Lecture 1 - Introduction

Moore's Law: 2x #transistors/chip every 1.5 years
=> 2x #cores/chip every 2 years
=> clock speed - not increasing (possibly decreasing)

Challenges:
- -> Hardware - processor technology + interconnection network
- -> Software - parallel compilers + grid computing environment
- -> Methods & Algorithms

Lecture 2 - Parallel Computer Architecture

Processor/Node Architecture

Von Neumann - program and data in same main memory

CPU - Central Processing Unit
- -> fetch, decode, execute program instructions
- -> fetch data
- -> store results

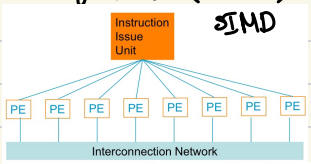
Graphics Controller
- -> render display images

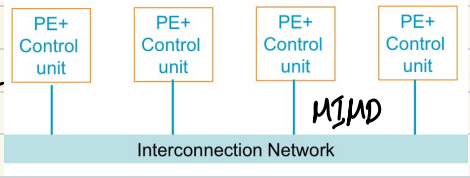
GPU - Graphics Processing Unit
- -> perform complex graphics operations

CPU - several cores vs. GPU - thousands of cores (SIMD)

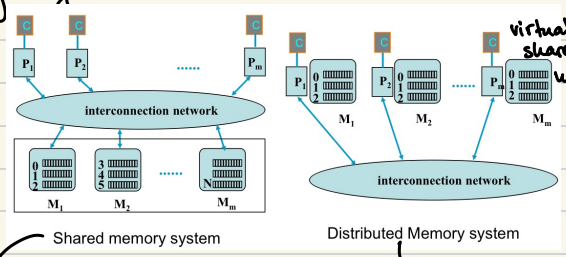
Flynn's Taxonomy

1) Single Instruction, Single Data
2) Multiple Instruction, Single Data
3) Single Instruction, Multiple Data
4) Multiple Instruction, Multiple Data





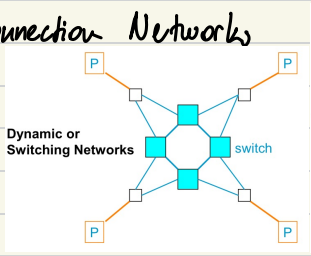
Memory Organization

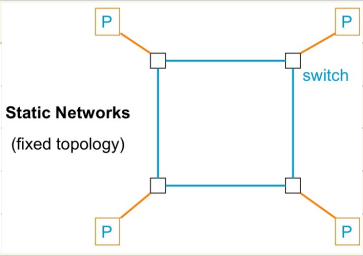


UMA - Uniform Memory Access

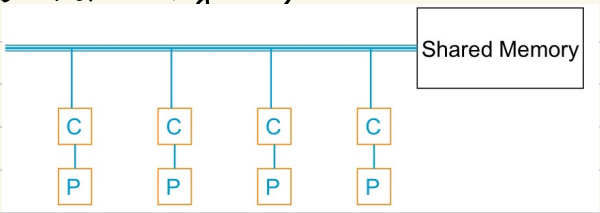
NUMA - Non-uniform Memory Access

Interconnection Networks

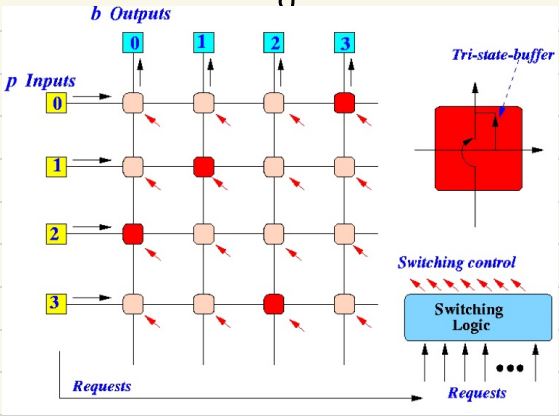




Bus Network (dynamic)



