SF2568: Parallel Computations for Large-Scale Problems

Lecture 7: Matrix Multiplications and Collective Communication

February 1, 2024

Acknowledgements

These slides are an extension of slides by Michael Hanke and Niclas Jansson.

Outline

Basic Matrix Operations

- Let A and B be two matrices of dimensions $M \times N$ and $K \times L$
- **Matrix Addition** $C = A + B$ (defined if $M = K$ and $N = L$)

$$c_{m,n} = a_{m,n} + b_{m,n} \quad (0 \leq m < M, 0 \leq n < N)$$

- **Matrix Mulitplication** $C = AB$ (defined if $N = K$)

$$c_{m,l} = \sum_{n=0}^{N-1} a_{m,n} b_{n,l}$$

$c_{m,l}$ is the product of the m-th row of A and the l-th column of B

Addition of Matrices

```
for n = 0:N-1
  for m = 0:M-1
    c(m,n) = a(m,n) + b(m,n);
  end
end
```

Question: How can we parallelize this?

- Embarrassingly parallel, all data access purely local
- Any data distribution will do, no overlap necessary

```
for (m,n) in 0:M-1 x 0:N-1
  c(m,n) = a(m,n) + b(m,n);
end
```

Multiplication of Matrices

- Assume that C is initialized to zero

```
for l = 0:L-1
  for m = 0:M-1
    for n = 0:N-1
      c(m,l) = c(m,l) + a(m,n) * b(n,l);
    end
  end
end
```

Question: How can we parallelize this?

- The innermost loop implies a **global** data dependency for every $c_{m,l}$

The Challenge

Find an algorithm and a data distribution such that

1. Data doubling (duplication) on different processes is avoided
2. Excessive data movement (communication!) is avoided

What Will Come

Strategy

Consider the loops from the innermost to the outermost one

1. The innermost (n-) loop is the scalar product of two vectors
2. The n- and the m-loops are a matrix-vector multiplication for each l
3. Finally, find an implementation of the complete algorithm

```
for l = 0:L-1
  for m = 0:M-1
    for n = 0:N-1
      c(m,l) = c(m,l) + a(m,n) * b(n,l);
    end
  end
end
```

Scalar Product of Two Vectors

- The scalar product of two vectors $x, y \in \mathbb{R}^M$ is defined as

$$s = \langle x, y \rangle = x^T y = \sum_{m=0}^{M-1} x_m y_m$$

- Assume some data distribution on P processors such that x and y are distributed identically
- Then it holds

$$s = \sum_{m=0}^{M-1} x_m y_m = \sum_{p=0}^{P-1} \underbrace{\sum_{i \in I_p} x_i y_i}_{\omega_p} = \sum_{p=0}^{P-1} \omega_p$$

- Assume that all local computations are done such that ω_p is available

Global Summation

Problem

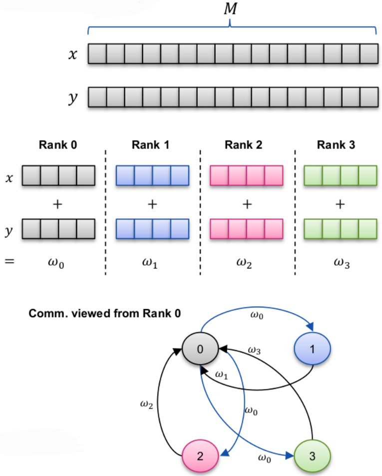
Compute the sum s on P processors,

$$s = \omega_0 + \omega_1 + \cdots + \omega_{P-1}$$

- Sequential complexity: $\mathcal{O}(P)$ operations
- Using MPI the solution to this problem: `MPI_Reduce`
- *How could it be realised?*

