aligned}
E &= \frac{T_s}{pT_p} = \frac{S}{p} = \\
&= \frac{p^{-0.0965}}{1 + \frac{t_w}{t_{mult}} \cdot \frac{2p-1}{p^{0.0965}n^{0.807}} + \frac{t_s}{t_{mult}} \cdot \frac{p^{0.9035}\left(2\log p+\frac{3}{2}\sqrt{p}\right)}{n^{2.807}}}
\end{aligned}$$

Again, noting that our message size is large, we can simplify the expression as

$$\begin{aligned}
E &= \frac{p^{-0.0965}}{1 + \frac{t_w}{t_{mult}} \cdot \frac{2p-1}{p^{0.0965}n^{0.807}}} \\
&= \frac{1}{p^{0.0965} + \frac{t_w}{t_{mult}} \cdot \frac{2p-1}{n^{0.807}}}
\end{aligned}$$

1.2.7 Cost-optimality

The condition for cost-optimality is

$$\begin{aligned}
p^{0.0965} + \frac{t_w}{t_{mult}} \cdot \frac{2p-1}{n^{0.807}} &= O(1) \\
p^{0.0965} + \frac{p}{n^{0.807}} &= O(1) \\
\frac{p}{n^{0.807}} &= \Theta(p^{0.0965}) \\
\frac{p}{p^{0.0965}} &= \Theta(n^{0.807}) \\
p^{0.9035} &= \Theta(n^{0.807})
\end{aligned}$$

Since our workload W is n^3, the isoefficiency function can be obtained as

$$\begin{aligned}
n^{0.807} &= \Theta(p^{0.9035}) \\
n^3 &= \Theta(p^{3 \cdot 0.9035/0.807}) \\
W &= k \cdot p^{3.3587}
\end{aligned}$$

1.3 Dekel, Nassimi, Sahni (DNS) Algorithm

The DNS algorithm is based on decomposing the intermediate data of the matrix multiplication algorithm and we chose it because we wanted to explore the effect of this data decomposition method on our running time. One of the main differences from Cannon's algorithm is that it uses a 3D mesh topology of $q \times q \times q$ processors, and requires the total number of processors to be a perfect cube (i.e., $p = q^3$). Each processor $P_{i,j,k}$ in the 3D mesh computes the multiplication of the $A_{i,k}$ and $B_{k,j}$ elements. In order to make DNS algorithm cost efficient for $p < n$, each processor receives a submatrix of the A and B matrices, each of size $(n/q) \times (n/p)$, which it then multiplies using a serial matrix multiplication algorithm.

The results are then reduced onto the $i - k$ plane and the final C matrix is gathered at the root processor (rank = 0);

1.3.1 Local memory usage

Since a cost efficient formulation of the DNS algorithm uses $p < n$, each processor gets an $(n/q \times n/q)$ submatrix, where $q = \sqrt[3]{p}$, for each matrix A and B. It also stores the output matrix C. Therefore the local storage requirements for the DNS algorithm is

$$3 \cdot \left(\frac{n}{\sqrt[3]{p}}\right)^2 = O\left(\frac{n^2}{p^{2/3}}\right)$$

```
Algorithm 1.1: DNS_Matrix_Multiplication($A, B$)

Create_3D_Mesh()

$Ablock \leftarrow$ MPI_Scatter_on_I-K($A$)
$Asub \leftarrow$ MPI_Broadcast_on_J($Ablock$)

$Bblock \leftarrow$ MPI_Scatter_on_K-J($B$)
$Bsub \leftarrow$ MPI_Broadcast_on_I($Bblock$)

$Csub \leftarrow$ SerialMatrixMultiply($Asub, Bsub$)

$Cred \leftarrow$ MPI_Reduce_on_K($Csub$)
$C \leftarrow$ MPI_Gather_on_I-J($Cred$)

return ($C$)
```

Figure 2: The DNS algorithm with the MPI calls used.

at each processor.

1.3.2 Parallel Computation Time (Local Computations)

Each processor has to serial multiply its submatrices, which are square matrices with dimension $n/\sqrt[3]{p}$, therefore the overall time spent for local computations is

$$T_{comp} = \left(\frac{n}{\sqrt[3]{p}}\right)^3 = \frac{n^3}{p}$$

1.3.3 Parallel Communication Time

The parallel communication time comes from the following operations:

- Two scatter operations for matrices A and B on a 2D mesh, with each mesh dimension having $q = \sqrt[3]{p}$ processors, and the message size $m = (n/q)^2 = n^2/p^{2/3}$.

$$\begin{aligned} T_{scatter,mesh} &= 2\left[t_s \log q + t_w m(q-1)\right] \\ &= 2\left[t_s \log \sqrt[3]{p} + t_w \left(\frac{n}{\sqrt[3]{p}}\right)^2 (\sqrt[3]{p}-1)\right] \\ &= 2\left[\frac{1}{3} t_s \log p + t_w \frac{n^2}{\sqrt[3]{p}} - t_w \frac{n^2}{\sqrt[3]{p^2}}\right] \end{aligned}$$

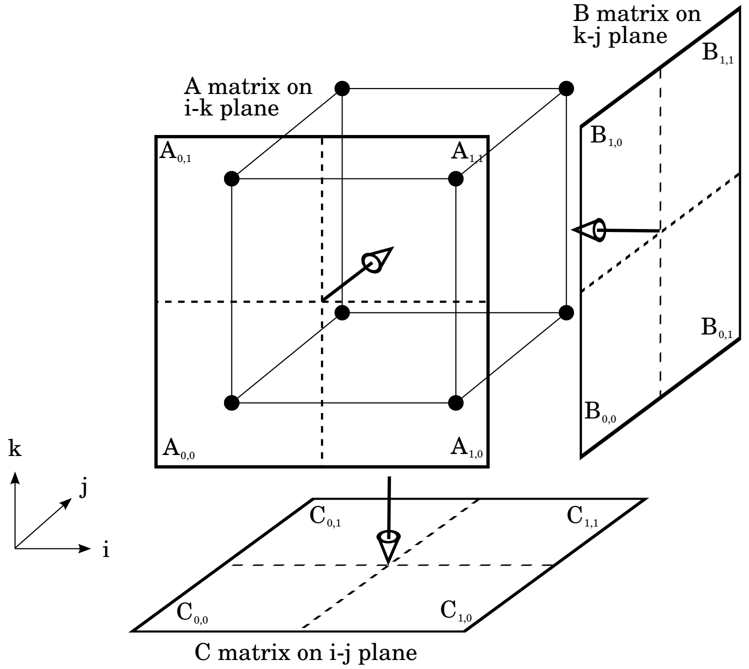

Figure 3: An overview of the DNS algorithm. The A, B and C matrix planes show the scatter/gather operations while the orthogonal arrows show the broadcast/reduce operations.

- Two one-to-all broadcast on a ring with $q = \sqrt[3]{p}$ processors and message size $m = (n/q)^2 = n^2/p^{2/3}$.

$$\begin{aligned} T_{broadcast,ring} &= 2(t_s + t_w \cdot m)\log q \\ &= \frac{2}{3}\left(t_s + t_w \frac{n^2}{p^{2/3}}\right)\log p \\ &= \frac{2}{3}\left(t_s \log p + t_w \log p \frac{n^2}{p^{2/3}}\right) \end{aligned}$$

- Reduce on a ring with $q = \sqrt[3]{p}$ processors. This time is identical to the one-to-all broadcast.

- Gather the final matrix, on a 2D mesh with each dimension having $q = \sqrt[3]{p}$ processors. This time is identical to the time it takes to do one scatter operation, and can be easily derived from the scatter formula above.

Therefore our total communication time is:

$$T_{comm} = T_{scatter,gather} + T_{broadcast,reduce}$$

$$\begin{aligned} &= 3\left(\frac{1}{3}t_s\log p + t_w\frac{n^2}{\sqrt[3]{p}} - t_w\frac{n^2}{\sqrt[3]{p^2}}\right) + \frac{3}{3}\left(t_s\log p + t_w\log p\frac{n^2}{p^{2/3}}\right) \\ &= 2t_s\log p + 3t_w\left(\frac{n^2}{p^{1/3}}\right) + t_w\frac{n^2}{p^{2/3}}(\log p - 3) \end{aligned}$$

1.3.4 Parallel Run Time

Using the formula $T_p = T_{comp} + T_{comm}$ we get that

$$T_p = \frac{n^3}{p} + 2t_s\log p + 3t_w\left(\frac{n^2}{p^{1/3}}\right) + t_w\frac{n^2}{p^{2/3}}(\log p - 3)$$

1.3.5 Speedup

$$\begin{aligned} S &= \frac{T_s}{T_p} \\ &= \frac{n^3}{\frac{n^3}{p} + 2t_s\log p + 3t_w\left(\frac{n^2}{p^{1/3}}\right) + t_w\frac{n^2}{p^{2/3}}(\log p - 3)} \\ &= \frac{1}{\frac{1}{p} + 2t_s\frac{\log p}{n^3} + \frac{3t_w}{p^{1/3}n} + \frac{t_w(\log p-3)}{p^{2/3}n}} \\ &= \frac{p}{1 + 2t_s\frac{p\log p}{n^3} + 3t_w\frac{p^{2/3}}{n} + t_w\frac{p^{1/3}(\log p-3)}{n}} \end{aligned}$$

It can be seen from the equation that as the matrix increases in size, the speedup tends to become linear.

1.3.6 Cost

$$\begin{aligned} C &= pT_p \\ &= n^3 + 2t_sp\log p + 3t_wp^{2/3}n^2 + t_wp^{1/3}n^2(\log p - 3) \end{aligned}$$

1.3.7 Efficiency

$$\begin{aligned} E &= \frac{T_s}{pT_p} = \frac{S}{p} \\ &= \frac{1}{1 + 2t_s\frac{p\log p}{n^3} + 3t_w\frac{p^{2/3}}{n} + t_w\frac{p^{1/3}(\log p-3)}{n}} \end{aligned}$$

1.3.8 Cost optimality and Isoefficiency function

In order for our algorithm to be cost optimal with regards to the serial formulation, the efficiency must remain constant as the number of processors and the size of the matrices being multiplied increase. In our case the efficiency is:

$$E \quad = \quad \frac{1}{1 + 2t_s \frac{p \log p}{n^3} + 3t_w \frac{p^{2/3}}{n} + t_w \frac{p^{1/3}(\log p - 3)}{n}}$$

and asymptotically the $p^{2/3}/n$ term dominates all the other terms in the denominator, and therefore, in order for the algorithm to be cost optimal

$$O(n) = O(p^{2/3})$$

For our problem of matrix multiplication, $W = n^3$, so therefore the isoefficiency function becomes:

$$W = O(p^2)$$

1.4 LU Decomposition using 2D partitioning

The naive algorithm for computing the determinant seems quite parallelizable, but its factorial time complexity is unattractive. Instead, we decided to perform LU factorization in order to compute the determinant quickly.