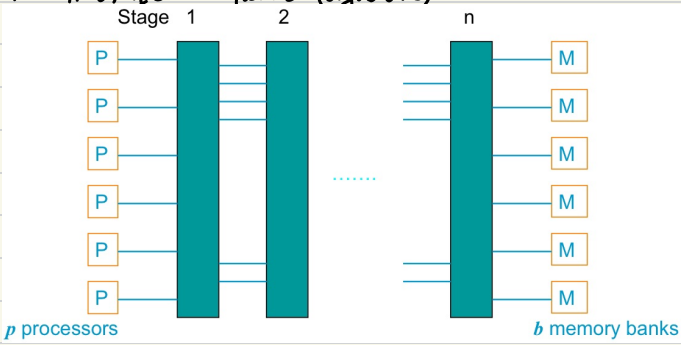
Crossbar Network (dynamic)



cost = #switches

$\theta(b \cdot p) \rightarrow \Omega(p^2)$, $b > p$

Multi-stage Network (dynamic)



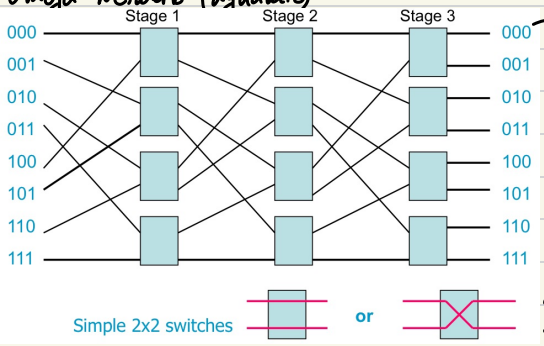
switches = $p \cdot \log_2 p$

$b = p$, $n = \log_2 p$

perfect shuffle:

$$j = \begin{cases} 2i, & 0 \le i \le \frac{p}{2} - 1 \\ 2i+1 - p, & \frac{p}{2} \le i \le p-1 \end{cases}$$

Omega Network (dynamic)



$\beta_1 \beta_2 .. \beta_k$ at stage k:

$\beta_k = 0$ → upper link of switch

$\beta_k = 1$ → lower link of switch

$d = \log p$, conn = 2

bisection = $\frac{p}{2}$

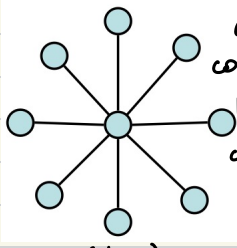
processors: p

stages: $\log p$

switches per stage: $\frac{p}{2}$

switches: $\frac{p}{2} \log p$

Star Network (static)



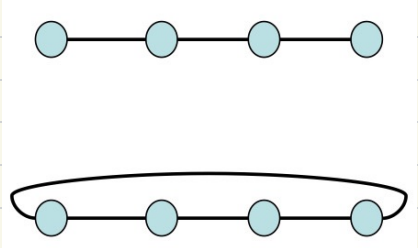
$d = 2$

conn = 1

bisect = 1

cost = $p - 1$

Link Arrays and Links (static)



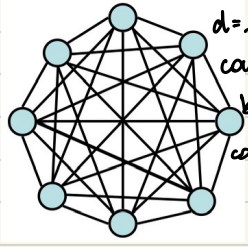
$d = \lfloor \frac{p}{2} \rfloor$

conn = 2

bisect = 2

cost = p

Completely Connected (static)



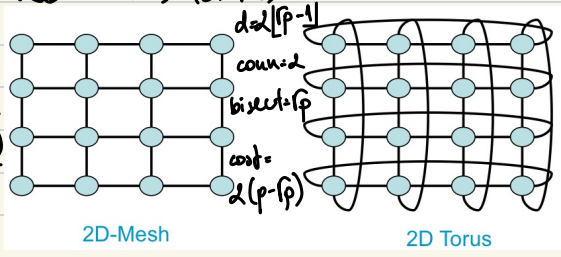
$d = 1$

conn = $p - 1$

bisect = $\frac{p^2}{4}$

cost = $\frac{p(p-1)}{2}$

2D Meshes (static)



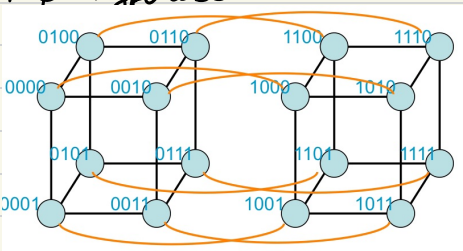
$d = 2(\lceil \sqrt{p} \rceil - 1)$

conn = 2

bisect = $\lceil \sqrt{p} \rceil$

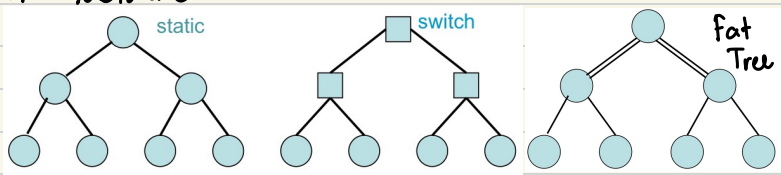
cost = $2(p - \lceil \sqrt{p} \rceil)$

4-D Hypercube



$p = 2^d$, d - dimension
connections/processor = $\log p$
max distance = $\log p$
$d = \log p$, conn = $\log p$, bisection = $\frac{p}{2}$
cost = $\frac{p \log p}{2}$