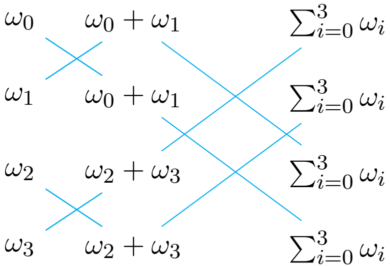
Global Summation – A First Attempt

- Compute the inner product of x and y
  - $s = \sum_{m=0}^{M-1} x_m y_m$
  - Split the elements into 4 parts of size $\widetilde{M}$
  - Sum $\widetilde{M}$ elements together on each rank
  - Add partial sums together $s = \sum_{i=0}^{3} \omega_i$
- **How to obtain the partial sums?**
  - We can only `send` and `recv` messages (data)
  - Let each rank i
    - Send ω_i to all other ranks j
    - Receive the other ranks ω_j
- Performance Analysis
  - Compute is optimal, how about communication?
  - $t_{comm} = \mathcal{O}\left(\left(P-1\right)\left(t_{startup} + t_{data}\right)\right)$

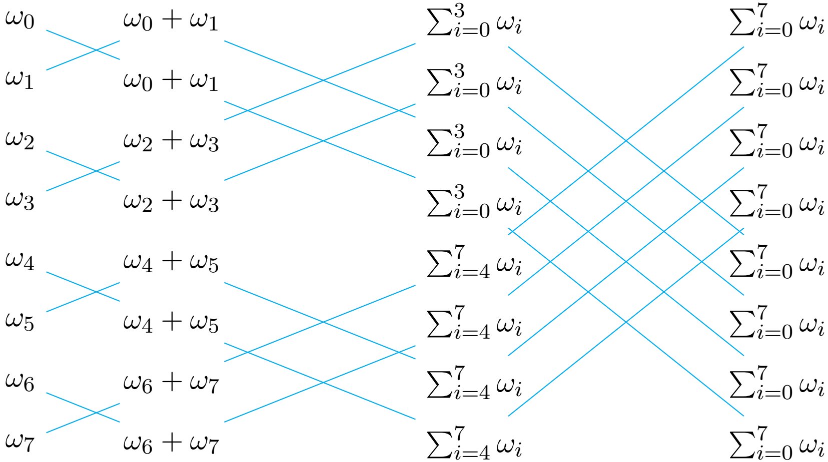


Global Summation – A New Idea

- New communication algorithm
- At step i, process p synchronizes with rank $p + 2^{i-1}$ (P power of 2)
  - In step i exchange data with rank $\bigoplus 2^i$ ($\bigoplus$ returns the bitwise XOR)
  - Combine received data with local data
- Divide and conquer "assembly pattern"
  - Quicksort, merge sort
  - Butterfly diagram (FFT)
- Performance analysis
  - How many communication steps do we need?
  - $\log P \leftarrow$ **Excellent!**



Global Summation



This communication pattern **is not** limited $P \leq 4$ processes

Global Summation – Implementation

- Assume that $P = 2^D$ is a power of 2. On process p, the program reads

```
s = omega_p;
for d = 0:D-1
  q = bitflip(p,d);
  send(s,q);
  receive(h,q);
  s = s+h;
end
```

- The `bitflip` operation inverts bit number `d` in the binary representation of `p`

Comments

- After execution of the program, every processor contains s
- This algorithm is called Recursive Doubling (Butterfly algorithm)
- The basic algorithm can be applied in many different contexts (examples: barriers, broadcast)
- Since every processor contains the final sum, its MPI counterpart is `MPI_Allreduce`
- The algorithm as given may dead-lock. Use `MPI_Sendrecv`

Modification For Any Number of Processes

Let $2^D \leq P < 2^{D+1}$

```
s = omega_p;
if p >= 2D
  send(s,bitflip(p,D));
end
if p < P-2D
  receive(h,bitflip(p,D));
  s = s+h;
end
if p < 2D
  for d = 0:D-1
    send(s,bitflip(p,d));
    receive(h,bitflip(p,d));
    s = s+h;
  end
end
if p < P-2D
  send(s,bitflip(p,D));
end
if p >= 2D
  receive(s,bitflip(p,D));
end
```