From assignment 2, we already had an implementation of LU decomposition using 2D partitioning of the matrix. We improved on it by getting rid of all broadcast calls and replacing them with pipelined communications. This brought a significant performance improvement. However, due to the significant likelihood of zeros in the matrix, we have to perform at least partial pivoting, and this puts a hamper on pipelining, because it forces us to perform an all-reduce operation along a column to choose the pivot element for each iteration of the algorithm.

This means that, once a pivot has been chosen, the elimination phase proceeds in a pipelined fashion: each processor receives the pivot, computes (or receives) the m values for the cells on the pivot column, sends them to the processor to its right, receives the a value for each cell on the pivot row, forwards the value downwards and performs elimination: the elimination front proceeds diagonally down and to the right, and most computations are performed behind the front. However, before the computations for the next iteration can begin, a new pivot needs to be chosen for the next row, which forces all processors to wait until the elimination front has traversed all processors.

Another drawback of this algorithm is that it requires that the number of processors be a perfect square, which makes it difficult to use with the DNS matrix multiplication algorithm, which requires a number of processors which is a cube. The minimum number of processors that satisfied both conditions (apart from the trivial 1, of course) is $2^6 = 64$, but we cannot get access to so many processors on Argo.

1.5 LU Decomposition using 1D partitioning

To avoid the drawbacks with the 2D partitioned version of LU decomposition, we implemented a new formulation of the LU algorithm, using 1D partitioning over rows. The processors are arranged in a ring, and each processor holds n/p rows of the matrix. To improve load balancing, the rows are not divided into blocks, but rather they are assigned to each processor in a cyclic way: processor k gets the rows whose row number $i = k(modp)$ (here both i and k are zero-based indices).

Since the pivot choice is performed by a single processor, there are no obstacles to pipelining: once the first pivot has been chosen and send down, the processor holding the next row (which is the next processor down on the ring) can immediately perform elimination, choose the pivot for the next row, and send it down. The third processor receives the second pivot row immediately after the first, and so on. The computation proceeds downwards like an unbroken wave, with computation overlapping communication.

This new version of the algorithm proved much faster than the 2D one: after running a special test to compare the two, we used the 1D decomposition for all other tests.

2 Parameter Ranges

Each of the programs was tested with $n \times n$ matrix input sizes of

$$n = \{16, 64, 256, 512, 1024, 2048, 4096\}$$

with powers of

$$k = \{2, 4, 8, 16\}$$

and with processors p and cores c ranging in

$$(p, c) = \{(1, 4), (2, 2), (4, 1), (4, 4), (8, 2)\}$$

Note that the DNS algorithm can only be run with a perfect cube number of processes, and on ARGO that limits us to 8 processes, as we did not want to make use of virtual processors.

3 Results

Tables **??**, **??**, **??** and **??** show the total runtime for the Cannon, Cannon/Strassen, DNS and DNS/Strassen formulations under a variety of values of p (processing node count), c (core count per node), n (matrix side) and k (power). Tables **??**, **??**, **??**, **??** show the timings for matrix multiplication alone, while **??**, **??**, **??**, **??**, **??** show the timings for the LU decomposition and determinant calculation.

All of the timings above use the 1D version of LU. We chose it after developing and evaluating two different formulations: the comparison between them is provided in tables **??** and **??**.

n	k \ p,c	(1,1)	(1,4)	(2,2)	(4,1)	(4,4)	(8,2)
16	2	0.000	0.126	0.128	0.090	0.140	3.136
	4	0.000	0.124	0.131	0.090	0.262	0.221
	8	0.001	0.126	0.130	0.093	0.185	0.187
	16	0.001	0.128	0.133	0.137	0.194	0.204
64	2	0.005	0.037	0.082	0.073	0.148	1.629
	4	0.011	0.055	0.101	0.104	0.237	0.122
	8	0.022	0.062	0.104	0.114	3.120	0.258
	16	0.043	0.069	0.115	0.091	0.134	0.141
256	2	0.332	0.215	0.247	0.203	0.207	0.196
	4	1.135	0.357	0.269	0.278	0.415	0.516
	8	1.804	0.687	0.618	0.648	2.561	2.692
	16	3.223	1.343	1.221	1.341	1.362	3.282
512	2	10.629	1.476	1.514	1.263	0.734	0.973
	4	29.117	3.723	2.254	3.588	1.079	1.840
	8	67.642	7.645	5.010	4.736	4.092	2.876
	16	150.848	16.205	11.302	14.848	3.827	5.708
1024	2	90.693	57.894	25.894	23.457	5.652	4.704
	4	251.303	163.092	80.998	62.297	13.904	10.239
	8	596.544	384.947	186.369	142.027	20.270	19.932
	16	1227.563	788.244	397.010	312.980	48.695	28.922
4096	2	6682.291	3728.372	2304.863	1682.259	-	-
	4	-	10705.260	6674.833	4738.603	-	-
	8	-	-	-	-	-	-
	16	-	-	-	-	-	-

Table 1: Total runtime (in seconds) for the Cannon formulation.

n	k \ p,c	(1,1)	(1,4)	(2,2)	(4,1)	(4,4)	(8,2)
64	2	0.036	0.068	0.108	0.071	0.118	0.200
	4	0.107	0.104	0.127	0.093	0.169	0.115
	8	0.245	0.179	0.165	0.178	0.243	0.210
	16	0.505	0.324	0.251	0.226	0.131	0.186
256	2	1.762	0.983	0.611	0.649	0.171	0.220
	4	5.115	2.814	1.579	1.595	0.328	0.261
	8	11.719	6.611	3.776	3.594	1.609	0.831
	16	24.770	13.472	7.360	8.563	1.625	1.714
512	2	12.632	6.574	4.126	3.993	0.766	3.669
	4	35.885	19.064	11.312	11.025	1.108	-
	8	82.428	43.501	24.208	24.836	2.145	-
	16	175.750	96.441	51.518	58.017	4.226	-
1024	2	89.222	48.935	26.795	26.862	7.889	-
	4	250.350	133.086	75.469	121.289	10.205	-
	8	577.792	319.440	172.052	171.236	-	-
	16	1228.597	663.503	357.386	370.693	-	-
4096	2	4584.392	2341.838	1285.277	1334.404	-	-
	4	12354.114	6784.850	3575.100	3855.360	-	-
	8	-	-	8386.903	-	-	-
	16	-	-	-	-	-	-

Table 2: Total runtime (in seconds) for the Cannon/Strassen formulation.

n \ k	p,c	(2,4)	(4,2)	(8,1)
16	2	0.044	0.085	0.085
	4	0.045	0.085	0.044
	8	0.046	0.088	0.063
	16	0.060	0.070	0.047
256	2	0.276	0.607	1.075
	4	0.509	1.576	3.014
	8	1.302	3.491	5.396
	16	2.359	6.367	10.376
512	2	1.337	1.765	3.145
	4	3.906	3.219	8.211
	8	7.226	6.282	17.304
	16	14.980	14.573	39.256
1024	2	23.208	16.034	16.589
	4	60.548	40.500	44.334
	8	136.412	86.463	89.575
	16	275.903	181.148	186.428
4096	2	1383.476	810.533	-
	4	-	-	-
	8	-	-	-
	16	-	-	-

Table 3: Total runtime (in seconds) for the DNS formulation.

	p,c	(2,4)	(4,2)	(8,1)
n	k			
16	2	0.082	0.084	0.123
	4	0.083	0.044	0.083
	8	0.111	0.046	0.085
	16	0.086	0.087	0.047
256	2	0.457	1.951	1.252
	4	0.611	4.628	2.558
	8	2.775	6.744	7.664
	16	3.897	7.251	12.589
512	2	2.302	2.832	2.757
	4	3.879	6.812	7.645
	8	10.871	13.751	14.836
	16	17.844	25.302	33.379
1024	2	14.736	14.632	15.081
	4	33.429	40.979	44.819
	8	62.281	89.156	89.668
	16	160.716	166.998	-
4096	2	-	1385.929	-
	4	-	-	-
	8	-	-	-
	16	-	-	-

Table 4: Total runtime (in seconds) for the DNS/Strassen formulation.

	p,c	(1,1)	(1,4)	(2,2)	(4,1)	(4,4)	(8,2)
n	k						
16	2	0.000	0.000	0.001	0.001	0.043	0.045
	4	0.000	0.001	0.002	0.002	0.090	0.123
	8	0.000	0.002	0.003	0.004	0.090	0.090
	16	0.001	0.003	0.006	0.008	0.099	0.104
64	2	0.003	0.001	0.002	0.002	0.004	0.004
	4	0.008	0.004	0.006	0.007	0.008	0.009
	8	0.019	0.012	0.010	0.016	0.014	0.022
	16	0.041	0.020	0.021	0.033	0.024	0.029
256	2	0.221	0.083	0.054	0.056	0.030	0.029
	4	1.020	0.247	0.149	0.167	0.219	0.294
	8	1.691	0.573	0.489	0.399	0.762	0.927
	16	3.110	1.230	1.091	1.189	1.125	1.502
512	2	9.689	0.928	1.080	0.772	0.312	0.276
	4	28.183	3.172	1.864	2.970	0.715	1.279
	8	66.711	7.095	4.621	4.260	2.056	2.479
	16	149.922	15.641	10.872	14.347	3.478	5.297
1024	2	83.273	53.562	23.329	20.198	2.983	2.831
	4	243.889	158.678	78.302	59.585	12.290	5.662
	8	588.994	380.529	183.592	139.308	15.574	18.707
	16	1220.120	783.711	394.317	310.211	43.790	27.425
4096	2	6210.243	3477.202	2171.188	1556.964	-	-
	4	-	10453.918	6541.318	4613.076	-	-
	8	-	-	-	-	-	-
	16	-	-	-	-	-	-

Table 5: Multiplication time (in seconds) for the Cannon formulation.