Tree Network

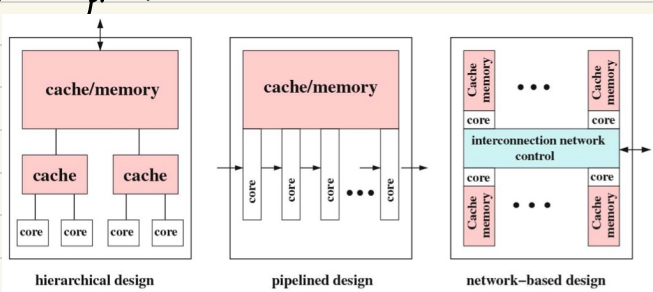


$d = 2 \log p$
conn = 1
bisection = 1
cost = $p - 1$

Evaluation Criteria Static Networks

diameter = max distance
connectivity = min # links removed to split in two disjoint parts
bisection width = min # links removed to split in two equal parts
cost = total # links

Multicore processor



Lecture 3 - Process, Thread & OpenMP

Parallelization of programs

1) decomposition of computations (partitioning)
2) assignment of tasks (scheduling)
3) mapping of processes to physical processors

Data dependencies

-> flow (true) dependency - output of S_1 is input of S_2
-> anti-dependency - input of S_1 overwritten by S_2
-> output dependency - S_1, S_2 store outputs at same place

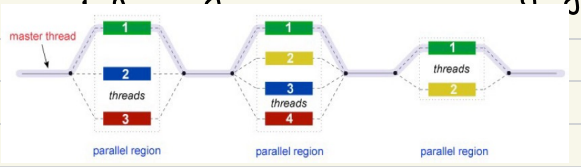
flow dependency	anti dependency	output dependency
I_1: $R_1 \leftarrow R_2+R_3$	I_1: $R_1 \leftarrow R_2+R_3$	I_1: $R_1 \leftarrow R_2+R_3$
I_2: $\underline{R_5} \leftarrow R_1+R_4$	I_2: $R_2 \leftarrow R_4+R_5$	I_2: $R_1 \leftarrow R_4+R_5$

Data parallelism - distribute the data structure evenly among processors and each perform the operation on its assigned elements
process - program in execution with own local address space (distributed mem.)
thread - sequence of instructions (shared memory)

Multi-threading

-> timesliced - processor/core switch threads after time interval elapsed
-> simultaneous - several threads executed at same time by multiple processors/cores

OpenMP - programming model for shared memory systems



for-construct schedule clause:

- parameter chunk - divide iterations into blocks of size chunk
- -> static - blocks are assigned to threads in a round-robin fashion (default)
  - default chunk = #iterations / #threads or #threads = #blocks
- -> dynamic - blocks are assigned to threads in order in which threads finish previous blocks
  - default chunk = 1
- -> guided - block size starts with implementation-defined value, then decreases exponentially to chunk

Data Scoping:

- -> private - variables local to each thread (new uninitialized instance is created)
  - loop control, non-static local to Parallel Regions (variables)
- -> firstprivate - initialization with Master's value
- -> lastprivate - value of last loop iteration is written back to Master
- -> shared - variables shared among all threads
  - static variables, general default for Parallel Region

Critical Region - executed by all threads, but only by one thread simultaneously (Mutual Exclusion)

barrier - wait until all threads have reached the barrier

worksharing constructs - implicit barrier at the end

explicit nowait clause - remove implicit barrier at the end

reduction(operation: list) - +, -, *, &, |, &&, ||, ^, max, min

Cache Efficiency

C - row-major storage

a[8][8]: a[0,0], a[0,1], a[0,2], ..., a[0,7] (1st row), a[1,0], a[1,1], ..., a[1,7] (2nd row), a[2,0] ... a[7,7]

