University of Eastern Finland

Computer Science

Parallel Computing

5 cr, 3621528

Spring 2020

Simo.Juvaste@uef.fi

2.3.2020 12:07

http://https://moodle.uef.fi/course/view.php?id=19368
Course key **parallel**

Sitting placements at the first lectures:

1) Sit within reach of someone (several) else.

2) The whole class must be connected.

Contents

Course contents (preliminary)

Ch 1: An Introduction to Parallel Computing (p. 2)
- What?, Why?, How?

Ch 2: PRAM (p. 22)
- A simple model of parallelism

Ch 3: Parallel algorithms (in PRAM-notation) (p. 30)
- Basic algorithms, e.g., counting, prefix, sorting, etc.

Ch 4: Practical shared memory (OpenMP) programming (p. 55)

Ch 5: Taking the real world into account (p. 56)
- Network delay models, memory access models

Ch 6: Message passing models (p. 73)
- Practical message passing parallel programming (MPI).

Ch 7: Other stuff (p. 75)
- Fortran 90, HPF
- GPU programming, CUDA/OpenCL/OpenACC (p. 82)
- Everyday parallel (and concurrent) programming. Processes, IPC, shared memory, phtreads, Java threads.

1 An Introduction to Parallel Computing

What?, Why?, How?

Some key concepts

Pros, Cons

Other similar terms

Examples

An animal experiment

Design issues

1.1 What is Parallel Computing?

⇒ Use more resources, i.e., several computers, to solve a single computational task in parallel!

- Two is better than one!
- One thousand is better than two...
  - Think human (manual) work.

⇒ The single task has to be divided in several parts.

- Some tasks are easy to divide, some are not.

⇒ The cooperating computers have to be able to communicate.

- One task, one solution.
- There are many ways to communicate.

⇒ The participating "computers" do not need to be complete!

- Processor, memory, communication medium (processing unit).
- Monitors do not process.
- The whole parallel computer still needs to have some I/O, etc.

Example: A human example: manual sorting of papers:

- Input: a bunch of A4 papers, each having a name.
  - Input size: 10, 100, 1 000, or 10 000 papers (1 mm, 1 cm, 10 cm, 1 m).
- Task: sort the bunch (alphabetically).

One (dexterous) person alone: [1st exercise in Data Structures and Algorithms]

- 10 papers: 30 s [3 s/paper]
  - method almost insignificant, insertion/selection sort often favoured
- 100 papers: 8 min [5 s/paper]
  - divide in 10 (or 5-27) sub-stacks according to the first letter(s), sort sub-stacks (insertion sort), combine
- 1 000 papers: 2 h [7 s/paper]
  - divide in 10 sub-stacks according to the first letter(s), apply recursively previous 100-sort.
- 10 000 papers: 25 h [9 s/paper]
  - divide in 10 sub-stacks according to the first letter(s), apply recursively previous 1 000-sort.
  - You might want some help...

Parallel manual paper sorting:

- 10, 100, 1 000, or 10 000 helpers (persons).
- Work organization is (a bit) more difficult than in single person sort.
- Exercises 1-3.

⇒ **The important question**:

- Will 10 helpers speed up the work 10 times?
  - 10 papers task: no (one helper can help a little).
  - 10 000 papers task: yes (at least almost 10 times).
- Will 10 000 helpers speed up the work 10 000 times?
  - 10 papers task: no.
  - 10 000 papers task: no (but we can use more than 10 helpers).
  - 100 000 000 papers task: yes (almost)

⇒ What is the optimal number of helpers for each number of papers?

- What is the goal? What means optimal?
  - Minimal wall clock time?
  - Efficiency (minimal person work hours, i.e., euros)?
  - ???

1.2 Little practice

Rules

- Physical messages, writing on a piece of paper.
  - Written message may include instructions, addresses, data.
- Connections to neighbours without standing up.
- Sending a message (synchronous communication):
  - Ask the neighbour to receive, wait until he/she is ready.
  - Hand out the message, say "here are you".
- Receiving a message:
  - Agree to receive
  - Receive, say "thank you"
- You can see and communicate only with your neighbours.
- Local operations are unlimited.

Tasks

- Max, count, search (single value, pattern), sum, sort, ...

Algorithm?

- For above rules?
- For different rules?
- Without rules (but no magic)?

Physical conditions/restrictions (i.e., challenges/possibilities):

- Open hall, no restrictions

  - Coordination: loudspeakers (for leaders), person-to-person communication, guidance painted on floor, rehearsal, etc.
- Testing, timing algorithm options, optimizing by measurements.
- "Cluster" of two/more door/video -connected halls?
- Sitting here, no person movement allowed.
  - Paper delivery only for neighbours vs anyone?
  - Only one paper at time vs. a bunch at a time.
- How to benefit use of a blackboard or an electronic message board?
- How to benefit from shouting?
- Without sight contact to neighbours.
- Load balancing (fast and slow workers).
- Fault tolerance (temporary, permanent faults?).

1.3 Why parallel computing is needed?

⇒ Why computers are needed?

- Because computers can compute (calculate) fast and they can have huge memory.

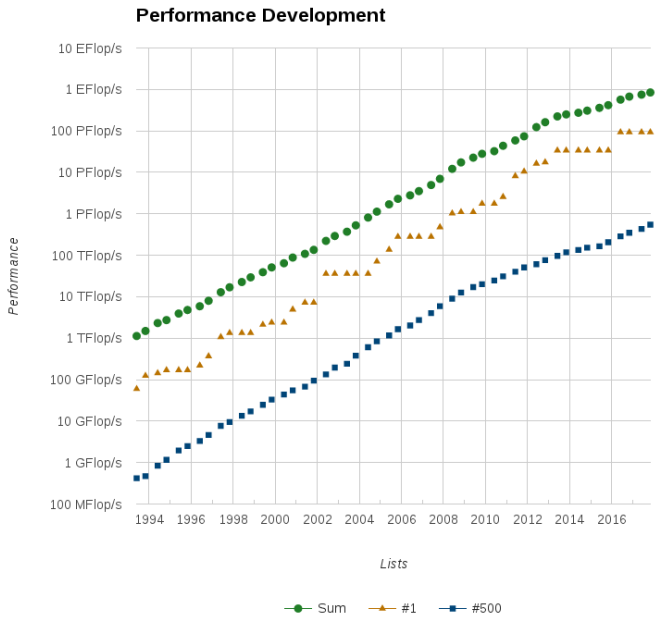
Why *i*9 at 4.3 GHz (2003 slides: 3 GHz) is not enough???