n	p,c k	(1,1)	(1,4)	(2,2)	(4,1)	(4,4)	(8,2)
64	2	0.034	0.017	0.013	0.012	0.009	0.006
	4	0.105	0.055	0.033	0.035	0.023	0.008
	8	0.243	0.130	0.072	0.079	0.054	0.015
	16	0.503	0.274	0.157	0.169	0.024	0.031
256	2	1.652	0.871	0.491	0.502	0.028	0.029
	4	5.005	2.703	1.453	1.443	0.074	0.097
	8	11.605	6.495	3.637	3.446	1.138	0.606
	16	24.658	13.356	7.237	8.407	1.237	1.536
512	2	11.700	6.016	3.677	3.483	0.381	0.316
	4	34.958	18.518	10.872	10.526	0.714	-
	8	81.496	42.953	23.721	24.360	1.742	-
	16	174.824	95.898	51.082	57.544	3.793	-
1024	2	81.757	44.546	24.230	24.184	6.080	-
	4	242.965	128.713	72.928	118.600	8.581	-
	8	570.359	315.016	169.502	168.558	-	-
	16	1221.206	659.062	354.858	368.032	-	-
4096	2	4113.942	2085.596	1150.396	1202.956	-	-
	4	11881.555	6531.955	3439.557	3724.244	-	-
	8	-	-	8250.449	-	-	-
	16	-	-	-	-	-	-

Table 6: Multiplication time (in seconds) for the Cannon/Strassen formulation.

	p,c	(2,4)	(4,2)	(8,1)
n	k			
16	2	0.001	0.001	0.000
	4	0.001	0.001	0.001
	8	0.002	0.003	0.021
	16	0.016	0.006	0.004
256	2	0.053	0.478	0.722
	4	0.387	1.221	2.612
	8	1.080	2.020	5.260
	16	2.215	4.675	9.844
512	2	0.826	1.229	2.175
	4	3.392	2.713	6.461
	8	6.760	5.958	14.536
	16	14.371	14.021	35.491
1024	2	19.811	13.719	12.432
	4	56.984	38.082	40.038
	8	133.075	83.677	86.387
	16	272.513	178.654	183.061
4096	2	1218.329	714.326	-
	4	-	-	-
	8	-	-	-
	16	-	-	-

Table 7: Multiplication time (in seconds) for the DNS formulation.

n	p,c k	(2,4)	(4,2)	(8,1)
16	2	0.001	0.001	0.000
	4	0.001	0.001	0.001
	8	0.028	0.002	0.002
	16	0.004	0.005	0.004
256	2	0.299	1.767	0.937
	4	0.398	1.728	2.219
	8	2.219	3.707	7.193
	16	3.731	6.825	12.065
512	2	1.250	2.453	1.881
	4	3.164	5.914	6.853
	8	10.189	12.886	13.971
	16	17.194	24.684	30.270
1024	2	10.337	11.698	12.025
	4	29.715	36.253	41.571
	8	58.128	84.742	86.160
	16	156.947	164.226	-
4096	2	-	1217.130	-
	4	-	-	-
	8	-	-	-
	16	-	-	-

Table 8: Multiplication time (in seconds) for the DNS/Strassen formulation.

p,c n	(1,1)	(1,4)	(2,2)	(4,1)	(4,4)	(8,2)
16	0.000	0.122	0.124	0.095	0.086	0.085
64	0.002	0.040	0.084	0.073	0.134	0.130
256	0.104	0.108	0.117	0.122	0.167	0.147
512	0.899	0.510	0.359	0.445	0.271	0.329
1024	7.314	4.241	2.470	2.652	1.410	1.235
4096	469.883	248.763	130.735	121.783	-	-

Table 9: LU decomposition time (in seconds) for the Cannon formulation.

p,c n	(1,1)	(1,4)	(2,2)	(4,1)	(4,4)	(8,2)
64	0.002	0.046	0.090	0.063	0.125	0.150
256	0.102	0.101	0.111	0.132	0.182	0.154
512	0.894	0.506	0.396	0.422	0.320	0.267
1024	7.280	4.246	2.356	2.446	1.384	-
4096	469.322	252.090	132.757	127.514	-	-

Table 10: LU decomposition time (in seconds) for the Cannon/Strassen formulation.

p,c n	(2,4)	(4,2)	(8,1)
16	0.043	0.079	0.053
256	0.172	0.907	0.352
512	0.497	0.461	2.298
1024	3.314	2.435	3.689
4096	163.098	95.187	-

Table 11: LU decomposition time (in seconds) for the DNS formulation.

p,c n	(2,4)	(4,2)	(8,1)
16	0.082	0.063	0.082
256	0.269	1.633	0.408
512	0.728	0.736	1.200
1024	3.874	3.925	3.250
4096	-	166.761	-

Table 12: LU decomposition time (in seconds) for the DNS/Strassen formulation.

	p,c	(1,1)		(1,4)		(2,2)		(4,4)		(8,2)	
n	k	1d	2d	1d	2d	1d	2d	1d	2d	1d	2d
64	2	0.005	-	0.037	0.779	0.082	-	0.148	0.403	1.629	0.335
256	2	0.332	-	0.215	-	0.247	9.697	0.207	-	0.196	-
512	2	10.629	16.583	1.476	-	1.514	-	0.734	-	0.973	-
1024	2	90.693	133.654	57.894	-	25.894	-	5.652	-	4.704	-

Table 13: Total runtime (in seconds) for the Cannon formulation with 1D and 2D LU decomposition.

	p,c	(1,1)		(1,4)		(2,2)		(4,4)		(8,2)	
n	k	1d	2d	1d	2d	1d	2d	1d	2d	1d	2d
64	2	0.002	-	0.033	0.775	0.077	-	0.134	0.392	0.114	0.319
256	2	0.102	-	0.120	-	0.126	9.624	0.150	-	0.135	-
512	2	0.905	6.830	0.506	-	0.378	-	0.277	-	0.408	-
1024	2	7.282	52.065	4.168	-	2.379	-	1.421	-	1.368	-

Table 14: LU decomposition time (in seconds) for the Cannon formulation with 1D and 2D LU decomposition.

4 Analysis and Lessons

We applied Cannon's parallelization to two different matrix multiplication algorithms: the naive one, which is $O(n^3)$, and Strassen's, which is $O(n^{2.807})$. The communication overhead is exactly the same, but Strassen's computation time is $t_{mult} \cdot n^{2.807} \cdot p^{-0.9035}$, compared to $t_{mult} \cdot n^3/p$: therefore, the version using Strassen's has a better runtime. However, its isoefficiency function is worse: $k \cdot p^{3.3587}$ compared to $k \cdot p^3$.

Comparing Cannon's algorithm with DNS, Cannon's has better local memory usage than DNS, however, we can see from the isoefficiency function that DNS is more scalable. So we are trading off memory usage for problem scalability.

The results show that Strassen's algorithm can be slightly slower for small matrices, but becomes advantageous for large matrices, as can be seen for matrix of size 4096 in Figure **??**. We could obtain the best of both worlds by falling back to the naive algorithms when Strassen is called for small matrices; this also occurs at the deepest levels of recursion when running Strassen on a large matrix, so it may bring a small improvement in that case as well.

It is difficult to compare Cannon and DNS from the result tables, since the former algorithm requires a number of processors that is a square number, while the latter requires a cube. The smallest number of processors that would satisfy both is $2^6 = 64$, but we do not have that many processors available on Argo. However, by interpolating the measurements, DNS does not appear to be faster than Cannon's. This becomes evident in the graph in Figure **??**.

Our version of LU decomposition from assignment 2 required a square grid, and thus was not directly compatible with DNS. We implemented a new version with 1D decomposition, which is we made much faster by implementing pipelining communication with `MPI_SendRecv` instead of `MPI_Bcast` operations.

5 Lessons

We found that it can be very beneficial to modularize the tasks performed by a program, and to separate the parallel and serial aspects. For this assignment we had two different parallel decompositions of matrix multiplication (Cannon's and DNS), two different serial formulations (naive and Strasser's), and two different formulations of the LU decomposition task (1D and 2D), and we were able to mix and match them freely and determine which was the best combination. It turned out that creating a hybrid of two different approaches (Cannon's and Strasser's) can produce much better results than taking each separately.

For LU factorization, we saw that the importance of communication and pipelining cannot be overestimated. In this part of the program, profiling was very important to determine where time was being spent; although we removed these calls from the final program for the sake of clarity, we performed a poor man's profiling by accumulating the time spent in several key spots (for instance, in each MPI call in LU-2D). This allowed us to gauge the true impact

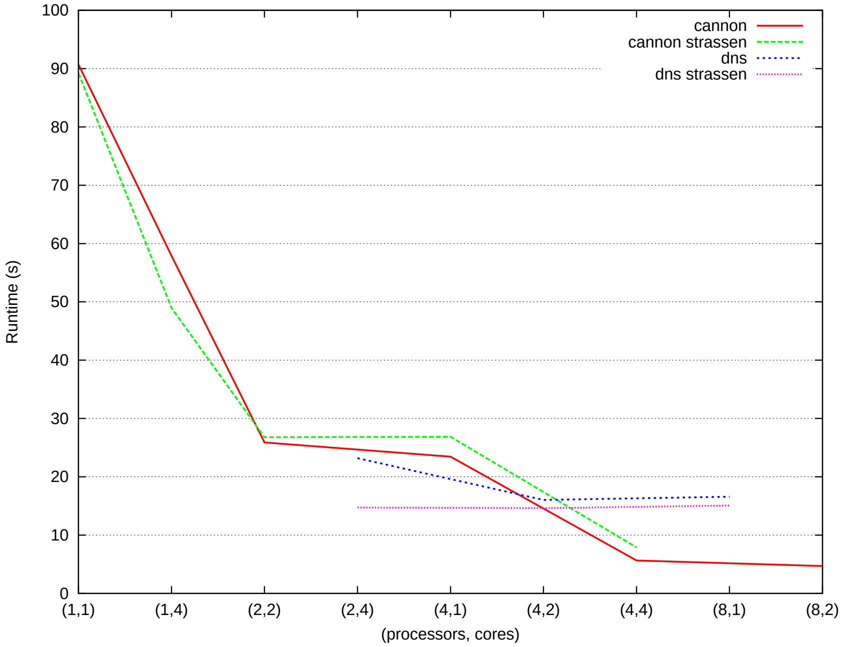

Figure 4: Runtimes of Cannon's and DNS algorithms with naive and Strassen's serial matrix multiplication algorithms for $n = 1204$ and $k = 2$.

of our use of MPI broadcasting primitives, and led first to a much improved implementation of LU-2D using pipelining, and then to its replacement with a different formulation that could make even better use of pipelining.