64B cache-line size => fetch a[0,0] => a[0,0], a[0,1], ..., a[0,7] (8x8B double) fetched

temporal locality - data is used frequently

spatial locality - consecutive data on same cache-line

False Sharing
- -> different threads use elements of same cache-line
- -> one thread writes to cache-line

Lecture 4 - Performance Analysis

user CPU time of A - time CPU spends executing A
system CPU time of A - time CPU spends executing A + time of operating system
wall-clock time (waiting time) of A - time elapsed from beginning to end (incl. I/O, execution of others)
parallel execution time of A - incl. thread synchronization and communication time

$T_{u\_cpu}(A) = n_{cycle}(A) \cdot t_{cycle}$ → clock cycle time of CPU
↳ number of CPU cycles for executing A
memory access: $T_{u\_cpu}(A) = (n_{cycle}(A) + n_{mm\_cycle}(A)) \cdot t_{cycle}$
number of additional cycles caused by memory access of A

cache miss - reading content not residing in cache ⇒ extra delay loading cache line from memory
$n_{read\_cycles}(A) = n_{read\_op}(A) \cdot r_{read\_miss}(A) \cdot n_{miss\_cycle}$
#read operations ↳ read miss rate ↳ # cycles to load cache line
$T_{u\_cpu}(A) = n_{instr}(A) \cdot (CPI(A) + f_{rw\_op} \cdot r_{miss}(A) \cdot n_{miss\_cycle}) \cdot t_{cycle}$
Clock cycles Per Instruction ↳ percentage of instructions executing read/write operations

T_s - Serial execution time
T_p - Parrallel execution time
$T_o = p \cdot T_p - T_s$ - Overhead function
$S = T_{serial\_best} / T_p$ - True Speed up
$S = T_{p=1} / T_p$ - Relative Speed up
$S_{linear} = p$ - Linear Speedup
$S_{sublinear} < p$ - Sublinear Speedup
$S_{superlinear} > p$ - Superlinear Speedup

Superlinear Speedup, 1 FLOP per memory access
Cache: 10ns
Main Memory: 100ns
$t_{access} = p_{cache} \cdot 10 + (1 - p_{cache}) \cdot 100$
sequential: $p_{cache} = 80\%$ ⇒ $t_{access} = 28ns$ ⇒ speed ≈ 36MFLOP
parallel: $p_{cache} = 90\%$ ⇒ $t_{access} = 19ns$ ⇒ speed per processor ≈ 53MFLOP
$S = \frac{2.53}{36} \approx 3$

Adding n numbers on n processors

$W(n) = n + \sum_{i=1}^{\log n} n/2^i + 1 \approx 2n + 1 = \Theta(n)$

$T(n) = \log n + 1 = \Theta(\log n)$

Adding n numbers on p processors

$T_p = \frac{n-1}{p} + 2\log p$

$\lim_{n\to\infty} S_p(n) = \lim_{n\to\infty} \frac{n-1}{\frac{n-1}{p} + 2\log p} = p$

$E = \frac{S}{p}: \quad E = \frac{1}{1 + \frac{2p\log p}{n-1}}$

Input : $n = 2^k$ numbers stored in array A

Output : The sum $S = \sum_{i=1}^{n} A[i]$

1. **for** $1 \le i \le n$ **do parallel**
   $B[i] := A[i]$
2. **for** $h = 1$ **to** $\log n$ **do sequential**
   for $1 \le i \le n/2^h$ **do parallel**
   $B[i] := B[2i-1] + B[2i]$
3. $S := B[1]$

Amdahl's Law

$W = T_{ser} + T_{par}$, $f = T_{ser}/W$ | $S = \frac{p}{1 + f(p-1)}$, $f = \frac{\frac{1}{S} - \frac{1}{p}}{1 - \frac{1}{p}}$

f - constant serial fraction

$\Rightarrow$ maximum speedup $S = \frac{1}{f}$

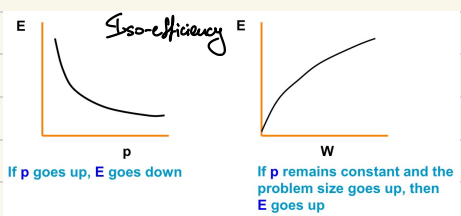
Scalability

$E = \frac{S}{p} = \frac{T_s}{p \cdot T_p} = \frac{1}{1 + \frac{T_o}{T_s}}$

Gustafson's law: $\lim_{n\to\infty} S_p(n) = \lim_{n\to\infty} \frac{n-1}{\frac{n-1}{p} + 2\log p} = p$