- Computing power will (used to?) about double every 18(-24) months. ["Moore"]
- Intel/AMD 4/6/8-core processors at 2-4 GHz are very cheap (from 100€)!
- 20 years ago governments would have paid millions for a 2020 PC.

What else we need?

- Humans are greedy and impatient...
- Some tasks are too demanding and urgent to be computed by one processor only.

we can use on them.



[top500.org]

What is so demanding and urgent?

- Word processing?
- WWW-surfing?
- Operating system update?
- Malware protection?
- Bank / stock exchange?
- eCommerce?
- Gaming?
- Real world simulation!
  - Matter consists of very tiny particles!
  - Every visible piece consists of very many particles.
  - We cannot simulate every (sub)atomic particle for a large (visible) object!

⇒ But: the smaller particles we can simulate, the more accurate simulation we have!

- Smaller particles ⇒ more particles ⇒ more calculations to do!

⇒ Unbounded amount of calculations!

Why we want to simulate real world?

- "Test" a piece of equipment without building it.
- Prediction of natural phenomena.
- Prediction of consequences of changes.
- "See" artificial things.
- Optimizing structures or models.

Example: weather forecasts

- History data, constants measurements, physical, geographical, and statistical models.
- Simulation of the future movement of air particles.
- Simulation of physical changes (temperature, pressure, humidity, velocity, etc.) of air in the atmosphere.
- Huge amounts of molecules move and interact quickly for several days.
- Incomprehensible amount of calculations.
  - No computer will ever do it in atomic (or subatomic) level.
  - Current computers can do (milli)litres in real time ...

- Solution: resolution reduction – much much less data to simulate.
  - E.g., 10×10×0.5km(×5min) block of air as 1 entity.

Block size (km), height 0.5 km	Gflop/s needed for "real" time simulation (2 minute steps)	5 days in 2 hours Gflop/s needed
1	1 804 492	108 269 544
32	1 762	105 732
1024	1.7	103

- Penalty: accuracy and reliability are reduced.

- Forecast as far to future as possible.
  - Unfortunately: inaccuracies multiply over time of simulation.

⇒ More powerful computer and/or more time gives us more accurate forecasts (and/or longer forecasts).

⇒ (Reliable) weather forecasts are very valuable!

- A late forecast is worthless.
- In real forecasts, the models exploit grid-wide differential equations instead of local simulation...
- Finnish Meteorological Institute: http://en.ilmatieteenlaitos.fi/weather-forecast-models:
  - 7.5 × 7.5 × 0.3 km (× 6 min) Canada .. Ural, 3-10 days (HIRLAM)
  - 2.5 × 2.5 × 0.? km (× ? min) Scandinavia (≈30M 3D blocks) (HARMONIE)
  - (44km->2,5km in 16 years)
  - 2× Cray XC30, (about) 384 × 8-core Xeon, 70 TFLOPS (theoretical)

⇒ Conclusion

- We want as powerful computer as possible!
- We are willing to pay for it.

⇒ Unfortunately

- No IA256 @ 30 GHz ever(?) (until 2040+?)
- Even if we pay all the money in the world.

Thus

⇒ We'll use several processors to achieve more computing power.

- Finnish CSC currently (sisu.csc.fi): Cray XC40
  - 1688 × (2 × 12-core 2.6GHz Xeon E5, 64GB, 130GB/s)
  - Theoretically 1689 TFlop/s, measured 1250 TFlop/s (Linpack)
  - https://research.csc.fi/csc-s-servers
  - see "Current parallel computers (briefly)", page 14.
- LUMI (Large Unified Modern Infrastructure)
  - To be installed 2020-2021 at Kajaani.
  - 200 PFLOP/s
  - Architecture to be announced, heterogenous anyway (CPUs and GPUs)

Other applications for processing power (parallelism)

- Molecular modelling (chemistry)
- Sub-molecular (sub-atomic) modelling (physics)
- Huge databases, urgent queries, data mining
- Digital signal/image/video processing
- Complex user interfaces (virtual reality, games)
- DNA matching
- DNA / other molecular modelling
- Environmental modelling (storms, pollution, earthquakes, sea currents)
- Astronomical modelling
- Optimization (aero/hydrodynamics, etc.)
- Structure strength calculations (car crash simulations, etc.)
- Cryptanalysis
- Pattern matching/recognition, audio/image/video surveillance

- Data mining/indexing/classification
- Artificial intelligence/machine learning
- Measurement data analysis and modelling (sensor values to big picture)
- Autonomous vehicles (processing of sensor data)

1.4 Some key concepts

Example: Building a small house:

- One skilled man can build a house in one year
- Two skilled men can do it in about half a year
- 12 men, one month: requires very careful planning (at least)
- 365 men, one day: probably impossible
- 1 million men, 10 seconds: definitely impossible

⇒ How to coordinate the fast (1-5 day) parallel building of a house?

- Skilled workers
- Perfect plans and very good leaders.
- Synchronization of work
- (Partly) independent components (roof, walls, etc.)
- More than one (levels of) leader(s)
- Good instructions and communication
- Detailed plan available to all (at least many) workers
  - Problem: single plan will be crowded
  - Solution: local partial copies of the plan

⇒ Lessons learned:

- Parallelization possibilities depends on the problem (ditch vs. well)
- Communication and coordination are vital
- Access to a SHARED plan with local copies is a fairly good communication method

⇒ There is a limit on efficient number of workers.

- Key concepts:
  - speedup, extra work, efficiency

Example: which one to choose?

Labour	Calendar time	Speedup	Work	Labour expenses	Efficiency
1 man	1 year	1.00	1.00 my	48,000 €	1.00
2 men	7 months	1.71	1.17 my	56,000 €	0.86
4 men	4.5 months	2.67	1.50 my	72,000 €	0.66
365 men	5 days	73.00	5.00 my	240,000 €	0.20

Think BIG!