We also improved our scripts for running experiments and collecting data, allowing us to automate the generation of not only tables, but also of graphs. It is definitely work it to put in time to automate these tasks up front, as it allowed us easily to keep refining our results as new test batches were run.

On the other hand, we realized that it is perhaps better not to get too carried away with the coding, and to begin working on the report earlier. Ideally, the two tasks should be pipelined, so that the report for a first formulation of the problem gets written while coding the second: this would improve system resilience by allowing us to present a complete product (albeit of smaller scope) in case our task runs into a processing time limit.

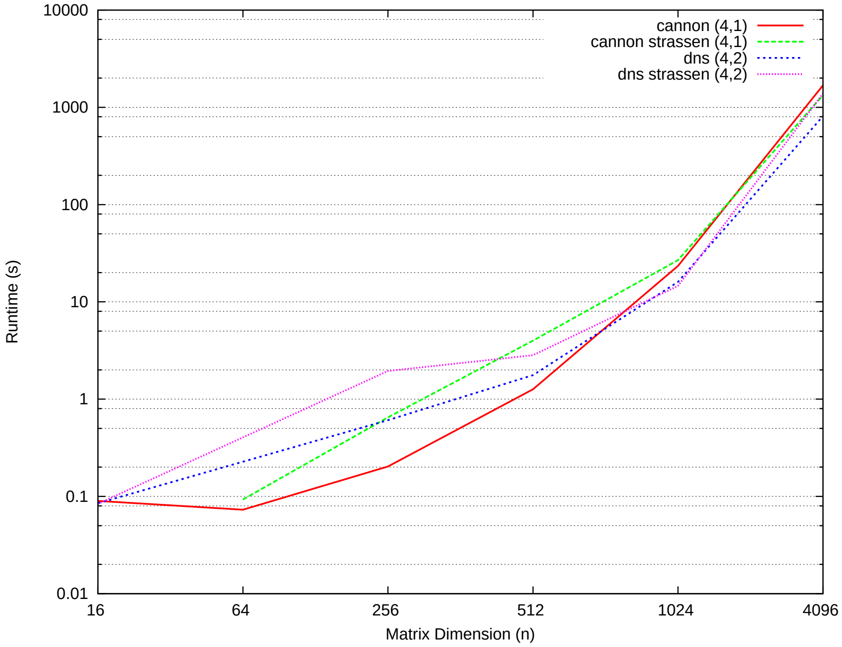

Figure 5: Runtimes of Cannon's algrithm with $(p, c) = (4, 1)$ and DNS algorithm with $(p, c) = (4, 2)$ with naive and Strassen's serial matrix multiplication algorithms over n for $k = 2$.