Amdahl's Law: $\lim_{p\to\infty} S = \frac{p}{1 + f(p-1)} = \frac{1}{f}$

Problem size $W(n)$ - #basic computation steps in best sequential algorithm



Iso-efficiency

$T_o = pT_p - W$, $S = \frac{W}{T_p} = \frac{W \cdot p}{W + T_o} \Rightarrow E = \frac{1}{1 + T_o/W}$

$W = \left(\frac{E}{1-E}\right) \cdot T_o = K \cdot T_o(W, p)$ with $K = \frac{E}{1-E}$

$W = T_s = n^3 + n$ | $T_o = pT_p - W = 2np\log p - n$

$T_p = \frac{n^3}{p} + 2n\log p$ | $T_o/W \approx 2p\log p / n^2 \Rightarrow$ const. E if $p = x$, $n = \sqrt{2}\sqrt[4]{p^3}x$

Parallel scalar (dot) product: $a \cdot b = \sum_{i=1}^{n} a_i b_i \Rightarrow \frac{n}{p}$ scalar products across p processors

$T(p,n) = \left(\frac{2n}{p} - 1\right)\alpha + \left(\frac{p}{2} - 1\right)(\alpha + \beta) \approx \frac{2n}{p}\alpha + \frac{p}{2}(\alpha + \beta)$

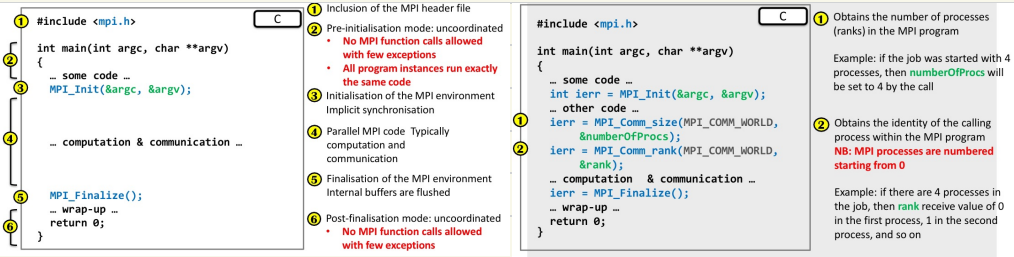
$W = T_s = (2n-1)\alpha \approx 2n\alpha$ | Iso-efficiency: $\frac{T_o}{W} = \frac{p^2(\alpha+\beta)}{4n\alpha} = \text{constant}$

$T_o = pT(p,n) - T_s = \frac{p^2}{2}(\alpha + \beta)$

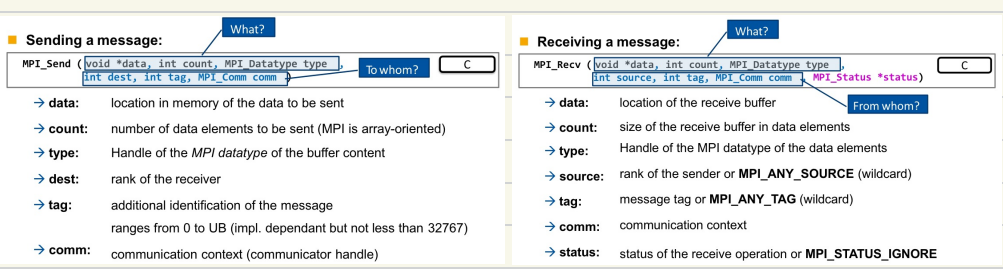
Lecture 5 - MPI

MPI program = set of processes each with own local data
Principle: MPMD (Multiple Program Multiple Data)
Practice: SPMD (Single Program Multiple Data)
Data exchange through messaging over the network (explicit (MPI) or implicit (PGAS))

Process - running in-memory instance of an executable file
- -> executable code
- -> one or more threads of execution (sharing memory address space)
- -> memory: data, heap, stack, processor state (CPU registers and flags)
- -> operating system context
- -> PID



Explicit messaging:
- -> send and receive primitives (operations)
- -> known addresses of both sender and receiver
- -> specification of what has to be sent/received