- Larger buildings
- Great Wall of China (in a day?)
  - 5 mm / ≈ 300 kg of wall for each Chinese
- Great Pyramid of Giza (in ???)
  - ≈ 60 kg for each Egyptian

Speedup (nopeutus), work (työ), efficiency (tehokkuus, hyötysuhde)

- An **optimal** sequential (uniprocessor) algorithm time = $T_s(N)$.
- Parallel algorithm with P processors, time = $T_p(N,P)$
- Speedup is defined as ratio T_s/T_p
  - Speedup is at most P ($T_s/T_p = O(P)$)
  - I.e., superlinear speedup is not possible, as it would imply a faster sequential algorithm (see Brent's theorem below, p. 10).
- Work (used resources) $W_p = T_p \times P$.
- If $T_p \times P = O(T_s)$, the algorithm is work optimal (työoptimaalinen).
  - $T_p \times P = o(T_s)$ is impossible!
- Efficiency = $T_s/(P \times T_p)$.
  - Inefficiency (overhead) = $P \times T_p/T_s$

Limits of parallelization

- Can we speed up a computation infinitely by adding more and more processors?
  - Not infinitely, most problems have a lower time bound (usually (poly)logarithmic time, with polynomial number of processors).
- In practice, the limit is money.
  - Hard problems are huge (input size (N) is large).
    - Huge problems have a lot of potential parallel computations.
    - E.g., a high-rise building vs. a single-family house.
  - Small problems are fast enough with one processor.
- In *theory*, the limit is 3-dimensional space and speed of light.
  - We can reach only constant number of processors in a constant time.
  - We cannot reach exponential number of processors in polynomial time,
  - nor polynomial number of processors in logarithmic time.
  - $T(N,P) = \Omega(P^{\frac{1}{13}-\epsilon})$

Amdahl's law on serial fractions within parallel programs

- If an algorithm has an (inherently)sequential part that will not be parallelized, it will limit whole parallelization.
  - Or, if we do not bother to parallelize some difficult part.
- Whole algorithm (sequential) time T, containing a sequential fraction α (0..1).

$$T_P = \alpha T + (1-\alpha)\frac{T}{P} \tag{1}$$

$$\text{Speedup} = \frac{T}{\alpha T + (1-\alpha)\frac{T}{P}} = \frac{1}{\alpha + \frac{1-\alpha}{P}} \to \frac{1}{\alpha} \quad (\text{when } P \to \infty) \tag{2}$$

$$\text{Efficiency} = \frac{T}{P(\alpha T + (1-\alpha)\frac{T}{P})} = \frac{1}{(P-1)\alpha + 1} \to \frac{1}{P\alpha + 1} \quad (\text{when } P \to \infty) \tag{3}$$

- I.e., if α is a positive constant, there is a limit on speedup, and thus efficiency drops as we add more processors (efficiency $\to 0$ when $P \to \infty$).

Scalability is possible in spite of Amdahl's law

- Plain Amdahl's law gives a bit too pessimistic view for possibilities of parallelization.
- In practice, as we have more processors, we'll solve larger inputs. Larger inputs usually have relatively lower α.
- We can relate the number of processors to the input size (or vice versa) (Gustafson's law [http://www.johngustafson.net/pubs/pub13/amdahl.htm]).
  - If we assume that sequential section has constant time s and input size increases with the number of processors, the portion of sequential time (α in Amdahl's law) gets smaller and smaller as P increases.
  - If the time complexity class of the sequential section is less that the time complexity class of the parallel section, then increasing the input size always helps.
- Most algorithms have several stages, often with varying level of parallelization possibilities.
  - It may be possible to adjust the number of processors used for each stage.

Possible goals for speedup and/or efficiency

- As fast as possible.
  - No matter how many processors (e.g., $P=N$ or $P=N^2$).
  - For most problems, there exists a (poly)logarithmic-time ($(\log n)^k$) algorithm (very fast!).
- As good efficiency as possible.
  - Unfortunately, the sequential algorithm is always the most efficient.

$\Rightarrow$ As fast as possible while maintaining (asymptotically full, or given) efficiency.

- Something between, or in real life:
  - In a given time by as few (and cheap) processors (and other resources) as possible.
  - By a given number of processors (and other resources) as fast as possible.
- See below efficiency classes (page 31).

Brent's theorem

- If our algorithm works with P processors in time T, we can execute it with $P' < P$ processors in time $T \times \lceil P/P' \rceil$.
  - Each of P' processors does the work of (simulates) $\lceil P/P' \rceil$ "virtual" processors of the original algorithm.

$\Rightarrow$ We can always design algorithms for as many processors as possible/efficient. The algorithm will work nicely with fewer processors.

- Even if we won't have thousands of processors, multithreaded processors work more efficiently with more threads.

$\Rightarrow$ In some cases, thought, an algorithm that is designed for fewer processors may be more efficient.

What is so difficult in parallel programming?

- Sometimes even sequential programming is difficult.
- In parallel programming we have to manage several processors, each of which must work correctly.
- The processors must communicate correctly.
  - Communication requires synchronization.

- Some problems are easy to parallelize, some more difficult or inefficient.

⇒ Parallel programming is difficult.

⇒ We often need more abstraction levels than in sequential programming.

- Concentrate on data and operations on data.

Parallelism is natural!

- In fact, sequential order is (sometimes) artificial.
- A "typical" algorithm segment:

  for all $elem \in A$ **do**
  $elem \leftarrow elem \times 2$
  end for

- A sequential programmer implements:

```
for (i = 0; i < A; i++)
    A[i] = A[i] * 2;
```

- Why to serialize an originally parallel (simultaneous) operation?
- Sometimes serialization might be a source of errors.
- A parallel version can be flexibly implemented with $1..N$ processors.
- Real world is concurrent (and very parallel) anyway.
- Parallelism is (almost) as old as Life.

1.5 Some similar terms (that are sometimes mixed up)

Distributed System (hajautettu järjestelmä)

⇒ A distributed system is a collection of autonomous computers linked by a computer network that appear to the users of the system as single system.