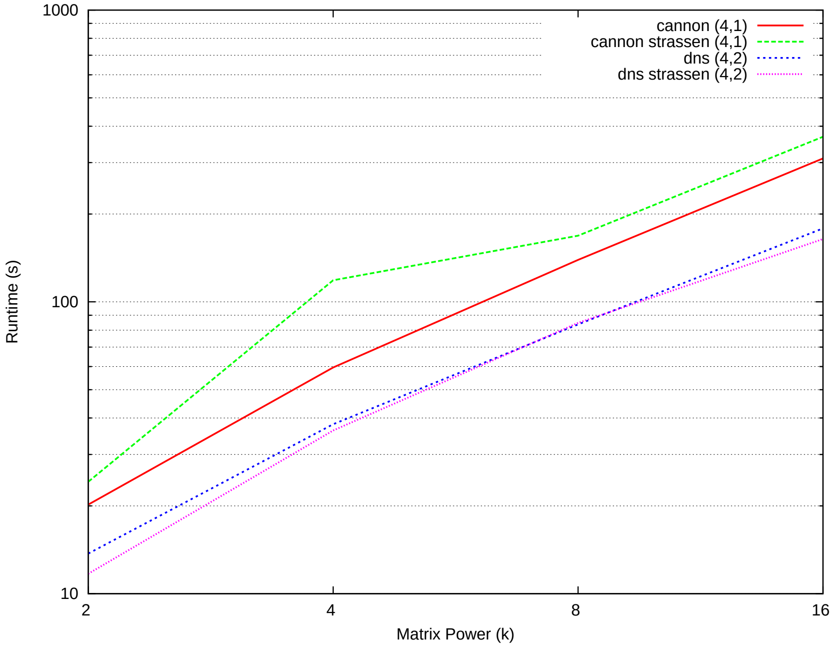

Figure 6: Runtimes of Cannon's algrithm with $(p, c) = (4, 1)$ and DNS algorithm with $(p, c) = (4, 2)$ with naive and Strassen's serial matrix multiplication algorithms over k for $n = 1024$.

```c
/* ===================================================================
 *
 * CS 566 - Assignment 03
 * Camillo Lugaresi, Cosmin Stroe
 *
 * This code implements Cannon's algorithm for matrix multiplication
 * on a cluster using MPI.
 *
 * Cannon's algorithm uses a 2D Mesh to partition the output data
 * of the matrix multiplication problem.
 *
 * =================================================================== */

#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <math.h>
#include <stddef.h>
#include <string.h>
#include <unistd.h>
#include "common.h"
#include "cannon.h"

// setup our variables
void setup_info(struct problem *info, int numprocs)
{
        info->p = numprocs;
        int sqp = 1;
        while (sqp*sqp < numprocs) sqp++;
        if (sqp*sqp != numprocs) {
                fprintf(stderr, "ERROR: processor count %d is not a square ", numprocs);
                MPI_Finalize();
                exit(1);
        }
        info->sqp = sqp;

        info->blksz = info->n / sqp;
        if (info->n % sqp) {
                fprintf(stderr, "ERROR: problem size %d is not multiple of square root of processor "
                        "count %d\n", info->n, numprocs);
                MPI_Finalize();
                exit(1);
        }
        info->blkcells = info->blksz*info->blksz;
}

// setup the mesh topology
void setup_mesh(struct problem *info)
{
        /* setup mesh */
        int dims[2] = {info->sqp, info->sqp};
        int periods[2] = {1, 1};        /* wraparound */
        MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 0, &info->mesh);
        MPI_Comm_rank(info->mesh, &(info->rank));

        /* find my coordinates */
        MPI_Cart_coords(info->mesh, info->rank, 2, info->coords);

        /* make row and column communicators */
        int keepdims[2];
        keepdims[HDIM] = 1;
        keepdims[VDIM] = 0;
        MPI_Cart_sub(info->mesh, keepdims, &info->hcomm);
        keepdims[HDIM] = 0;
        keepdims[VDIM] = 1;
        MPI_Cart_sub(info->mesh, keepdims, &info->vcomm);
```

```c
}

// allocate the matrix according to the input parameters
void setup_matrix(struct problem *info, struct input_params *m_in)
{
        if (info->rank == 0) {
                alloc_matrix(&info->X, info->n);
                fill_matrix(&info->X, m_in);

                info->Xblocks = malloc(sizeof(*info->Xblocks)*info->n*info->n);
                /* rearrange data into blocks for scatter */
                matrix_to_blocks(&info->X, info->Xblocks, info->blksz, 0);
        }

        alloc_matrix(&info->Xb, info->blksz);
        MPI_Scatter(info->Xblocks, info->blkcells, MPI_INT, info->Xb.data, info->blkcells, MPI_INT, 0,
info->mesh);

        alloc_matrix(&info->A, info->blksz);
        alloc_matrix(&info->B, info->blksz);
        alloc_matrix(&info->C, info->blksz);
        info->temp = malloc(sizeof(*info->temp)*info->blkcells);
}


// the shift operation performed (twice) at every iteration
void ring_shift(int *data, int *temp, size_t count, MPI_Comm ring, int delta)
{
        int src, dst;
        MPI_Status status;

        MPI_Cart_shift(ring, 0, delta, &src, &dst);
        MPI_Sendrecv(data, count, MPI_INT, dst, 0,
                                temp, count, MPI_INT, src, 0,
                                ring, &status);
        memcpy(data, temp, sizeof(int)*count);
}

// the cannon matrix multiplication algorithm.
void cannon(struct problem *info, int strassen)
{
        int i, j;
        struct matrix *A = &info->A, *B = &info->B, *C = &info->C;

        /* skew */
        ring_shift(A->data, info->temp, info->blkcells, info->hcomm, -info->coords[VDIM]);
        ring_shift(B->data, info->temp, info->blkcells, info->vcomm, -info->coords[HDIM]);

        /* shift and compute */
        bzero(C->data, sizeof(*C->data)*info->blkcells);
        for (i = 0; i < info->sqp; i++) {
                if (strassen) strassen_matrix_mult_add(C, A, B);
                else naive_matrix_mult_add(C, A, B);
                ring_shift(A->data, info->temp, info->blkcells, info->hcomm, -1);
                ring_shift(B->data, info->temp, info->blkcells, info->vcomm, -1);
        }
}

int main( int argc, char *argv[] )
{
        int numprocs, namelen, i;
        char processor_name[MPI_MAX_PROCESSOR_NAME];

        /* setup */
        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
        MPI_Get_processor_name(processor_name,&namelen);
        double start_time = MPI_Wtime();

        struct problem info_s, *info = &info_s;
        struct input_params m_in_s, *m_in = &m_in_s;

        int result = parse_args(argc, argv, &info->n, &info->k, m_in);
        if (result != 0) {
                MPI_Finalize();
                exit(result);
        }

        setup_info(info, numprocs);
        setup_mesh(info);
        setup_matrix(info, m_in);

        double load_time = MPI_Wtime();

        /* run Cannon's algorithm */
        memcpy(info->C.data, info->Xb.data, sizeof(*info->Xb.data)*info->blkcells);
        for (i = 1; i < info->k; i++) {
                memcpy(info->A.data, info->Xb.data, sizeof(*info->Xb.data)*info->blkcells);
                memcpy(info->B.data, info->C.data, sizeof(*info->C.data)*info->blkcells);
                cannon(info, m_in->strassen);
        }

        double cannon_time = MPI_Wtime();

        /* gather the product */
        MPI_Gather(info->C.data, info->blkcells, MPI_INT, info->Xblocks, info->blkcells, MPI_INT, 0, info-
>mesh);
        if (info->rank == 0) {
                alloc_matrix(&info->Xpow, info->n);
                blocks_to_matrix(&info->Xpow, info->Xblocks, info->blksz, 0);
        }

        double gather_time = MPI_Wtime();

        number_type determinant = m_in->lu2d ? lu2d_determinant(info) : lu1d_determinant(info);

        double det_time = MPI_Wtime();

        /* print results */
        if (info->rank == 0) {
                if (m_in->print) {
                        printf("X:\n");
                        print_matrix(&info->X);
                        printf("X^%d:\n", info->k);
                        print_matrix(&info->Xpow);
                }

                /* print result */
                extern double convert_time;
                extern double lu_time;
                printf("determinant: %f\n", determinant);
                double elapsed_time = MPI_Wtime() - start_time;
                printf("data loading time: %f\n", load_time - start_time);
                printf("matrix multiplication time: %f\n", cannon_time - load_time);
                printf("gather time: %f\n", gather_time - cannon_time);
                printf("convert time: %f\n", convert_time - gather_time);
                printf("LU time: %f\n", lu_time - convert_time);
                printf("determinant time: %f\n", det_time - lu_time);
                printf("total time: %f\n", elapsed_time);

                /* profiling */
                printf("\n");
        }

        MPI_Finalize();
        return 0;
}
```

```
/* ===================================================================
 *
 * CS 566 - Assignment 03
 * Camillo Lugaresi, Cosmin Stroe
 *
 * Header file for shared declarations.
 *
 * =================================================================== */

#define VDIM 0
#define HDIM 1

struct problem {
        MPI_Comm mesh;
        int rank;
        int n;
        int k;
        int blksz;
        int blkcells;
        int p;
        int sqp;
        int coords[2];
        MPI_Comm hcomm, vcomm;
        struct matrix X;        // only root uses this
        int *Xblocks;
        struct matrix Xpow;     // only root uses this
        struct matrix Xb;       // original block:
        struct matrix A;        // cannon matrix 1
        struct matrix B;        // cannon matrix 2
        struct matrix C;        // cannon product
        int *temp;                      // scratch space for shift
        MPI_Comm rowring;
        int rowblksz;
};

/* LU things */

#define MPI_number_type MPI_DOUBLE
#define number_type double

struct fmatrix {
        int n;
        number_type *data;
};
void alloc_fmatrix(struct fmatrix *m, int n);

struct pivot {
        int row;
        number_type value;
};

void best_pivot( void *invec, void *inoutvec, int *len, MPI_Datatype *datatype);
int setup_pivot_struct(MPI_Datatype *pivot_type, MPI_Op *best_pivot_op);
void LU_decomp(struct problem *info, struct fmatrix *X, int *reorder, MPI_Datatype pivot_type, MPI_Op
best_pivot_op);
int count_swaps(int *reorder_all, int n);
number_type lu1d_determinant(struct problem *info);
number_type lu2d_determinant(struct problem *info);
```

```
/* ===================================================================
 *
 * CS 566 - Assignment 03
 * Camillo Lugaresi, Cosmin Stroe
 *
 * This code implements the DNS (Dekel, Nassimi, Sahni) matrix multiplication
 * algorithm on a cluster using MPI.
 *
 * The DNS algorithm uses a 3D Mesh to partition the intermediate data
 * of the matrix multiplication problem.
 *
 * =================================================================== */

#include "mpi.h"
#include "common.h"
#include "cannon.h" // for struct problem
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <strings.h>
#include <string.h>

#define iDIM 0
#define jDIM 1
#define kDIM 2

struct mesh_info {
        MPI_Comm mesh3d;
        MPI_Comm mesh_ik, ring_j;
        MPI_Comm mesh_jk, ring_i;
        MPI_Comm mesh_ij, ring_k;

        int myrank;
        int coords[3]; // my coordinates in the 3D mesh

        int numprocs; // the number of processors in our cluster

        struct matrix mtx; // the matrix we are multiplying (n x n)

        int q;  // the number of processors in each dimension of the 3D-Mesh
                // each processor will receive two blocks of (n/q)*(n/q) elements

        int n; // the size of the matrix dimension

        int k; // the power
};


void create_topology( struct mesh_info *info , int argc, char *argv[] ) {

        // MPI Initialization
        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &info->numprocs);

        for( info->q = 1; (info->q+1)*(info->q+1)*(info->q+1) <= info->numprocs; info->q++ );

        // Create the Topology which we will be using.
        int *dims = malloc( sizeof(int) * 3 ); // 3 dimensions
        int *periods = malloc( sizeof(int) * 3); // wraparound
        dims[iDIM] = dims[jDIM] = dims[kDIM] = info->q;
        periods[iDIM] = periods[jDIM] = periods[kDIM] = 1;

        MPI_Cart_create(MPI_COMM_WORLD, 3, dims, periods, 0, &info->mesh3d);

        MPI_Comm_rank( info->mesh3d, &info->myrank);
        MPI_Cart_coords( info->mesh3d, info->myrank, 3, info->coords);
        // the i-k plane
        dims[iDIM] = dims[kDIM] = 1;
        dims[jDIM] = 0;

        MPI_Cart_sub( info->mesh3d, dims, &info->mesh_ik );

        // the j ring
        dims[iDIM] = dims[kDIM] = 0;
        dims[jDIM] = 1;

        MPI_Cart_sub( info->mesh3d, dims, &info->ring_j);

        // the j-k plane
        dims[jDIM] = dims[kDIM] = 1;
        dims[iDIM] = 0;

        MPI_Cart_sub( info->mesh3d, dims, &info->mesh_jk);

        //the i ring
        dims[jDIM] = dims[kDIM] = 0;
        dims[iDIM] = 1;

        MPI_Cart_sub( info->mesh3d, dims, &info->ring_i);

        // the i-j plane
        dims[iDIM] = dims[jDIM] = 1;
        dims[kDIM] = 0;

        MPI_Cart_sub( info->mesh3d, dims, &info->mesh_ij);

        // the k rings
        dims[iDIM] = dims[jDIM] = 0;
        dims[kDIM] = 1;

        MPI_Cart_sub( info->mesh3d, dims, &info->ring_k);
}

/*