MPI data type	C data type
MPI_CHAR	char
MPI_SHORT	short
MPI_INT	int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_UNSIGNED_INT	unsigned int

send = receive ✓
send > receive ✗
C: MPI_Status status
-> status.MPI_SOURCE message source rank
-> status.MPI_TAG message tag
-> status.MPI_ERROR receive status code

```
MPI_Sendrecv (void *senddata, int sendcount, MPI_Datatype sendtype,
              int dest, int sendtag, void *recvdata, int recvcount,
              MPI_Datatype recvtype, int source, int recvtag,
              MPI_Comm comm, MPI_Status *status)
```

	Send	Receive
Data	senddata	recvdata
Count	sendcount	recvcount
Type	sendtype	recvtype
Destination	dest	-
Source	-	source
Tag	sendtag	recvtag
Communicator	comm	comm
Receive status	-	status

-> send 1 message + receive 1 message (no deadlock)
-> send and receive buffer shouldn't overlap

Non-blocking:

```
MPI_Isend (void *data, int count, MPI_Datatype dataType,
       int dest, int tag, MPI_Comm comm, MPI_Request *request)
```

```
MPI_Irecv (void *data, int count, MPI_Datatype dataType,
       int source, int tag, MPI_Comm comm, MPI_Request *request)
```

Blocking:

```
MPI_Wait (MPI_Request *request, MPI_Status *status)
```

Test:

```
MPI_Test (MPI_Request *request, int *flag, MPI_Status *status)
```

MPI_Waitany / MPI_Testany
- → Wait for one of the specified requests to complete and free it
- → Test if one of the specified requests has completed and free it if it did

MPI_Waitall / MPI_Testall
- → Wait for all the specified requests to complete and free them
- → Test if all of the specified requests have completed and free them if they have

MPI_Waitsome / MPI_Testsome
- → Wait for any number of the specified requests to complete and free them
- → Test if any number of the specified requests have completed and free these that have

Send Modes

Standard mode

→ The call blocks until the message has either been transferred or copied to an internal buffer for later delivery

Synchronous mode

→ The call blocks until a matching receive has been posted and the message reception has started

Buffered mode

→ The call blocks until the message has been copied to a user-supplied buffer. Actual transmission may happen at a later point

Ready mode (don't use!)

→ The operation succeeds only if a matching receive has already been posted. Behaves as standard send in every other aspect

→ **MPI_Send**	blocking standard send
→ **MPI_Isend**	non-blocking standard send
→ **MPI_Ssend**	blocking synchronous send
→ **MPI_Issend**	non-blocking synchronous send
→ **MPI_Bsend**	blocking buffered send
→ **MPI_Ibsend**	non-blocking buffered send
→ **MPI_Rsend**	blocking ready-mode send
→ **MPI_Irsend**	non-blocking ready-mode send

→ **MPI_Buffer_attach (void *buf, int size)**

→ **MPI_Buffer_detach (void *buf, int *size)**

```
MPI_Abort (MPI_Comm comm, int errorcode)
```

```
double MPI_Wtime ()
```

```
MPI_Get_processor_name (char *name, int *resultlen)
```

MPI_Reduce

MPI_Reduce(void* send_data, void* recv_data, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm communicator)

The reduction operations defined by MPI include:

- MPI_MAX - Returns the maximum element.
- MPI_MIN - Returns the minimum element.
- MPI_SUM - Sums the elements.
- MPI_PROD - Multiplies all elements.
- MPI_LAND - Performs a logical *and* across the elements.
- MPI_LOR - Performs a logical *or* across the elements.
- MPI_BAND - Performs a bitwise *and* across the bits of the elements.
- MPI_BOR - Performs a bitwise *or* across the bits of the elements.
- MPI_MAXLOC - Returns the maximum value and the rank of the process that owns it
- MPI_MINLOC - Returns the minimum value and the rank of the process that owns it.