Performance Analysis

- $T_1^* = (2M-1)t_a$
- The local computation time is $t_{comp,1} = 2(I_p - 1)t_a$
- Time for recursive doubling

$$t = D(2(t_{startup} + 1 \cdot t_{data}) + t_a)$$

- Using a load balanced data distribution, i.e., $I_p = M/P$, it holds

$$T_p = (2I_p - 1)t_a + \log P(2(t_{startup} + 1 \cdot t_{data}) + t_a)$$

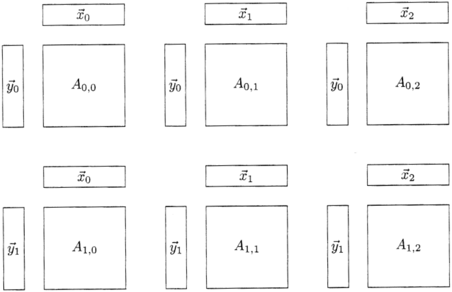
Matrix-Vector Multiplication

- For a given $M \times N$-matrix A and an N-dimension vector x, $y = Ax$ is given by

$$y_m = \sum_{n=0}^{N-1} a_{m,n} x_n = \langle a^m, x \rangle$$

  where a^m is the m-th row of A
- Use a **load-balanced data distribution** on $R = P \times Q$ processes in an array topology to store A (no overlap!)
- x must have the same data distribution as the **rows** of A
- Similarly, y has the same data distribution as the **columns** of A

Data Distribution



Attention

There is additional memory needed to store x and y: In the sequential version, $M + N$ memory locations are necessary. In the parallel version, $QM + PN$ locations are necessary

Algorithm

- Once the data is distributed, all individual components y_m can be computed in parallel using the Recursive Doubling Algorithm
- Do not forget to block (combine) data exchanges to avoid excessive startup times!
  - Use **one** recursive doubling exchange for all components
  - Remember $t_{startup} \gg t_{data}$
- The performance analysis is similar to the one given before

Matrix-Matrix Multiplication

- Let A be an $M \times N$-matrix, B an $N \times K$-matrix. Then $C = AB$ is

$$c_{m,k} = \sum_{n=0}^{N-1} a_{m,n} b_{n,k} = \langle a^m, b_k \rangle$$

where a_m is the m-th row of A and b_b is the k-th column of B

- Why not simply generalize matrix-vector multiplication?
  - We would need excessive additional memory $(QMK + PNK)$

Matrix-Matrix Multiplication – Potential

Observations

If $M = N = K$, the matrix multiplication requires M^3 operations on M^2 data. So we expect a good potential for parallelization.

In the following, we assume $M = N = K$ and $R = P \times P$

A Simple Algorithm

- Assume that we have $P = M$
- Assign (a^m, b_k) to process (m, k)
- Compute $c_{m,k}$ on process (m, k)
- Advantages
  - Computation time $t_{comp} = (2M - 1)t_a$
  - No communication at the computation stage
- Drawbacks
  - A lot of communication for initialization
  - Or, large memory requirements

Cannon's Algorithm

- In a first step, assume $P = M$
- Data distribution: process (m, n) holds $a_{m,n}, b_{m,n}, c_{m,n}$

$$
\overset{\text{A}}{\begin{array}{|c|c|c|}\hline \phantom{a_{20}} & \phantom{a_{21}} & \phantom{a_{22}} \\ \hline & & \\ \hline a_{20} & a_{21} & a_{22} \\ \hline \end{array}}
\times
\overset{\text{B}}{\begin{array}{|c|c|c|}\hline \phantom{b_{01}} & b_{01} & \phantom{b_{01}} \\ \hline & b_{11} & \\ \hline & b_{21} & \\ \hline \end{array}}
=
\overset{\text{C}}{\begin{array}{|c|c|c|}\hline \phantom{c_{21}} & \phantom{c_{21}} & \phantom{c_{21}} \\ \hline & & \\ \hline & c_{21} & \\ \hline \end{array}}
$$