   Distribute the left or right hand matrix.
*/
void distribute_matrix(struct mesh_info *info, struct matrix *A, struct matrix *Asub, int ringdim, int
bycolumn) {

        int *Ablocks;
        int blksz = info->n / info->q;
        alloc_matrix(Asub, blksz);


        MPI_Comm mesh, ring;
        if (ringdim == jDIM) {
                mesh = info->mesh_ik;
                ring = info->ring_j;
        } else {
                mesh = info->mesh_jk;
                ring = info->ring_i;
        }

        if( info->myrank == 0 ) {
                Ablocks = malloc( sizeof(*A->data) * A->n * A->n );
                matrix_to_blocks(A, Ablocks, blksz, bycolumn);
        }

        if( info->coords[ringdim] == 0 ) {
                MPI_Scatter( Ablocks, blksz*blksz, MPI_INT, Asub->data, blksz*blksz, MPI_INT, 0, mesh );
        }

        if( info->myrank == 0 ) free (Ablocks);
```

```c
        MPI_Bcast( Asub->data, blksz*blksz, MPI_INT, 0, ring);
}


void gather_C(struct mesh_info *info, struct matrix *Cred, struct matrix *C) {

        int *Cblocks;
        int blksz = info->n / info->q;

        if( info->myrank == 0 ) Cblocks = calloc( sizeof(*C->data), C->n * C->n );

        MPI_Gather( Cred->data, blksz*blksz, MPI_INT, Cblocks, blksz*blksz, MPI_INT, 0, info->mesh_ij);
        if( info->myrank == 0 ) {
                blocks_to_matrix(C, Cblocks, blksz, 0);
                free(Cblocks);
        }
}

/*
   This is the DNS algorithm implementation.
   The multiplied matrix C will only be returned on the root (rank = 0) node.
*/
void dns_multiply(struct mesh_info *info, struct matrix *C, struct matrix *A, struct matrix *B, int
strassen)
{
        struct matrix Asub, Bsub, Csub, Cred;

        // distribute A along the i-k plane and then in the j direction
        distribute_matrix(info, A, &Asub, jDIM, 0);

        // distribute B along the k-j plane and then in the i direction
        distribute_matrix(info, B, &Bsub, iDIM, 1);

        alloc_matrix(&Csub, Asub.n);

        // do the serial matrix multiplication
        bzero(Csub.data, sizeof(*Csub.data)*Csub.n*Csub.n);
        if( strassen ) {
                strassen_matrix2_mult(&Csub, &Asub, &Bsub);
        }
        else {
                naive_matrix_mult_add(&Csub, &Asub, &Bsub);
        }

        // reduce along k dimension to the i-j plane
        if( info->coords[kDIM] == 0 ) alloc_matrix(&Cred, Csub.n);
        MPI_Reduce( Csub.data, Cred.data, Csub.n*Csub.n, MPI_INT, MPI_SUM, 0, info->ring_k);

        // gather on the i-j plane to the root node.
        if( info->coords[kDIM] == 0 ) gather_C(info, &Cred, C);

}


int main( int argc, char *argv[] ) {

        struct input_params m_in_s, *m_in = &m_in_s;
        struct mesh_info info;

        create_topology(&info, argc, argv);

        int result = parse_args(argc, argv, &info.n, &info.k, m_in);
        if (result != 0) {
                MPI_Finalize();
                exit(result);
        }
        double start_time = MPI_Wtime();

        // error checking
        if( info.q*info.q*info.q != info.numprocs ) {
                fprintf(stderr, "ERROR: The number of processors must be a perfect cube (not %d).\n",
info.numprocs);
                MPI_Finalize();
                return 1;
        }

        // required for calling the LU decomposition functions.
        struct problem pinfo;
        pinfo.n = info.n;
        pinfo.p = info.numprocs;
        pinfo.k = info.k;
        pinfo.rank = info.myrank;

        alloc_matrix(&pinfo.X, pinfo.n);
        fill_matrix(&pinfo.X, m_in);


        double load_time = MPI_Wtime();

        struct matrix B, C;
        if( info.myrank == 0 ) {
                alloc_matrix(&C, info.n);
                alloc_matrix(&B, info.n);
                memcpy(B.data, pinfo.X.data, sizeof(*pinfo.X.data)*pinfo.X.n*pinfo.X.n);
        }

        int i;
        for( i = 1; i < info.k; i++ ) {
                dns_multiply( &info, &C, &pinfo.X, &B, 0);
                if( info.myrank == 0 ) {
                        memcpy(B.data, C.data, sizeof(*pinfo.X.data)*pinfo.X.n*pinfo.X.n);
                }
        }


        pinfo.Xpow.data = C.data;
        pinfo.Xpow.n = C.n;

        double dns_time = MPI_Wtime();

        number_type determinant = lu1d_determinant(&pinfo);

        double lu_time = MPI_Wtime();

        if( info.myrank == 0 ) {
                if( m_in->print ) {
                        printf("X:\n");
                        print_matrix(&pinfo.X);
                        printf("X^%d:\n", pinfo.k);
                        print_matrix(&pinfo.Xpow);
                }
                printf("determinant: %f\n", determinant);

                double elapsed_time = MPI_Wtime() - start_time;
                printf("data loading time: %f\n", load_time - start_time);
                printf("matrix multiplication time: %f\n", dns_time - load_time);
                printf("LU time: %f\n", lu_time - dns_time);
                printf("total time: %f\n", elapsed_time);
        }

        MPI_Barrier(info.mesh3d);  // avoid processors that end early from killing the rest.

        MPI_Finalize();
        return 0;

}
```

```
/* ================================================================
 *
 * CS 566 - Assignment 03
 * Camillo Lugaresi, Cosmin Stroe
 *
 * This code implements LU decomposition using a ring topology,
 * with pipelined communication on a cluster using MPI.
 *
 * ================================================================ */

#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <math.h>
#include <stddef.h>
#include <unistd.h>
#include <string.h>
#include "common.h"
#include "cannon.h"

struct rmatrix {
        int n;
        int m;
        int *data;
};

struct rfmatrix {
        int n;
        int m;
        number_type *data;
};

void print_rfmatrix(struct rfmatrix *m)
{
        int i;
        int count = m->n*m->m;
        number_type *p = m->data;

        for (i = 1; i <= count; i++) {
                printf("% f", *p);
                if (i % m->n == 0) printf("\n");
                else printf(" ");
                p++;
        }
}

struct cpivot {
        int column;
        number_type value;
};

int setup_cpivot_struct(MPI_Datatype *pivot_type)
{
        int blocklen[2] = {1,1};
        MPI_Aint offsets[2] = { 0, offsetof(struct cpivot, value) };
        MPI_Datatype types[2] = { MPI_INT, MPI_number_type };
        MPI_Type_create_struct(2, blocklen, offsets, types, pivot_type);
        MPI_Type_commit(pivot_type);
        return 0;
}

void pipeline_down(struct problem *info, int start, void *buf, int count, MPI_Datatype datatype, int tag,
MPI_Request *req)
{
        MPI_Status status;
        int up = (info->rank+(info->p)-1) % (info->p);
        int down = (info->rank+1)%(info->p);
        if (info->rank != start) {
                MPI_Recv(buf, count, datatype, up, tag, info->rowring, &status);
        }
        if (down != start) {
                MPI_Isend(buf, count, datatype, down, tag, info->rowring, req);
        }
}

void LU_decomp_1d(struct problem *info, struct rfmatrix *X, int *reorder, MPI_Datatype pivot_type)
{
        int gr, r, c;
        number_type *pivot_row = malloc(sizeof(*pivot_row)*info->n);
        MPI_Request pivot_req = MPI_REQUEST_NULL, pivot_row_req = MPI_REQUEST_NULL;
        MPI_Status status;

        for (gr = 0; gr < info->n; gr++) {      /* global row number */
                if (pivot_req != MPI_REQUEST_NULL) MPI_Wait(&pivot_req, &status);

                int rproc = gr % (info->p);     /* proc with this row */
                r = gr / (info->p);                     /* row within this proc */

                /* we do partial pivoting, so the pivot is on this row */
                struct cpivot pivot = { -1, 0. };
                if (info->rank == rproc) {
                        for (c = 0; c < info->n; c++) {
                                if (reorder[c] > gr && (pivot.value == 0 || fabs(CELL(X, r, c)) > fabs
(pivot.value))) {
                                        pivot.column = c;
                                        pivot.value = CELL(X, r, c);
                                }
                        }
                }
                pipeline_down(info, rproc, &pivot, 1, pivot_type, 89, &pivot_req);

                /* fill in reorder */
                reorder[pivot.column] = gr;

                /* distribute a vector */
                if (pivot_row_req != MPI_REQUEST_NULL) MPI_Wait(&pivot_row_req, &status);
                if (info->rank == rproc) memcpy(pivot_row, &CELL(X, r, 0), sizeof(*pivot_row)*info->n);
                pipeline_down(info, rproc, pivot_row, info->n, MPI_number_type, 94, &pivot_row_req);

                /* elimination */
                int startr = gr / (info->p);
                if (info->rank <= rproc) startr++;

                for (r = startr; r < info->rowblksz; r++) {

                        number_type m = CELL(X, r, pivot.column) / pivot.value;
                        CELL(X, r, pivot.column) = m;

                        for (c = 0; c < info->n; c++) {
                                if (reorder[c] > gr) {  /* this also excludes the pivot column */
                                        CELL(X, r, c) -= m*pivot_row[c];
                                }
                        }
                }
        }
}

double convert_time;
double lu_time;

/* note that this only returns the correct value in processor with rank 0 */
number_type lu1d_determinant(struct problem *info)
{
        int i;
        info->rowblksz = info->n / info->p;
        int *allblockdata, *blockdata;
        int rowblkcells = info->n*info->rowblksz;

        if (info->rank == 0) {
                allblockdata = malloc(sizeof(*allblockdata)*info->n*info->n);
                matrix_to_rowblocks_cyclic(&info->Xpow, allblockdata, info->rowblksz);
        }

        /* setup rowblock ring */
        int dims[1] = {info->p};
        int periods[1] = {1};   /* wraparound */
        MPI_Cart_create(MPI_COMM_WORLD, 1, dims, periods, 0, &info->rowring);

        /* distribute row blocks */
        blockdata = malloc(sizeof(*blockdata)*rowblkcells);
        MPI_Scatter(allblockdata, rowblkcells, MPI_INT, blockdata, rowblkcells, MPI_INT, 0, info-
>rowring);

        /* convert to float */
        struct rfmatrix fblk;
        fblk.n = info->n;
        fblk.m = info->rowblksz;
        fblk.data = malloc(sizeof(*fblk.data)*rowblkcells);
        for (i = 0; i < rowblkcells; i++) fblk.data[i] = blockdata[i];
        convert_time = MPI_Wtime();

        /* setup data type for pivoting */
        MPI_Datatype pivot_type;
        setup_cpivot_struct(&pivot_type);

        /* prepare reorder array */
        int *reorder = malloc(info->n * sizeof(*reorder));
        for (i = 0; i < info->n; i++) reorder[i] = INT_MAX;

        LU_decomp_1d(info, &fblk, reorder, pivot_type);
        lu_time = MPI_Wtime();

        /* calculate the determinant */
        number_type prod = 1.0;
        for (i = 0; i < info->n; i++) {
                int gr = reorder[i];
                int rproc = gr % (info->p);     /* proc with this row */
                if (rproc == info->rank) {
                        int r = gr / (info->p);                 /* row within this proc */
                        prod *= CELL(&fblk, r, i);
                }
        }
        number_type determinant;
        MPI_Reduce(&prod, &determinant, 1, MPI_number_type, MPI_PROD, 0, info->rowring);

        /* we must adjust the determinant's sign based on the permutations */
        if (info->rank == 0) {
                if (count_swaps(reorder, info->n) % 2) determinant *= -1;
        }
        return determinant;
}
```

```c
/* ===================================================================
 *
 * CS 566 - Assignment 03