Lecture 6 - Parallel Algorithms for Systems of Linear Equations

Gaussian Elimination

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n = b_1 \\ \vdots \\ a_{i1}x_1 + a_{i2}x_2 + \dots + a_{in}x_n = b_i \\ \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nn}x_n = b_n \end{cases}$$

For $k=1,\dots n$, eliminate the k-*th* lower column of $A^{(k)}$

$$l_{ik} = a_{ik}^{(k)} / a_{kk}^{(k)}, \qquad i = k+1, \dots, n.$$

Update $A^{(k)}$:

$$a_{ij}^{(k+1)} = a_{ij}^{(k)} - l_{ik} a_{kj}^{(k)} \quad \text{for } k < j \le n \text{ and } k < i \le n$$
$$b_i^{(k+1)} = b_i^{(k)} - l_{ik} b_k^{(k)}$$

Backward substitution: $x_n, x_{n-1}, \dots, x_1$

$$x_k = \frac{1}{a_{kk}^{(n)}} \left(b_k^{(n)} - \sum_{j=k+1}^{n} a_{kj}^{(n)} x_j \right)$$

$A^{(1)}=A$, for $k \ge 2$:

$$A^{(k)} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1,k-1} & a_{1k} & \cdots & a_{1n} \\ 0 & a_{22}^{(2)} & \cdots & a_{2,k-1}^{(2)} & a_{2k}^{(2)} & \cdots & a_{2n}^{(2)} \\ \vdots & \ddots & \ddots & \vdots & \vdots & & \vdots \\ \vdots & & \ddots & a_{k-1,k-1}^{(k-1)} & a_{k-1,k}^{(k-1)} & \cdots & a_{k-1,n}^{(k-1)} \\ \vdots & & & 0 & a_{kk}^{(k)} & \cdots & a_{kn}^{(k)} \\ \vdots & & & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & \cdots & 0 & a_{nk}^{(k)} & \cdots & a_{nn}^{(k)} \end{bmatrix}.$$

LU decomposition, $A=LU$

$U = A^{(n)}$, $L = \begin{bmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{n1} & l_{n,n-1} & 1 \end{bmatrix}$, $Ax=b$, $y=Ux$

solve y: $Ly=b$, forward substitution

solve x: $Ux=y$, backward substitution

Recursive Doubling

$$A = \begin{pmatrix} b_1 & c_1 & & & 0 \\ a_2 & b_2 & c_2 & & \\ & a_3 & b_3 & \ddots & \\ & & \ddots & \ddots & c_{n-1} \\ 0 & & & a_n & b_n \end{pmatrix}$$

Solve $A\underline{x} = \underline{y}$

Step k:

1. Compute $l_{ik} := a_{ik}^{(k)} / a_{kk}^{(k)}$ for $i = k+1, \dots, n$.
2. Subtract l_{ik} times the kth row from the rows $i = k+1, \dots, n$, i.e., compute

$$a_{ij}^{(k+1)} = a_{ij}^{(k)} - l_{ik} \cdot a_{kj}^{(k)} \quad \text{for } k \le j \le n \text{ and } k < i \le n.$$
$$y_i^{(k+1)} = y_i^{(k)} - l_{ik} \cdot y_k^{(k)}$$

Considering the 3 neighboring equations *i-1*, *i*, and *i+1 (i=3,4,…,n-2)*:

$$\begin{aligned} a_{i-1}x_{i-2} + b_{i-1}x_{i-1} + c_{i-1}x_i &= y_{i-1} \\ a_i x_{i-1} + b_i x_i + c_i x_{i+1} &= y_i \\ a_{i+1}x_i + b_{i+1}x_{i+1} + c_{i+1}x_{i+2} &= y_{i+1} \end{aligned}$$

Use equation $i-1$ to eliminate x_{i-1} from the i^{th} equation and use equation $i+1$ to eliminate x_{i+1} from the i^{th} equation. (Step 1)

The resulting new equation i is: $a_i^{(1)} x_{i-2} + b_i^{(1)} x_i + c_i^{(1)} x_{i+2} = y_i^{(1)}$

$$A^{(1)} = \begin{pmatrix} b_1^{(1)} & 0 & c_1^{(1)} & & & 0 \\ 0 & b_2^{(1)} & 0 & c_2^{(1)} & & \\ a_3^{(1)} & 0 & b_3^{(1)} & \ddots & \ddots & \\ & a_4^{(1)} & \ddots & \ddots & \ddots & c_{n-2}^{(1)} \\ & & \ddots & \ddots & \ddots & 0 \\ 0 & & & a_n^{(1)} & 0 & b_n^{(1)} \end{pmatrix}$$

In step 2, consider the equations $i-2$, i, $i+2$ of the $A^{(1)}x = y^{(1)}$ for $i = 5, 6, \dots, n-4$. Use equation $i-2$ to eliminate x_{i-2} from the ith equation and use equation $i+2$ to eliminate x_{i+2} from the i^{th} equation.