- The machines are autonomous; this means they are computers which, in principle, could work independently;
  - Separate computers work concurrently, without global clock, and may appear, fail and recover independently.
- The user's perception: the distributed system is perceived as a single system providing a service (even though, in reality, we have several computers placed in different locations).

⇒ Each part of the distributed system may be a part of (i.e., participate in) several distributed systems.

- Not part of this course.

Distributed computing (hajautettu laskenta)

- Term often used when several computers (often geographically distributed) are used to compute a single computational problem in parallel.
  - Message passing programming, tolerate long and/or unpredictable delays, low bandwidth
  - E.g., SETI@home, distributed DNA matching, etc.
  - Boundary between parallel and distributed computing depends on the speaker.
  - Sometimes, "distributed computing" is used on "distributed systems".

- ”Grid computing” (ritilälaskenta).
  - New (2010’s) buzzword for distributed computing.
  - Sometimes also used to do load-balancing and (automated) assignment of jobs among computers.
- Part of this course.
- Sometimes ”distributed” or ”grid” computing are used for just remote access of computing power.

(Parallel) cloud computing

- Goal is usually more dynamic usage of hardware resources (on demand).
- Rent/buy processing power on demand from Amazon, Google, Microsoft, etc.
  - Either processor time (1000 virtual processors for an hour), or processing time (1000 processing hours).
- Providers use virtualization and dynamic load balancing techniques to provide the processing power.
- Either
  - Rent a virtual computer of P processors and use a parallel program (e.g., using MPI), or
  - use parallel data processing primitives provided by cloud providers.
    - Platform provides tools needed for scalability: distributed file system, job monitoring, fault tolerance, etc.
    - See Hadoop/MapReduce, p. 85.
- As virtualization masks the true communication architecture, software cannot be optimized.
- Performance penalty (overhead) compared to a native implementation depends on the need of communication within the algorithm.
- Not for absolute highest performance (nor efficiency).
- More expensive compared to real owned machines (in continuous use).
- Dynamic virtual processor creation suits very well for occasional parallel computing needs.

Concurrent system (samanaikainen)

- Things occurring apparently simultaneously.
  - In reality, only one (process, etc.) is executing at a time, and the process is changed frequently enough.
    - E.g., processes in a multitasking OS execute at $\approx$10 ms time slices.
  - Can also occur really simultaneously in multiprocessors systems.
  - Concurrency is defined with respect to a slow observer (human).
- Order of concurrent events is nondeterministic.
- Can be (usually is) implemented using time-sharing (sometimes several processors).
- Tasks are not necessarily (tightly) related.
- Parallel and distributed systems are concurrent by nature.
  - Processes in different computers execute simultaneously
- The communication in asynchronous distributed systems is concurrent.
  - To achieve most flexibility and performance, the processes (computers, software) that participate in a DS are usually concurrent (multithreaded).
- Concurrency theory (or practical handling) is not part of this course.

Multithreading (säikeistys)

- The standard mechanism to implement a concurrent process (one process multiplexing several points of execution)
- As opposed to distinct processes, the threads of a single process share the same data.
- Not part of this course.

Multithreading according to processor manufactures

- Processor includes special circuits to execute several processes simultaneously.
- Depending on the implementation, the processes may execute at full speed, or at slightly lower speed.
- Benefit: more efficient utilization of functional units.
- OS (and processes) "see" several processors.
- E.g., Intel HyperThreading(tm), SUN CMT.
- Relates to this course.
- See "Processor multithreading" (page 17).

Distributed operating system

- Single system image (user point of view) for several computers.
  - User will not know in which physical computer their processes run.
  - (Semi)automatic job/process distribution, balancing, migration.
- "Grid computing"
- E.g., Mosix

Parallel computation/computer (rinnakkaislaskenta, -tietokone)

- Use several processors/computers to solve a single computation in parallel
  - The only goal is to make hard computing faster.
    - Up to P times faster using P processors.
  - Useful (only) if we are in a hurry (simulation/forecast, real-time applications)
- A parallel computer often has dozens .. thousands of similar processors with a tight interconnection and often a (virtual) shared memory.

Parallel, distributed, and concurrent systems and programming have a lot in common.

- Task division.
- Interprocess communication, dividing data.
- Nondeterminism.
- Synchronization challenges.
- Deadlock possibility.
- Loadbalancing.
- Error possibilities, fault-tolerance techniques.

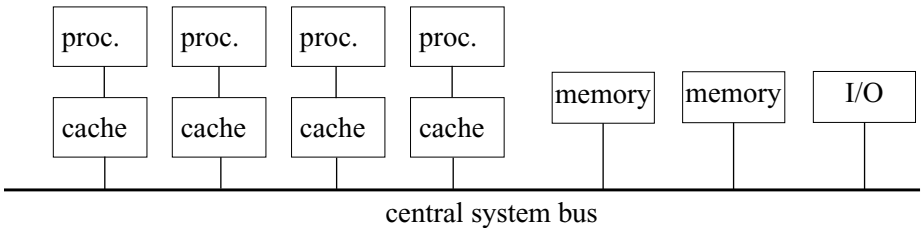
⇒ Hardware, tools, and goals differ.

- In this course, we concentrate on parallelism, but we'll might have something (threads, processes) on concurrency.

1.6 Current parallel computers (briefly)

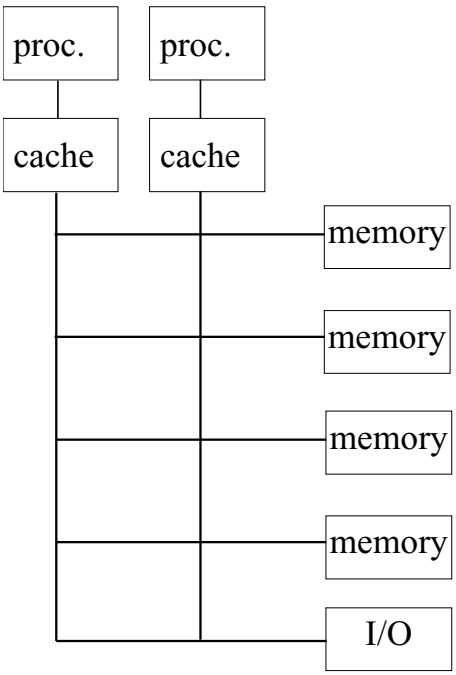
SMP (Symmetric MultiProcessor)

- 2-16 (-64) processors on the same memory bus (or switch).

