

Chapter 1 - Motivation and History

Data parallelism: same operation to different elements of the data set (e.g. vector addition)

Function parallelism: different operations to different data elements

1.5. A task can be divided into m subtasks.

How much time is needed for a m -stage pipeline to process n tasks?

-> $n + m - 1$

1.6. Printing n pages takes $5+10+3+n$ s. What is the minimum capacity of the feeder tray so that asymptotic throughput reaches 40 pages/min?

$$40 \frac{\text{pages}}{\text{min}} \rightarrow 1.5 \frac{\text{s}}{\text{page}} \quad 1.5 = \frac{18+n}{n} \rightarrow 1.5n = 18+n \rightarrow n = 2 \cdot 18 = \underline{\underline{36}}$$

1.8 Performance increases $\times 10$ every 5 years, how long does it take to double?

$$t: \text{time in years} \quad 10^{\frac{t}{5}} = 2 \quad \frac{t}{5} = \log_{10} 2 \quad t = 5 \log_{10} 2 = \frac{5}{\log_2 10} \approx 1.505$$

1.10: Calculate new problem size that can be solved in the same time when the computer is 100x faster. (Old problem size: 100 000.)

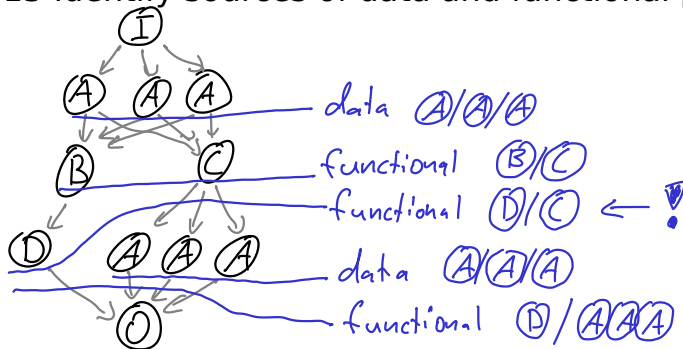
$$a) \Theta(n) \quad \frac{n}{10^5} = 100 \quad n = 100 \cdot 10^5 = 10^7$$

$$b) \Theta(n \log_2 n) \quad \frac{n \log_2 n}{10^5 \log_2 10^5} = 100 \rightarrow \text{solve for } n$$

$$c) \Theta(n^2) \quad \frac{n^2}{(10^5)^2} = 100 \quad n = \sqrt{10^{12}} = 10^6$$

$$d) \Theta(n^3) \quad \frac{n^3}{(10^5)^3} = 100 \quad n = \sqrt[3]{10^{17}}$$

1.13 Identify sources of data and functional parallelism



1.14 preallocate N/p documents vs. put documents in a list and let processors remove documents as fast as they can process them

The advantage of preallocation is that it reduces the overhead associated with assigning documents to idle processors at run-time. The advantage of putting documents on a list and letting processors remove documents as fast as they could process them is that it balances the work among the processors. We do not have to worry about one processor being done with its share of documents while another processor (perhaps with longer documents) still has many to process.

Chapter 2 - Parallel Architectures

Shuffle-exchange: - shuffle link to LeftCycle(i)
- exchange link to xor(i, 1)

Vector computers:

Pipelined vector processor: streams data through pipelined arithmetic units

Processor array: many identical, synchronized arithmetic processing units

Multiprocessors: multiple-CPU computer with shared memory

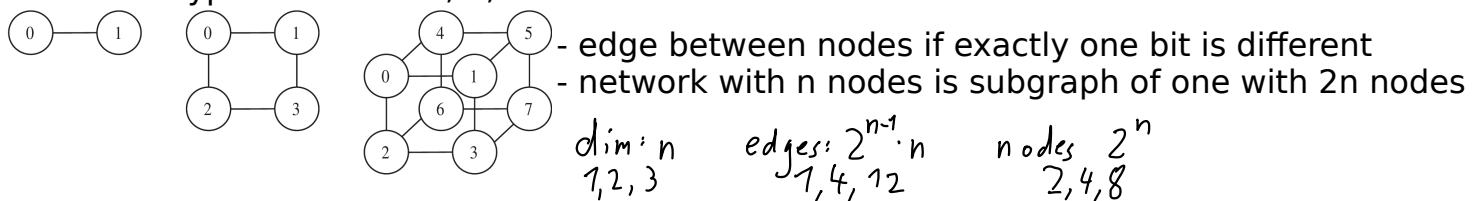
centralized (bus, same memory for all CPUs) or distributed (interconnection network)