$$A^{(2)} = \begin{pmatrix} b_1^{(2)} & 0 & 0 & 0 & c_1^{(2)} & & & 0 \\ 0 & b_2^{(2)} & & & & c_2^{(2)} & & \\ 0 & & \ddots & & & & \ddots & \\ 0 & & & \ddots & & & & c_{n-4}^{(2)} \\ a_5^{(2)} & & & & \ddots & & & 0 \\ & a_6^{(2)} & & & & \ddots & & 0 \\ & & \ddots & & & & \ddots & 0 \\ 0 & & & a_n^{(2)} & 0 & 0 & 0 & b_n^{(2)} \end{pmatrix}$$

Step k: Considering equations $i - 2^{k-1}$, i, $i + 2^{k-1}$:

Use equation $i - 2^{k-1}$ to eliminate x_{i-2k-1} from the i^{th} equation and use equation $i + 2^{k-1}$ is used to eliminate x_{i+2k-1} from the i^{th} equation.

Cyclic Reduction

1. Elimination phase: For $k = 1, \ldots, \lfloor \log n \rfloor$ compute $a_i^{(k)}$, $b_i^{(k)}$, $c_i^{(k)}$ and $y_i^{(k)}$ with $i = 2^k, \ldots, n$ and step-size 2^k. The number of equations of the form (8.26) is reduced by a factor of $1/2$ in each step. In step $k = \lfloor \log n \rfloor$ there is only one equation left for $i = 2^N$ with $N = \lfloor \log n \rfloor$.

$$a_i^{(k)} x_{i-2^k} + b_i^{(k)} x_i + c_i^{(k)} x_{i+2^k} = y_i^{(k)} \tag{8.26}$$

2. Substitution phase: For $k = \lfloor \log n \rfloor, \ldots, 0$ compute x_i according to Eq. (8.26) for $i = 2^k, \ldots, n$ with step-size 2^{k+1}:

$$x_i = \frac{y_i^{(k)} - a_i^{(k)} \cdot x_{i-2^k} - c_i^{(k)} \cdot x_{i+2^k}}{b_i^{(k)}} . \tag{8.27}$$

Lecture 7 - Parallel Iterative Algorithms for Linear Systems

$A = M - N$ s.t. M^{-1} is easily computed

$M x^{k+1} = N x^k + b$ or $x^{k+1} = C x^k + d$, $C = M^{-1}N$, $d = M^{-1}b$

$A = D - L - R$, $D = diag(A)$

Jacobi: $D x^{k+1} = (L+R) x^k + b$, $C_{Ja} = D^{-1}(L+R) = D^{-1}(A-D)$

Gauss-Seidel: $(D-L) x^{k+1} = R x^k + b$, $C_{GS} = (D-L)^{-1} R$

JOR: $A = \frac{D}{\omega} - L - R - \frac{1-\omega}{\omega} D$

SOR: $(D - \omega L) x^{k+1} = (1-\omega) D x^k + \omega R x^k + \omega b$

Jacobi, $A\vec{x} = \vec{b}$

$\vec{x}^k = D^{-1} \cdot (\vec{b} - (A-D) \cdot \vec{x}^{k-1}) = (I - D^{-1} \cdot A) \cdot \vec{x}^{k-1} + D^{-1} \cdot \vec{b}$

Design Issues

→ Optimality: minimal parallel execution time

↳ Trade-off: maximal load balance vs. minimal communication overhead

→ Data Locality = $\frac{\text{computing time b/w 2 consecutive communications}}{\text{communication time (overhead)}}$

→ Grain Size: computing time b/w 2 consecutive communications

Data Locality Ratios

$A, B \in \mathbb{R}^{n \times n}$, $\vec{x} \in \mathbb{R}^n$, c - time of 1 computation, τ - time of communication of 1 number

Matrix-vector product $A\vec{x}$:

$$\frac{\frac{n}{p}(2n-1)c}{(\frac{n^2}{p}+n)\tau} \approx \frac{2c}{\tau}$$

Matrix-matrix product $A \cdot B$:

$$\frac{\frac{n^2}{p}(2n-1)c}{n^2(\frac{1}{p}+1)\tau} \approx \frac{2nc}{p\tau}$$

Gauss-Seidel

$\vec{x}^k = D^{-1}\vec{b} - D^{-1} \cdot (L\vec{x}^k + R\vec{x}^{k-1})$

D-diagonals, L-lower triangular, R-upper triangular