 * Camillo Lugaresi, Cosmin Stroe
 *
 * This code implements LU decomposition using a 2D mesh topology,
 * with pipelined communication on a cluster using MPI.
 *
 * =================================================================== */

#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <math.h>
#include <stddef.h>
#include <unistd.h>
#include "common.h"
#include "cannon.h"

void alloc_fmatrix(struct fmatrix *m, int n)
{
        m->n = n;
        m->data = malloc(sizeof(*m->data) * m->n * m->n);
}

void best_pivot( void *invec, void *inoutvec, int *len, MPI_Datatype *datatype)
{
        struct pivot *a = invec, *b = inoutvec;
        int i;
        /* it's important to break ties, or different rows may believe they have
           the pivot */
        for (i = 0; i < *len; i++, a++, b++) {
                if ((fabs(a->value) > fabs(b->value)) ||
                        (fabs(a->value) == fabs(b->value) && a->row < b->row)) {
                        *b = *a;
                }
        }
}

int setup_pivot_struct(MPI_Datatype *pivot_type, MPI_Op *best_pivot_op)
{
        int blocklen[2] = {1,1};
        MPI_Aint offsets[2] = { 0, offsetof(struct pivot, value) };
        MPI_Datatype types[2] = { MPI_INT, MPI_number_type };
        MPI_Type_create_struct(2, blocklen, offsets, types, pivot_type);
        MPI_Type_commit(pivot_type);

        MPI_Op_create(best_pivot, 1, best_pivot_op);
        return 0;
}

double pivot_time = 0;
double pivot_allr_time = 0;
double pivot_bcast_time = 0;
double m_bcast_time = 0;
double a_bcast_time = 0;

void pipeline_right(struct problem *info, int pivot_h, void *buf, int count, MPI_Datatype datatype, int
tag, MPI_Request *req)
{
        MPI_Status status;

        if (info->coords[HDIM] > pivot_h && info->coords[HDIM] < info->sqp) {
                MPI_Recv(buf, count, datatype, info->coords[HDIM]-1, tag, info->hcomm, &status);
        }
        if (info->coords[HDIM] >= pivot_h && info->coords[HDIM] < info->sqp-1) {
                MPI_Isend(buf, count, datatype, info->coords[HDIM]+1, tag, info->hcomm, req);
        }
}

void LU_decomp(struct problem *info, struct fmatrix *X, int *reorder, MPI_Datatype pivot_type, MPI_Op
best_pivot_op)
{
        MPI_Request req_spiv, req_sa, req_sm;
        MPI_Status status;

        number_type *m = malloc(info->blksz * sizeof(*m));
        int diag;
        for (diag = 0; diag < info->n; diag++) {
                /* we do partial pivoting, so the proc with the pivot is on this column: */
                int pivot_h = diag / info->blksz;
                int r, c, i;

                double start_time = MPI_Wtime();
                double start_time2;

                struct pivot pivot = { -1, 0. };
                /* choose pivot across the column */
                if (info->coords[HDIM] == pivot_h) {
                        /* column with pivot in block */
                        int pivot_c = diag % info->blksz;
                        /* Argo doesn't want aliasing in allreduce */
                        struct pivot pivot_cand = { -1, 0. };
                        for (i = 0; i < info->blksz; i++) {
                                if (reorder[i] > diag && fabs(CELL(X, i, pivot_c)) > fabs
(pivot_cand.value)) {
                                        pivot_cand.row = info->blksz*info->coords[VDIM] + i;
                                        pivot_cand.value = CELL(X, i, pivot_c);
                                }
                        }
                        start_time2 = MPI_Wtime();
                        MPI_Allreduce(&pivot_cand, &pivot, 1, pivot_type, best_pivot_op, info->vcomm);
                        pivot_allr_time += MPI_Wtime() - start_time2;
                }
                /* broadcast pivot choice across row towards the right */
                start_time2 = MPI_Wtime();
                pipeline_right(info, pivot_h, &pivot, 1, pivot_type, 45, &req_spiv);
                pivot_bcast_time += MPI_Wtime() - start_time2;

                pivot_time += MPI_Wtime() - start_time;

                /* find rank of proc with pivot on the vertical communicator */
                int pivot_v = pivot.row / info->blksz;

                /* fill in reorder */
                if (info->coords[VDIM] == pivot_v) {
                        reorder[pivot.row % info->blksz] = diag;
                }

                /* calculate and distribute the ms */
                for (r = 0; r < info->blksz; r++) {
                        if (reorder[r] > diag) {
                                if (info->coords[HDIM] == pivot_h) {
                                        int pivot_c = diag % info->blksz;
                                        m[r] = CELL(X, r, pivot_c) / pivot.value;
                                        CELL(X, r, pivot_c) = m[r];
                                }
                                /* broadcast m towards right */
                                start_time = MPI_Wtime();
                                pipeline_right(info, pivot_h, &m[r], 1, MPI_number_type, 64, &req_sm);
                                m_bcast_time += MPI_Wtime() - start_time;
                        }
                }
                /* distribute the pivot row and eliminate */
                int startc = 0;
                if (info->coords[HDIM] == pivot_h) startc = (diag+1) % info->blksz;
                if (info->coords[HDIM] < pivot_h) startc = info->blksz;
                /* elimination */
                for (c = startc; c < info->blksz; c++) {
                        number_type a;
                        if (info->coords[VDIM] == pivot_v) {
                                a = CELL(X, pivot.row % info->blksz, c);
                        }

                        start_time = MPI_Wtime();

                        int up = (info->coords[VDIM]+info->sqp-1)%info->sqp;
                        int down = (info->coords[VDIM]+1)%info->sqp;
                        if (info->coords[VDIM] != pivot_v) {
                                MPI_Recv(&a, 1, MPI_number_type, up, 78, info->vcomm, &status);
                        }
                        if (down != pivot_v) {
                                MPI_Isend(&a, 1, MPI_number_type, down, 78, info->vcomm, &req_sa);
                        }

                        a_bcast_time += MPI_Wtime() - start_time;

                        for (r = 0; r < info->blksz; r++) {
                                if (reorder[r] > diag) {
                                        CELL(X, r,c) -= m[r]*a;
                                }
                        }
                        if (down != pivot_v) MPI_Wait(&req_sa, &status);
                }
        }
}

double convert_time;
double lu_time;

/* note that this only returns the correct value in processor with rank 0 */
number_type lu2d_determinant(struct problem *info)
{
        int i;

        /* compute the determinant using LU decomposition */
        /* setup data type for pivoting */
        MPI_Datatype pivot_type;
        MPI_Op best_pivot_op;
        setup_pivot_struct(&pivot_type, &best_pivot_op);

        int *reorder = malloc((info->rank == 0 ? info->n : info->blksz) * sizeof(*reorder));
        for (i = 0; i < info->blksz; i++) reorder[i] = INT_MAX;

        /* convert int matrix to float */
        struct fmatrix fC;
        alloc_fmatrix(&fC, info->C.n);
        for (i = 0; i < info->blkcells; i++) {
                fC.data[i] = info->C.data[i];
        }
        MPI_Barrier(info->mesh);
        convert_time = MPI_Wtime();

        /* do the LU */
        LU_decomp(info, &fC, reorder, pivot_type, best_pivot_op);

        MPI_Barrier(info->mesh);
        lu_time = MPI_Wtime();
        /* calculate the determinant */
```

```c
        number_type prod = 1.0;
        for (i = 0; i < info->blksz; i++) {
                if (reorder[i] == INT_MAX) continue;
                int c = reorder[i] - info->coords[HDIM]*info->blksz;
                if (c >= 0 && c < info->blksz) {
                        prod *= CELL(&fC, i, c);
                }
        }
        number_type determinant;
        MPI_Reduce(&prod, &determinant, 1, MPI_number_type, MPI_PROD, 0, info->mesh);

        /* we must adjust the determinant's sign based on the permutations */
        /* but only the rightmost column has the full reorder! */
        MPI_Request req;
        MPI_Status status;
        if (info->coords[HDIM] == info->sqp-1) {
                MPI_Isend(reorder, info->blksz, MPI_INT, 0, 75, info->hcomm, &req);
        }
        if (info->coords[HDIM] == 0) {
                MPI_Recv(reorder, info->blksz, MPI_INT, info->sqp-1, 75, info->hcomm, &status);

                MPI_Gather(reorder, info->blksz, MPI_INT, reorder, info->blksz, MPI_INT, 0, info->vcomm);
                if (info->rank == 0) {
                        if (count_swaps(reorder, info->n) % 2) determinant *= -1;
                }
        }
        return determinant;
}
```

```c
/* ==================================================================
 *
 * CS 566 - Assignment 03
 * Camillo Lugaresi, Cosmin Stroe
 *
 * Common operations, including serial matrix multiplication algorithms
 * and their helper functions (naive and Strassen), matrix allocation functions,
 * matrix reordering functions, and various other helper functions.
 *
 * ================================================================== */

#include "common.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

void alloc_matrix(struct matrix *m, int n)
{
        m->n = n;
        m->data = malloc(sizeof(*m->data) * m->n * m->n);
}

void alloc_matrix2(struct matrix2 *m, int n)
{
        m->n = m->stride = n;
        m->data = malloc(sizeof(*m->data) * m->n * m->n);
}

void make_matrix2(struct matrix *m, struct matrix2 *m2)
{
        m2->n = m2->stride = m->n;
        m2->data = m->data;
}

void make_submatrix(struct matrix2 *m, struct matrix2 *sub, int r, int c, int blksz)
{
        sub->stride = m->stride;
        sub->data = m->data + m->stride*r + c;
        sub->n = blksz;
}

/* C = C + A*B */
void naive_matrix_mult_add(struct matrix *C, struct matrix *A, struct matrix *B)
{
        int i,j,k;
        int n = C->n;
        for (i = 0; i < n; i++) {
                for (j = 0; j < n; j++) {
                        int c = 0;
                        for (k = 0; k < n; k++) c += CELL(A,i,k)*CELL(B,k,j);
                        CELL(C,i,j) += c;
                }
        }
}

/* C = A*B */
void naive_matrix2_mult(struct matrix2 *C, struct matrix2 *A, struct matrix2 *B)
{
        int i,j,k;
        int n = C->n;
        for (i = 0; i < n; i++) {
                for (j = 0; j < n; j++) {
                        int c = 0;
                        for (k = 0; k < n; k++) c += CELL2(A,i,k)*CELL2(B,k,j);
                        CELL2(C,i,j) = c;
                }
        }
}

/* C = A+B */
void matrix2_sum(struct matrix2 *C, struct matrix2 *A, struct matrix2 *B)
{
        int i,j;
        int n = A->n;
        for (i = 0; i < n; i++) {
                for (j = 0; j < n; j++) {
                        CELL2(C,i,j) = CELL2(A,i,j) + CELL2(B,i,j);
                }
        }
}

/* C = A-B */
void matrix2_diff(struct matrix2 *C, struct matrix2 *A, struct matrix2 *B)
{
        int i,j;
        int n = A->n;
        for (i = 0; i < n; i++) {
                for (j = 0; j < n; j++) {
                        CELL2(C,i,j) = CELL2(A,i,j) - CELL2(B,i,j);
                }
        }
}

void strassen_split(struct matrix2 *A, struct matrix2 As[2][2])
{
        int blksz = A->n/2;
        make_submatrix(A, &As[0][0], 0, 0, blksz);
        make_submatrix(A, &As[0][1], 0, blksz, blksz);
        make_submatrix(A, &As[1][0], blksz, 0, blksz);
        make_submatrix(A, &As[1][1], blksz, blksz, blksz);
}

/* C = A*B */