Multicomputer: distributed memory multiple CPU computer, message passing

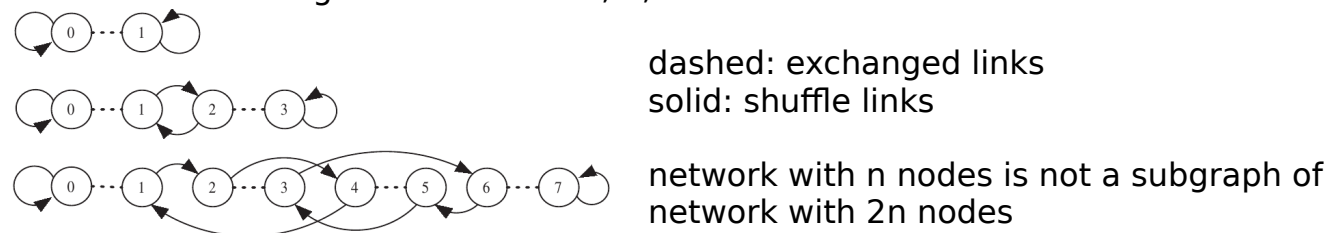
asymmetrical: one front-end computer and many back-end computers

symmetrical: all nodes are equal

2.1. Draw hypercube with 2, 4, 8 nodes



2.8. Shuffle-exchange network with 2, 4, 8 nodes



2.13. Why are processor arrays well suited for executing data-parallel programs?

-> every processing element performs the same operation, but with different data

2.14. Processor array with 8 processing elements, each 10M integer ops/s. Determine performance for adding two vectors of size 1 to 50.

(According to the solution, the answer should be integer ops/s, not vector ops/s!)

$$\text{performance} = \frac{\text{number of ops}}{\text{time}} = \frac{n \text{ ops}}{\frac{\lceil n/8 \rceil}{10 \text{ million}} \text{ s}} = \frac{10 \cdot n}{\lceil n/8 \rceil} \text{ million ops/s}$$

2.15. Processor array, case statement with k cases.

a) Efficiency if each case contains same number of elements

-> The tricky thing is that the processor array always executes the same instructions on all elements and that means that for a case statement, it will be really inefficient.

$$\text{efficiency} = \frac{\text{sequential execution time}}{\text{processors used} \times \text{parallel execution time}} = \frac{n}{n \times k} = \frac{1}{k}$$

b) Efficiency if case i has I_i instructions and probability P_i

$$\text{sequential time: } n \times \text{average time} = n \sum_{i=1}^k P_i I_i$$

$$\text{efficiency} = \frac{n \sum_{i=1}^k P_i I_i}{n \times \sum_{i=1}^k I_i} = \frac{\sum_{i=1}^k P_i I_i}{\sum_{i=1}^k I_i}$$

2.16 Why are large data and instruction caches desirable in multiprocessors?

Large caches reduce the load on the memory bus, enabling the system to utilize more processors efficiently.

2.17 Why is the number of processors in a centralized multiprocessor limited to a few dozen?

-> All processors communicate over the bus, which is the bottleneck.

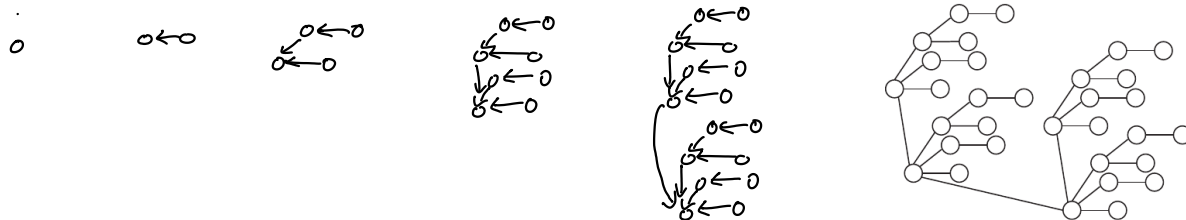
Chapter 3 - Parallel Algorithm Design

Task/Channel Model: Tasks (Program, local memory, IO ports) communicate with channels

Fosters Design Methodology: Partitioning, Communication, Agglomeration, Mapping

3.3. Draw binomial tree with 16 and 32 nodes

-> always duplicate smaller version and connect with one line



-> subgraph of hypercube, used for reduction etc

3.8. Prove that n-element reduction has time complexity $\Omega(\log n)$

TODO look at homework and solution (different)

3.9a Efficient parallel algorithm implementing broadcast

-> Use binomial tree, i.e. do inverse of reduction, $\lceil \log p \rceil$ communication steps

3.10 Why do all-gather and scatter have the same time complexity despite all-gather sending way more data around?

-> For scatter, most processes are idle (e.g. half of the processes only receive data in the last step and never send something. For all-gather, the communication happens in parallel, so the channels can be utilized better.