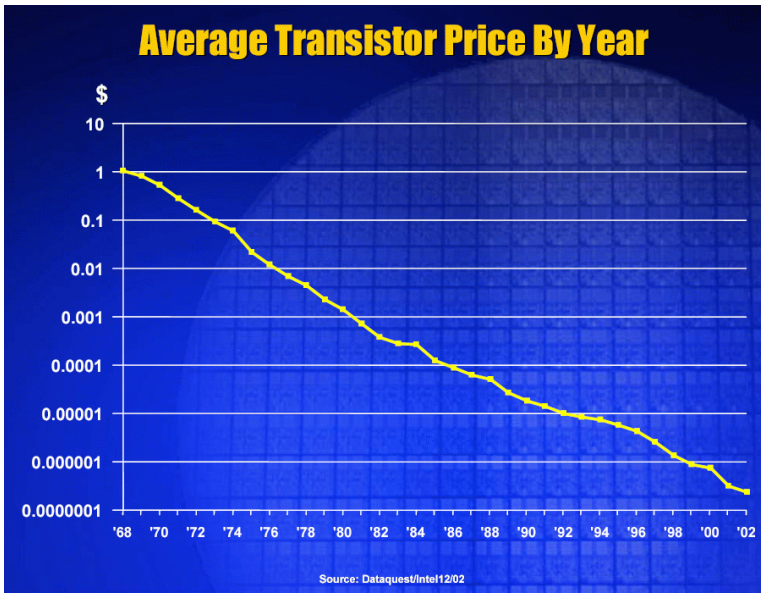


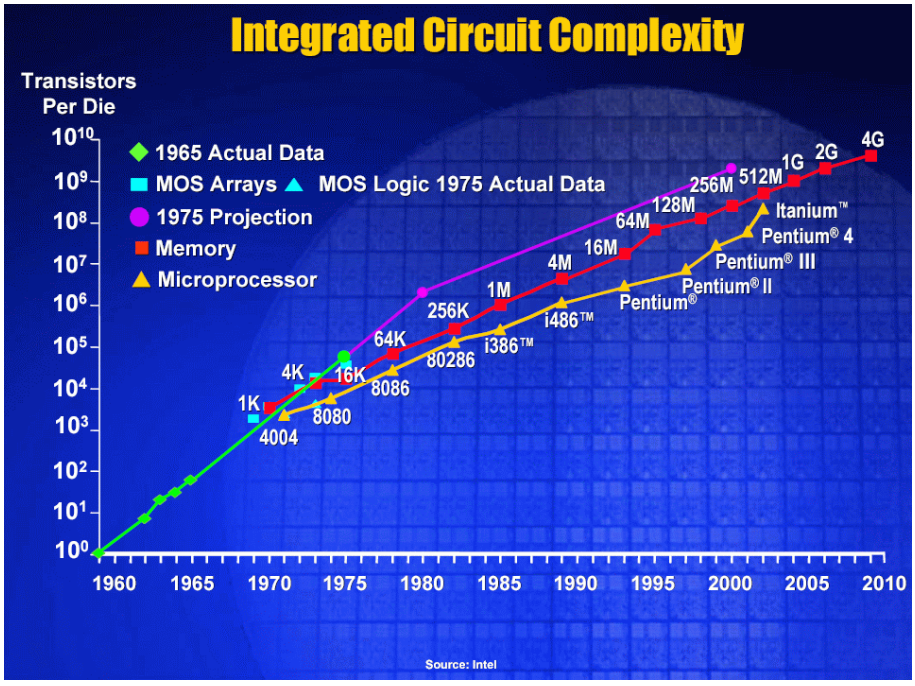
- Several banks of memory.
- Each processor has its own cache (to reduce bus traffic).
- Not very scalable approach (as bus, a bit more with a switch, see below).

- cs.uef.fi (2015) 4 × Intel Xeon E7-4860 v2 @ 2.60GHz (12 cores each, 2 threads each).
- In larger units (P ≥ 8-16), processors are usually clustered.
- Processes do not communicate directly, memory is used for communication.
  - Processors may use a direct links (e.g., Intel QPI) for cache coherency or/and memory access (NUMA).
  - If (as) processors have integrated memory controllers, the memory is not actually physically shared. Instead, there will be a lot of memory access traffic between processors (see p. 16).
- Usually used to improve throughput in a concurrent system, can be used for small scale parallel computation as well.



Why parallel (once again) [Gordon Moore, ISSCC2003, www.intel.com]





Multicore SMP, SMT, CMT

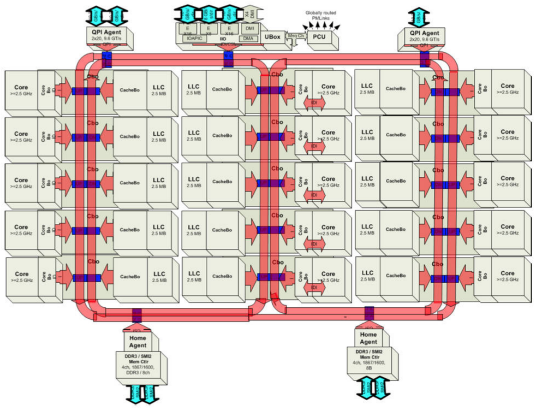
- As the silicon manufacturing process improves, more and more transistors can be fitted in a chip (mainframe/supercomputer: in a board).
- How to use the exponentially growing transistor count efficiently?
  - 1940's to 70's: more and more bit-parallelism and more complex instructions.
    - Eventually diminishing returns.
  - (70's), 80's, 90's: deeper pipelining (more MHz), wider superscalar, VLIW.
    - Usefulness of deeper pipelines and wider superscalar is limited by code/compilers, eventually diminishing returns.
  - Since late 80's: more and more cache to balance slow memory.
    - Difference of 2MB and 4MB L2 caches is small in speed, but has more transistors

than an ALU, eventually diminishing returns.
  - Since mid 2000's: more cores (and more cache).
    * More memory bandwidth.
    * (And more integration (MMU, IO, GPU, etc) for cheap PCs)
  - Same transistor count: 12 000 × i386 and single 8-core i9-9900K!