void strassen_matrix2_mult(struct matrix2 *C, struct matrix2 *A, struct matrix2 *B)
{
        /* if we can't divide it in 4 parts, use naive algorithm */
        if (C->n % 2) {
                naive_matrix2_mult(C, A, B);
                return;
        }

        struct matrix2 As[2][2];
        strassen_split(A, As);
        struct matrix2 Bs[2][2];
        strassen_split(B, Bs);
        struct matrix2 Cs[2][2];
        strassen_split(C, Cs);

        int blksz = A->n/2;
        int i;
        struct matrix2 M[7];
        for (i = 0; i < 7; i++) alloc_matrix2(&M[i], blksz);

        struct matrix2 tmp1, tmp2;
        alloc_matrix2(&tmp1, blksz);
        alloc_matrix2(&tmp2, blksz);

        // M1 = (A11 + A22)*(B11 + B22)
        matrix2_sum(&tmp1, &As[0][0], &As[1][1]);
        matrix2_sum(&tmp2, &Bs[0][0], &Bs[1][1]);
        strassen_matrix2_mult(&M[0], &tmp1, &tmp2);

        // M2 = (A21 + A22)*B11
        matrix2_sum(&tmp1, &As[1][0], &As[1][1]);
        strassen_matrix2_mult(&M[1], &tmp1, &Bs[0][0]);

        // M3 = A11*(B12-B22)
        matrix2_diff(&tmp1, &Bs[0][1], &Bs[1][1]);
        strassen_matrix2_mult(&M[2], &As[0][0], &tmp1);

        // M4 = A22*(B21-B11)
        matrix2_diff(&tmp1, &Bs[1][0], &Bs[0][0]);
        strassen_matrix2_mult(&M[3], &As[1][1], &tmp1);

        // M5 = (A11+A12)*B22
        matrix2_sum(&tmp1, &As[0][0], &As[0][1]);
        strassen_matrix2_mult(&M[4], &tmp1, &Bs[1][1]);

        // M6 = (A21-A11)*(B11+B12)
        matrix2_diff(&tmp1, &As[1][0], &As[0][0]);
        matrix2_sum(&tmp2, &Bs[0][0], &Bs[0][1]);
        strassen_matrix2_mult(&M[5], &tmp1, &tmp2);

        // M7 = (A12-A22)*(B21+B22)
        matrix2_diff(&tmp1, &As[0][1], &As[1][1]);
        matrix2_sum(&tmp2, &Bs[1][0], &Bs[1][1]);
        strassen_matrix2_mult(&M[6], &tmp1, &tmp2);

        // C11 = M1 + M4 - M5 + M7
        matrix2_sum(&tmp1, &M[0], &M[3]);
        matrix2_diff(&tmp2, &tmp1, &M[4]);
        matrix2_sum(&Cs[0][0], &tmp2, &M[6]);

        // C12 = M3 + M5
        matrix2_sum(&Cs[0][1], &M[2], &M[4]);

        // C21 = M2 + M4
        matrix2_sum(&Cs[1][0], &M[1], &M[3]);

        // C22 = M1 - M2 + M3 + M6
        matrix2_diff(&tmp1, &M[0], &M[1]);
        matrix2_sum(&tmp2, &tmp1, &M[2]);
        matrix2_sum(&Cs[1][1], &tmp2, &M[5]);

        for (i = 0; i < 7; i++) free(M[i].data);
        free(tmp1.data);
        free(tmp2.data);
}

/* C = A*B */
void strassen_matrix_mult(struct matrix *C, struct matrix *A, struct matrix *B)
{
        struct matrix2 A2,B2,C2;
        make_matrix2(A, &A2);
        make_matrix2(B, &B2);
        make_matrix2(C, &C2);

        struct matrix2 tmp;
        alloc_matrix2(&tmp, A->n);
        strassen_matrix2_mult(&tmp,&A2,&B2);
        matrix2_sum(&C2, &C2, &tmp);
        free(tmp.data);
}

/* C = A*B */
void strassen_matrix_mult_add(struct matrix *C, struct matrix *A, struct matrix *B)
{
        struct matrix2 A2,B2,C2;
        make_matrix2(A, &A2);
        make_matrix2(B, &B2);
```

```c
        make_matrix2(C, &C2);
        strassen_matrix2_mult(&C2,&A2,&B2);
}

int parse_args(int argc, char *argv[], int *n, int *k, struct input_params *m_in)
{
        int ch;
        m_in->print = 0;
        m_in->lu2d = 0;
        m_in->strassen = 0;
        while ((ch = getopt(argc, argv, "p2s")) != -1) {
                switch (ch) {
                case 'p':
                        m_in->print = 1;
                        break;
                case '2':
                        m_in->lu2d = 1;
                        break;
                case 's':
                        m_in->strassen = 1;
                        break;
                }
        }
        argc -= optind;
        argv += optind;

        if (argc < 2) {
                fprintf(stderr, "ERROR: must specify matrix size and power");
                return 1;
        }
        *n = atoi(argv[0]);
        *k = atoi(argv[1]);

        if (argc == 4) {
                m_in->mode = 1;
                m_in->u.prob[0] = atof(argv[2]);
                m_in->u.prob[1] = atof(argv[3]);
        } else if (argc == 6) {
                m_in->mode = 2;
                m_in->u.pattern[0] = atoi(argv[2]);
                m_in->u.pattern[1] = atoi(argv[4]);
                m_in->u.pattern[2] = atoi(argv[5]);
                m_in->u.pattern[3] = atoi(argv[6]);
        } else {
                fprintf(stderr, "ERROR: Invalid number of arguments (%d).\n", argc);
                return 1;
        }

        return 0;
}


/*
  Input method 1: Probabilities of -1 and +1.
        Arguments: n p1 p2

  Input method 2: Four numbers that get repeated with a shift.
        Arguments: n x1 x2 x3 x4

*/
void fill_matrix(struct matrix *m, struct input_params *m_in)
{
        int count, i;

        count = m->n*m->n;

        // Input method 1
        if (m_in->mode == 1) {
                double pm = m_in->u.prob[0];
                double pp = m_in->u.prob[1];

                for (i = 0; i < count; i++) {
                        double x = random()/(double)RAND_MAX;
                        if (x < pm) m->data[i] = -1;
                        else if (x > 1.0 - pp) m->data[i] = 1;
                        else m->data[i] = 0;
                }
        } else {
                int shift = 0;
                int r, c;

                for (r = 0; i < m->n; r++) {
                        shift += (r+1);
                        for (c = 0; c < m->n; c++) {
                                CELL(m,r,c) = m_in->u.pattern[(shift+c) % 4];
                        }
                }
        }
}

void copy_block(int blksz, struct matrix *sm, int sr, int sc,
                                                   struct matrix *dm, int dr, int dc)
{
        int i;

        int *src = sm->data + sm->n * sr + sc;
        int *dst = dm->data + dm->n * dr + dc;

        for (i = 0; i < blksz; i++) {
                memcpy(dst, src, sizeof(*dm->data)*blksz);
                src += sm->n;
                dst += dm->n;
        }
}

/* rearrange data into blocks for scatter */
void matrix_to_blocks(struct matrix *m, int *blockdata, int blksz, int column_first)
{
        struct matrix tmp;
        int i, j, s;

        tmp.data = blockdata;
        tmp.n = blksz;
        s = m->n / blksz;
        if (column_first) {
                for (j = 0; j < s; j++) {
                        for (i = 0; i < s; i++) {
                                copy_block(blksz, m, i * blksz, j * blksz, &tmp, 0, 0);
                                tmp.data += tmp.n*tmp.n;
                        }
                }
        } else {
                for (i = 0; i < s; i++) {
                        for (j = 0; j < s; j++) {
                                copy_block(blksz, m, i * blksz, j * blksz, &tmp, 0, 0);
                                tmp.data += tmp.n*tmp.n;
                        }
                }
        }
}

void blocks_to_matrix(struct matrix *m, int *blockdata, int blksz, int column_first)
{
        struct matrix tmp;
        int i, j, s;

        tmp.data = blockdata;
        tmp.n = blksz;
        s = m->n / blksz;
        if (column_first) {
                for (j = 0; j < s; j++) {
                        for (i = 0; i < s; i++) {
                                copy_block(blksz, &tmp, 0, 0, m, i * blksz, j * blksz);
                                tmp.data += tmp.n*tmp.n;
                        }
                }
        } else {
                for (i = 0; i < s; i++) {
                        for (j = 0; j < s; j++) {
                                copy_block(blksz, &tmp, 0, 0, m, i * blksz, j * blksz);
                                tmp.data += tmp.n*tmp.n;
                        }
                }
        }
}

/* rearrange data into blocks for scatter */
void matrix_to_rowblocks_cyclic(struct matrix *m, int *blockdata, int blksz)
{
        int i, j, s;
        int *dst = blockdata;

        s = m->n / blksz;
        for (i = 0; i < blksz; i++) {
                int *src = m->data + m->n*i;
                for (j = 0; j < s; j++) {
                        memcpy(dst, src, sizeof(*dst)*m->n);
                        dst += m->n;
                        src += m->n*blksz;
                }
        }
}

void print_matrix(struct matrix *m)
{
        int i;
        int count = m->n*m->n;
        int *p = m->data;

        for (i = 1; i <= count; i++) {
                printf("% d", *p);
                if (i % m->n == 0) printf("\n");
                else printf(" ");
                p++;
        }
}

/*
   Integer computation of the log base 2 of n.
*/
int intlog2( int n ) {

        int power = 0;
        int value = 1;

        while( (value << 1) <= n ) {
                power++;
                value <<= 1;
        };

        return power;
```

```c
}

/*
    Integer power of 2.
*/
int intpow2( int power ) {

        int pwr = 0;
        int value = 1;

        while( (pwr + 1) <= power ) {
                power++;
                value <<= 1;
        };

        return value;
}

int count_swaps(int *reorder_all, int n)
{
        /* to count the number of swaps performed, follow the cycles in the perm vector */
        int count = 0, i;
        for (i = 0; i < n; i++) {
                if (reorder_all[i] == -1) continue;
                int p = i, q;
                do {
                        q = p;
                        p = reorder_all[p];
                        reorder_all[q] = -1;
                        count++;
                } while (p != i);
                count--;
        }
        return count;
}
```

```c
/* ====================================================================
 *
 * CS 566 - Assignment 03
 * Camillo Lugaresi, Cosmin Stroe
 *
 * Header file for common.c
 *
 * ==================================================================== */

/* CELL(matrix,r,c) - for matrices in row,column order */
#define CELL(m,r,c) (((m)->data)[((m)->n)*(r) + (c)])

/* BLK_CELL */
#define BLK_CELL(data,blksz,b,r,c) ((data)[(blksz*blksz)*(b) + (blksz)*(r) + (c)])

struct matrix {
        int n;
        int *data;
};

struct matrix2 {
        int n;
        int stride;             // width of "mother" matrix
        int *data;
};

/* CELL2(matrix2,r,c) - for submatrices */
#define CELL2(m2,r,c) (((m2)->data)[((m2)->stride)*(r) + (c)])

struct input_params {
        int print;
        int lu2d;
        int strassen;

        int mode;
        union {
                double prob[2];
                int pattern[4];
        } u;
};

void alloc_matrix(struct matrix *m, int n);
void naive_matrix_mult_add(struct matrix *C, struct matrix *A, struct matrix *B);
int parse_args(int argc, char *argv[], int *n, int *k, struct input_params *m_in);
void fill_matrix(struct matrix *m, struct input_params *m_in);
void print_matrix(struct matrix *m);
void make_matrix2(struct matrix *m, struct matrix2 *m2);
void make_submatrix(struct matrix2 *m, struct matrix2 *sub, int r, int c, int blksz);
void copy_block(int blksz, struct matrix *sm, int sr, int sc,
                                                   struct matrix *dm, int dr, int dc);
void matrix_to_blocks(struct matrix *m, int *blockdata, int blksz, int column_first);
void blocks_to_matrix(struct matrix *m, int *blockdata, int blksz, int column_first);
void matrix_to_rowblocks_cyclic(struct matrix *m, int *blockdata, int blksz);
int intlog2( int n );
int intpow2( int power );
int count_swaps(int *reorder_all, int n);
```

```
# Makefile for all our formulations.

.PHONY : all

all : common cannon dns

clean:
        rm -f common.o cannon dns


common: common.c common.h
        mpicc -c -o common.o common.c

cannon: cannon.c lu1d.c lu2d.c common
        mpicc -o cannon cannon.c lu1d.c lu2d.c common.o

dns: dns.c lu1d.c lu2d.c common
        mpicc -o dns dns.c lu1d.c lu2d.c common.o
```