3.12-3.18 See solutions

MPI Function List

MPI_Init(&argc, &argv)

MPI_Comm_rank(MPI_COMM_WORLD, &id) - process id

MPI_Comm_size(MPI_COMM_WORLD, &p) - number of processes

MPI_Finalize()

MPI_Reduce(...)

MPI_Barrier(MPI_COMM_WORLD) - wait until all processes reach barrier

MPI_Wtime() - returns time

MPI_Bcast(...) - broadcast to all processes

MPI_Send(...) - send message to other process

MPI_Recv(...) - receive message from other process

4 Message-Passing Programming

4.1 Cyclic allocation of n pieces of work ($0 \dots n-1$) to p processes ($0 \dots p-1$).

a) Which pieces are assigned to process k ?

-> $k, k+p, k+2p, \dots$

b) Which process is responsible for work j ?

-> $j \bmod p$

c) Most pieces of work assigned to any process?

$$\lceil \frac{n}{p} \rceil$$

d) Which processes have the most pieces of work?

-> $0, 1, \dots, (n \bmod p) - 1$

e) Fewest pieces of work assigned to any process?

$$\lfloor \frac{n}{p} \rfloor$$

f) Which processes have the fewest pieces of work?

-> $n \bmod p, \dots, p-1$

5 The Sieve of Eratosthenes

Block decomposition: Split data into chunks for each process

the first index for process i is $\lfloor i \frac{n}{p} \rfloor$

the last index for process i is $\lfloor (i+1) \frac{n}{p} \rfloor$

the responsible process for index j is $\lfloor (p(j+1)-1)/n \rfloor$

5.1 Block allocation scheme where the first $p-1$ processes get $\text{ceil}(n/p)$ tasks and the last process get what's left over.

a) Find values for n, p where the last process does not get any elements

TODO

b) Find values for n, p where $\text{floor}(p/2)$ do not get any values ($p > 1$)

TODO

6 Floyd's Algorithm

Calculate execution time:

$$\text{iterations} \cdot \chi + \text{messages} \cdot \left(\lambda + \frac{\text{data}}{\beta} \right)$$

χ cell update time
 λ latency
 β bandwidth

Example for Floyd's Algorithm (counting full communication time):

$$n \lceil \frac{n}{p} \rceil \chi + n \lceil \log p \rceil \left(\lambda + \frac{4n}{\beta} \right) \leftarrow 4 \text{ byte}$$

6.1 TODO

6.2 Why should process 3 and not process 0 read and send the file?

-> Process 3 has $\text{ceil}(n/p)$ rows and process 0 $\text{floor}(n/p)$, so process 3 can use the normal buffer to store the data for all other process while process 0 can't. (Also, like this the file can be read sequentially and process 3 has the correct data in the buffer at the end.)

7 Performance Analysis

$\Psi(n, p)$ - speedup with size n on p processors

$\sigma(n)$ - inherently sequential computations

$\varphi(n)$ - potentially parallel computations

$k(n, p)$ - parallel overhead (communication, redundant computations)

Speedup

$$\Psi(n, p) \leq \frac{\sigma(n) + \varphi(n)}{\sigma(n) + \frac{\varphi(n)}{p} + k(n, p)} \quad (\leq \text{since work might not be divided evenly})$$

Efficiency

$$\varepsilon(n, p) = \frac{\text{sequential execution time}}{\text{processors used} \times \text{parallel execution time}} = \frac{\Psi}{n} \leq \frac{\sigma(n) + \varphi(n)}{p\sigma(n) + \varphi(n) + pk(n)}$$

Amdahl's law

- constant problem size, increase p to reduce time
- ignore overhead \rightarrow tends to overestimate the speedup

$$f = \frac{\sigma(n)}{\sigma(n) + \varphi(n)} \quad (\text{share of sequential computation})$$

$$\Psi \leq \frac{1}{f + \frac{1-f}{p}}$$

e.g. 8 processors, 90 % can be done in parallel:

$$\Psi \leq \frac{1}{0.1 + \frac{0.9}{8}} \approx 4.7$$

Gustafson-Barsis's law

- time is constant, increase p to increase problem size n

$$s = \frac{\sigma(n)}{\sigma(n) + \frac{\varphi(n)}{p}} \quad \Psi \leq p + (1-p)s$$