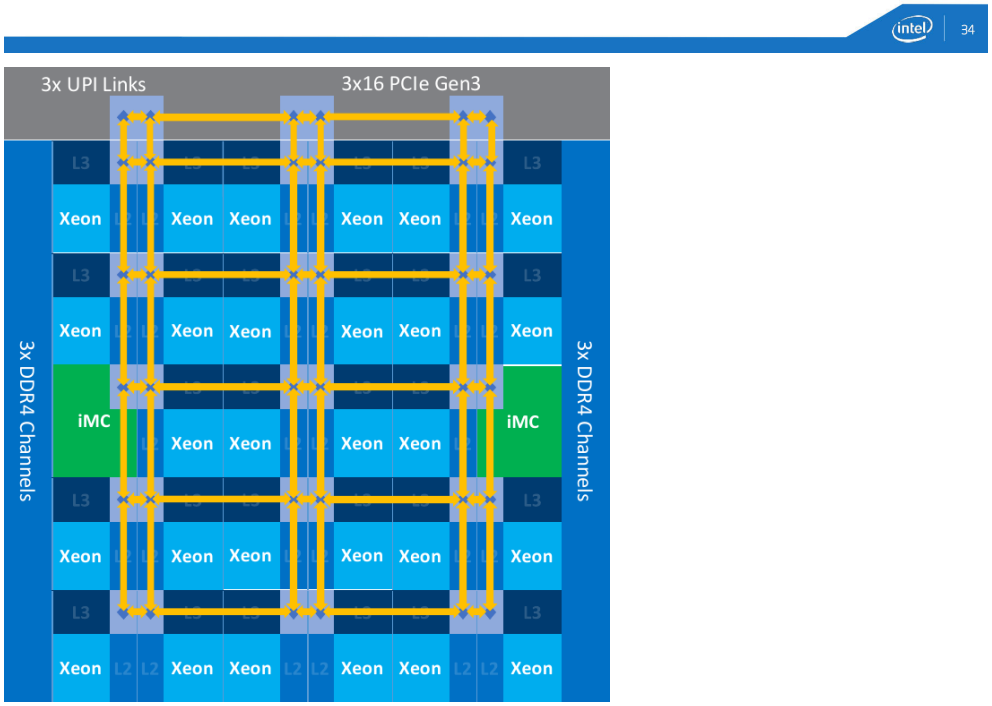
- Multicore SMPs have several CPUs within the single silicon chip / package
  - Each CPU has its own ALU(s), L1 (& L2) cache, usually also FPU.
  - CPUs share L3 (&L2) cache, MMU, and external connections
- Multicore benefit
  - P times processing potential for approx. the same price
- Drawback
  - Memory and I/O bandwidth do not increase accordingly, eventually diminishing returns.

Closer look at IVB EX

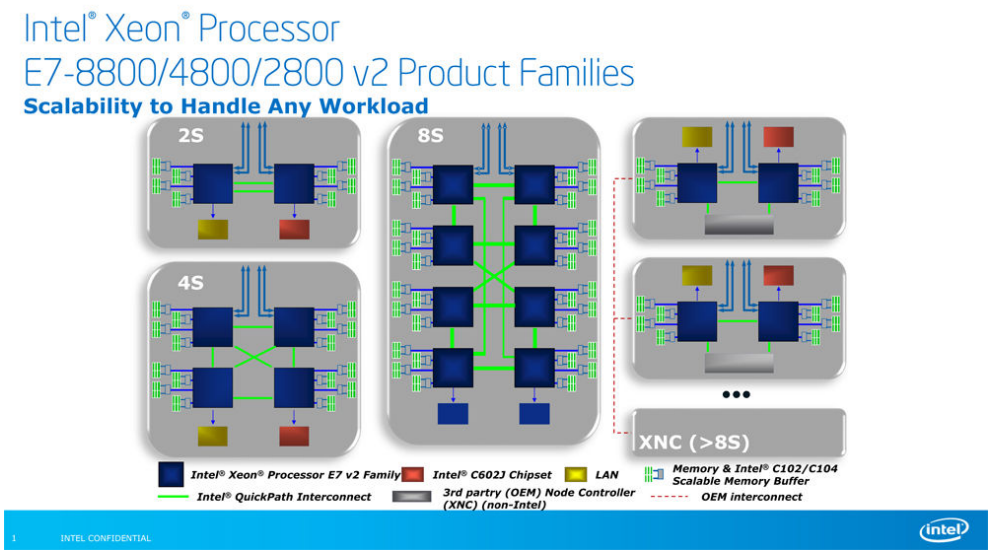
E7-4890 v2 block diagram



- 15 IVB Cores
- Large shared Last Level Cache (L3)
- 2 Home Agents/Memory Controllers
- 3 Intel® QPI links
- Integrated PCIe* Gen3
- Scalable on die interconnect



Mesh connection of Intel Xeon Skylake-SP.



Processor multithreading

- Better utilization of the several functional units of a processor.
- Instructions of separate threads/processes are guaranteed to be independent.
- Each core executes several processes (threads).
  - Reduces the impact of memory latency by making each virtual processor slower.
  - Sun UltraSPARC T3
    - 16 cores, 8 threads each → OS sees 128 threads ("processors")
  - Cray XMT (originally Tera MTA)
    - 128 threads per processor.

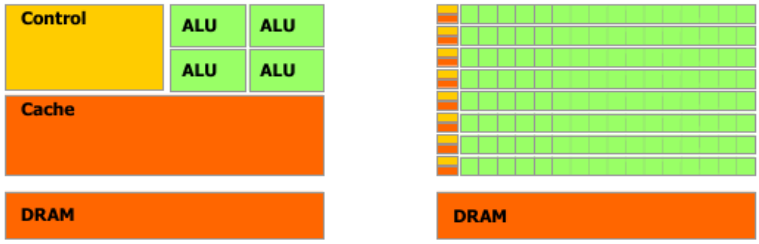
⇒ Multicore is mainstream now (2006 slides: "soon").