Cannon's Algorithm – Partial Products

- On the "diagonal" processors (m, n), parts of the sum

$$c_{m,n} = a_{m,1}b_{1,m} + \cdots + a_{m,m}b_{m,n} + \cdots + a_{m,M}b_{M,m}$$

are available

A

a_{10}	a_{11}	a_{12}

$\times$

B

	b_{01}	
	b_{11}	
	b_{21}	

$=$

C

	c_{11}	

- Shifting the rows of A cyclic to the right and those of B cyclic downwards provides the next term

A

a_{12}	a_{10}	a_{11}

$\times$

B

	b_{21}	
	b_{01}	
	b_{11}	

$=$

C

	c_{11}	

- Repeating this cyclic exchange ones again completes the calculation of the diagonal elements

The Outer Diagonal Elements

- Consider element $c_{0,1}$

$$c_{0,1} = a_{0,0}b_{0,1} + a_{0,1}b_{1,1} + \dots$$

- While $a_{0,1}$ is available, $b_{1,1}$ is not
- This can be corrected by shifting the second column of B (cyclically) upwards
- In order not to change the indices on the diagonal, the second row of A must be shifted (cyclically) to the left

Initial Data Distribution

- More general, the matrices A and B must be initialized as follows
  - The m-th row of A is shifted cyclically $m-1$ positions to the left
  - The n-th column of B is shifted cyclically $n-1$ positions upwards

A

a_{00}	a_{01}	a_{02}
a_{11}	a_{12}	a_{10}
a_{22}	a_{20}	a_{21}

B

b_{00}	b_{11}	b_{22}
b_{10}	b_{21}	b_{20}
b_{20}	b_{01}	b_{12}

Algorithm

1. Initially, process (p, q) has elements $a_{p,q}, b_{p,q}$
2. Shift the rows of A and the columns of B into the structure described above
3. For $k = 1, \dots, N$
    1. Multiply the local values on each process
    2. Shift the rows of A and the columns of B cyclically by one process
4. If necessary: undo step 1

Remark

- If the number of processors is much less than N^2, a blocked version will do
- The last shift in step 3.2 can be omitted

Efficiency Analysis

- Assume: $R = P \times P$ and P divides N
- **Step 2**: Use red-black communication

$$t_2 = 4(t_{startup} + I_p^2 t_{data})$$

- **Step 3.1**: $t_{3,1} = 2I_p^3 t_a$
- **Step 3.2**: $t_{3,2} = 4(t_{startup} + I_p^2 t_{data})$

$$T_R = T_{P\times P} = 2PI_p^3 t_a + 4(P+1)(t_{startup} + I_p^2 t_{data})$$
$$T_S^* = 2N^3 t_a$$

Speedup

$$
\begin{aligned}
S_R &= \frac{2N^3 t_a}{2PI_p^3 t_a + 4(P+1)(t_{startup} + I_p^2 t_{data})} \\
&= \frac{2N^3 t_a}{2P(N/P)^3 t_a + 4(P+1)(t_{startup} + (N/P)^2 t_{data})} \\
&= R \frac{1}{1 + 2\frac{R(\sqrt{R}+1)}{N^3}\frac{t_{startup}}{t_a} + 2\frac{P+1}{N}\frac{t_{data}}{t_a}}
\end{aligned}
$$

Other Algorithms

There are a number of other algorithms available which have a better efficiency than Cannon's algorithm:

- Strassen's algorithm reduces the complexity of matrix multiplication from $\mathcal{O}(N^3)$ to $\mathcal{O}(N^{2.81\cdots})$. This algorithm can be refined to complexity $\mathcal{O}(N^{2.376\cdots})$
- A refined communication strategy on a differently organized processor topology was proposed by Dekel, Nassimi and Sahni (DNS algorithm)