Karp-Flatt Metric

- includes communication overhead

experimentally determined serial fraction $e = \frac{\frac{1}{\Psi} - \frac{1}{p}}{1 - \frac{1}{p}}$

Isoefficiency

scalability: ability to increase performance as number of processors increases
(by increasing problem size)

total overhead: total amount of time spent by processes doing work not done by sequential algorithm

$$T_o(n, p) = (p-1)\sigma(n) + p k(n, p)$$

sequential execution time

$$T(n, 1) = \sigma(n) + \varphi(n)$$

Isoefficiency relation (maintain efficiency if p increases):

$$T(n, 1) \geq C T_o(n, p)$$

$M(n)$ - memory needed to store problem of size n

- solve isoefficiency for n : $n \geq f(p)$

- $M(f(p))/n$ - scalability function: memory needed per processor

Chapter 11 - Matrix Multiplication

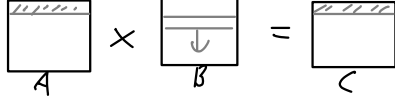
recursive, block oriented algorithm (not parallel)

This is used as the core for the parallel algorithms to do the actual multiplication of parts.

$$A \times B = C \quad A = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} \quad B = \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix} \quad C = \begin{pmatrix} A_{00} B_{00} + A_{01} B_{10} & A_{00} B_{01} + A_{01} B_{11} \\ A_{10} B_{00} + A_{11} B_{10} & A_{10} B_{01} + A_{11} B_{11} \end{pmatrix}$$

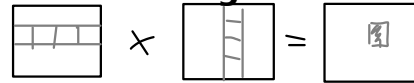
recursively divide matrix into smaller submatrices until they fit in the cache

row-wise block-striped matrix decomposition



row-shaped blocks of B are passed in a circular way between processes, every process has row-shaped part of A and calculates row-shaped part of C

Cannon's algorithm



Every process calculates block of C
Blocks of A and B are passed around




11.2: (row-wise) why not replicate B across all processes?

B would have to fit in primary memory of each processor, limits max. problem size

11.3: a) why is Cannon better for the block oriented algorithm?

-> submatrices are more or less square

b) How to modify block oriented algorithm for block striped algorithm?

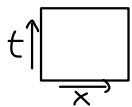
-> don't divide like  , but  or  depending on largest dimension

Chapter 12 - Solving Linear Systems

-> stupid

Chapter 13 - Finite Difference Methods

discretize PDE -> matrix, e.g.:



calculation step e.g.:



parallelize: neighboring elements are needed for calculation step

-> use ghost points that store redundant copies of data



Chapter 14 - Sorting

Quicksort

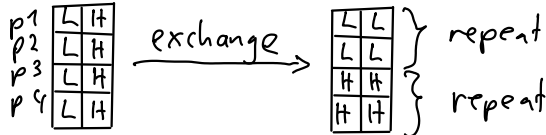
- choose pivot value
- partition the list in low and high sublists
- recurse on low and high sublist
- return [low, pivot, high]

Parallel sorting

- input is evenly distributed along processes
- output is distributed, but not necessarily evenly

First algorithm

- one process globally broadcasts pivot, each process partitions and the lists are exchanged



- afterwards, each process does sequential quicksort of its list
- problem: list sizes are not balanced, since pivot is usually not the median

Hyperquicksort

- first, each process sorts its data using quicksort
- one process chooses median as pivot, all processes split and exchange lists
- afterwards, the two sorted sublists are merged
- communication pattern follows hypercube structure

Parallel Sorting by Regular Sampling (PSRS)

- > more balanced, avoid repeated communication of keys, does not require $p=2,4,8,\dots$
- first: quicksort
- each process samples at $0, \frac{n}{p^2}, 2 \frac{n}{p^2}, \dots, (p-1) \frac{n}{p^2}$
- one process gathers and sorts samples, selects p pivots
- each process uses pivots to split the lists and sends sublists to corresponding process
- each process merges the p sublists

14.1: quicksort is not stable

14.5: Is hyperquicksort or PSRS less disrupted if list is already sorted?

PSRS performs best, since the pivots are globally selected and almost no data needs to be transferred