- Intel
  - 4-core Celeron, 8-core i7, 18-core i9, 56-core Xeon (Hyperthreading: 2 threads/core)
  - Dual core P4 at 2005, quad core at 2007.
- AMD: 64-core EPYC/Threadripper, 16-Core Ryzen (×2 threads/core).
- PlayStation4 Pro
  - 8 CPU cores 2300 GPU cores.
- SUN/ORACLE 12-core SPARC M6, 16-core T3/5
  - SUN dual core UltraSPARC IV at 2004, 8-core T1 at 2006.
- IBM 12-core POWER8, dual core PPC970 at 2004.
- Nvidia Pascal 2016: up to 3840 cores, 96 threads/core, @700€, or 768 cores @150€ (0.20€/core).
- All new smartphones are dual/quad core, or more.
- No single core servers or PCs available, no/few new single core smartphones.

⇒ Nowdays, we can assume that our software is run mostly on parallel machines!

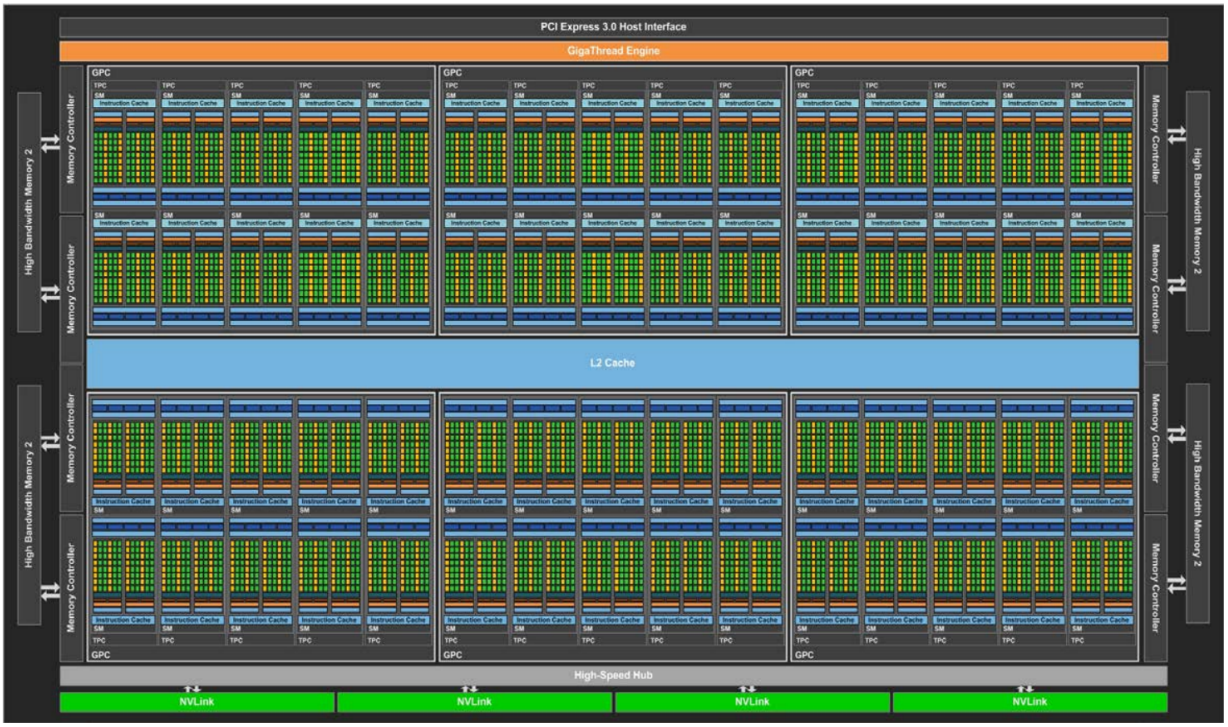
GPGPU

- General Purpose Graphical co-Processor Unit computing.
- Graphical co-Processors have up to thousands of "processors" to accelerate 2/3D image generations. Also, good/excellent memory bandwidth.
- What to do with 18 billion transistors in a chip? https://www.nvidia.com/fi-fi/titan/titan-rtx/ vs. https://www.intel.com/content/www/us/en/products/processors/core/x-series/i9-10980xe.html



[NVIDIA]

  - Notice: Control and Caches are actually mixed with ALUs.
- MIMD vs. SIMD/SIMT (see p. 20).
- Hierarchical (embedded) memory.
- Highly multithreaded execution.



Vector (super)computers

- Classical supercomputers since Cray 1 at 1977.
- 1-32 (even more in clusters) extremely powerful processors.
- Each up to 100 GFLOPS (2008).
- ≈8 MUL-and-ADD floating point operations / clock cycle / processor
  - E.g., dot product
  - Requires several long (1000+ element) arrays (vectors) for peak performance.
- On each clock cycle, up to 16 words (á 64B) from/to memory.
  - Average PC: 0.1 .. 1 B/cc
- No caches, but hardware prefetch (very deep pipeline) and very wide memory channels (and SRAM memory).
- Cray, Hitachi, Fujitsu, NEC.
- Very expensive, even per FLOPS.
- More or less extinct in original form, current implementations approach MPPs, see below.
- NEC SX-9 (2012): 100 GFLOPS/proc, 256 GB/s memory bandwidth/proc

SX-9 Vector Supercomputer Specifications

	Multi-node	Single-node	
	2 – 512 nodes	1 node	
	SX-9	SX-9/A	SX-9/B
Central Processing Unit (CPU)			
Number of CPUs	32 – 8,192	8-16	4-8
Logical Peak Performance	3.8T – 969.9TFLOPS	947.2G – 1,894.4GFLOPS	473.6G – 947.2GFLOPS
Peak Vector Performance	3.3T – 838.9TFLOPS	819.2G – 1,638.4GFLOPS	409.6G – 819.2GFLOPS
Main Memory Unit (MMU)			
Memory Architecture	Shared and distributed memory	Shared memory	
Capacity	1T – 512TB	512GB, 1TB	256GB, 512GB
Peak Data Transfer Rate	2,048TB/s	4TB/s	2TB/s
Internode Crossbar Switch (IXS)			
Peak Data Transfer Rate	128GB/s bidirectional (per node)	-	

MPP (Massively Parallel Processing)

- Tens .. thousands of processors (or more).
- Each processing node is a 1-4 processor SMP (× 8-64 cores/proc) and memory.
- Separate I/O nodes.
- Processing nodes connected by an interconnection network, topologies vary.