Google's Problem

The Birth of Google

Lawrence Page, Sergey Brin, Rajeev Motwani, Terry Winograd:
The PageRank citation ranking: Bringing order to the web.
Technical Report SIDL-WP-1999-0120,
`https://api.semanticscholar.org/CorpusID:1508503`

A search engine faces two problems

1. Find all web pages which contain the search phrase (more general: satisfy some search criterion)
2. Among all these pages, find the most relevant

- Web crawler
- **PageRank citation ranking**

Page Ranking

- Every web page has a number of forward links (pointers to other web pages) and a number of backlinks (web pages pointing to the given page)
- A web page seems to be more important if the number of backlinks is large
- Simple counting the number of backlinks may not be sufficient: A backlink with a high importance should have a higher weight
- This is a recursive definition: A page is important if the backlinks are important.
- But what is the root of this recursion? The web does not have a root :(

Definition of PageRank

- Let m be a webpage
- Let F_m be the set of pages m points to
- Let B_m be the set of pages that point to m
- Let $N_m = |F_m|$ be the number of forward links, and c be a normalization factor
- Then we define PageRank

$$R_m = c \sum_{n \in B_m} \frac{R_n}{N_n}$$

- Actually, this definition is slightly simplified...

Reformulation of PageRank

- Define a matrix A by

$$a_{mn} = \begin{cases} 1/N_n, & \text{If there is a link from } m \text{ to } n, \\ 0, & \text{otherwise} \end{cases}$$

- Let $R = (R_1, \ldots, R_M)^T$ be the PageRank vector. This it holds

$$R = cAR$$

PageRank

Definition

The PageRank of a given connectivity matrix A is an eigenvector R corresponding to the largest eigenvalue c of A

Remark: Actually, A is a **sparse matrix**, i.e. a matrix which contains of mostly zero entries. For the time being, we neglect this property. In connection with algorithms on graphs, we will consider sparse matrices.

The Power Method For Eigenvalue Problems

- Given a square matrix A and an initial guess $x^{[0]} \neq 0$
- Form the sequence $x^{[k+1]} = Ax^{[k]}, k = 0, 1, \dots$
- It can be shown that, for many matrices A and almost all initial guesses $x^{[0]}$ the sequence $\{x^{[k]}/\|x^{[k]}\|\}$ converges towards an eigenvector to the largest eigenvalue c of A
- An estimation c_k of c can be obtained by $c_k = \frac{\langle x^{[k+1]}, x^{[k]} \rangle}{\langle x^{[k]}, x^{[k]} \rangle}$

Algorithm

```
% Let x = x0 be chosen
err = tol;
c = 0;
while err >= tol*abs(c)
  nrm = 1/sqrt(<x,x>);  % recursive doubling
  x = nrm*x;
  y = A*x;              % matrix-vector mult
  cnew = <y,x>;         % recursive doubling
  err = abs(cnew-c);
  c = cnew;
  x = y;                % vector transposition
end
```

Vector Transposition

- Assume a $P \times Q$ processor mesh and A distributed accordingly
- Let y be distributed in the same way as the columns of A
- The simple case: $P = Q$ and row column data distributions are equal
  - `MPI_Alltoall` can be used to transpose y
- The general case is much more involved. Essentially it is equivalent to one recursive doubling step

Optimization of Parallel Algorithms

- In a sequential algorithm, the program will run faster if the consecutive steps are optimized individually
- Hence, optimizing a sequential algorithm is a **local** problem
- In a parallel environment, for a given algorithm the execution time depends both on the computation time and **the data distribution**
- A certain data distribution may be optimal at one step of the algorithm while suboptimal for another
- **Thus, optimizing a parallel program is a global problem!**