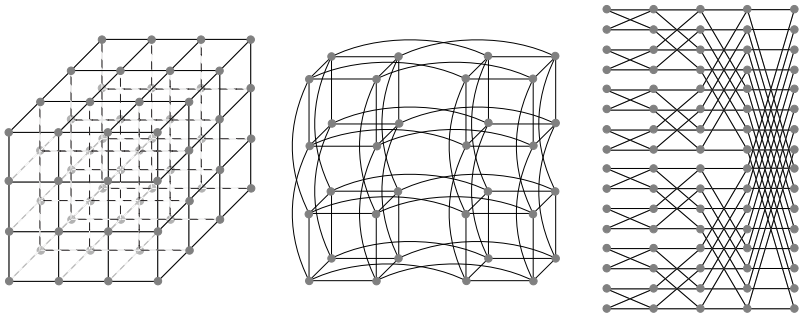


Figure 1: A 64-node 3D mesh, a 32-node binary hypercube, and an 80-node butterfly (with 16 input/output nodes)

- Usually hardware supports virtual shared memory.
- Scales large enough (can be built to consume any budget).
- Communication network is expensive (up to half of the machine cost).
  - Full bandwidth network (having constant amount of machine-wide bandwidth per processor, hypercube, sparse mesh, butterfly) is very expensive.

  - Most mainstream machines have lesser networks (mesh, fat tree), but the bandwidth does not scale with increasing number of processors.
- Communication is usually hybrid and hierarchical (up to 5-6 levels, see below p. 21).
  - Multicore – SMP – crossbar – torus
- Special purpose machines can be tailor-designed to balance the costs of subsystems (processors, memory, bandwidth, I/O) with the given task.
- Processing nodes might have also vector, GPGPU, or other extra processing hardware in addition to the main CPU.
- General purpose computers provide compromises between price and interconnection and memory performance.
- Originally SIMD was used to save silicon area, (single control unit, multiple ALU/FP units), later mostly MIMD or hybrid.
- E.g., (ILLIAC IV), Thinking Machines CM-1, -2, -5, Cray T3E, XT4/5, XE6, Digital (HP) Alphaserver SC, IBM eServer, Intel ASCI Red, SGI, etc.

NOW (Network of Workstations)

⇒ Personal workstations are 99% idle (nights, editor usage).

- Free cycles can be used by: `nice compute`
- "Free" (unused) computing power:
  - School of Computing: 300 PCs × 5 GFLOPS = 1.5 TFLOPS.
  - UEF: 5000 PCs × 5 GFLOPS = 25 TFLOPS.
  - Finland: 1.5 M PCs × 3GFLOPS = 5 PFLOPS > Blue Gene.
- Ordinary Unix (WinNT) workstations, TCP/IP connection.
- A switch . . . LAN . . . WAN . . . Internet.
- Sometimes (nowadays) also a dedicated cluster (ryväs).
  - 1(0) Gb Ethernet, PCIe, Infiniband, ATM, FC, or Myrinet; no displays, etc.
  - Blade racks to save space, reduce loose wires.

⇒ Slow(ish) communication restricts algorithm choice.

⇒ Cheapest FLOPS because of mass production!

Parallel architectures seem to converge

- In SMP-computers the buses are replaced by (clustered)networks.
- Vector supercomputers are (were) implemented in CMOS, use caches and DRAM, P increases, nodes are clustered (memory performance degrades or no uniform shared memory).
- Vector techniques and virtual shared memory are used in MPP computers.
- Multithreading and multicore are used in CPUs and GPUs.
- Workstation (or server computing nodes) have parallel vector units.
- MPP computers are build from commodity parts like NOWs.
- Workstation processors have vector (SIMD) instructions (e.g., Intel SSE)
- Dedicated "NOWs" are used for parallel computation.
- Several (even heterogeneous) computers are connected for joint work (grid computing).
- Blade server racks look like a mainframe. . .
- Hybrid models (large DMM of SMP/GPU nodes), also in programming.

Flynn's taxonomy of parallel computer architectures

- General, coarse classification for parallel machines, especially for processor capabilities.

	Single **I**nstruction stream	**M**ultiple **I**nstruction stream
Single **D**ata stream	SISD	MISD
Multiple **D**ata stream	SIMD	MIMD

- SISD: sequential computer
- SIMD: single control unit, all ALUs execute always the same instruction (or idle).
  - saves transistors for actual computation
  - more complex programs execute inefficiently
  - used new especially in GPUs.
- MIMD: processors work independently
  - now more common for general purpose computation.
- Sometimes also term MPMD (**M**ultiple **P**rogram) is used to emphasize role of a separate program in an auxiliary processor (vs. SPMD).

Current top computers: http://www.top500.org/

IBM Summit

- 4608 × (2 × 22-core POWER9 + 512 GB+ 6 × 5120-core TESLA V100).
- 148 PFLOPS, 13 MW
- https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/

Sunway TaihuLight

- 40 960 × (4 × (64-core vector processor) ASIC) – 93 (125) Pflop/s, $260M.
- http://www.netlib.org/utk/people/JackDongarra/PAPERS/sunway-report-2016.pdf

Tianhe-2

- 16 000 × (2 × Intel Xeon E5-2692v2 12-core 2.2 GHz + 3 × Xeon Phi (64-core SIMD)). 1.3 PB memory (mostly for E5s))
- 34 PFLOPS, 18MW.
- Fat tree interconnect [Dongarra 2013].

IBM Sequoia - BlueGene/Q

- 98 304 × 16-core PowerPC
- 16 PFLOPS, 1.5 PB memory, 7900 kW

Additional bonus on parallel computers

- As we can have unlimited performance via parallelization, we do not need the fastest processor. Instead, we'll select the best by performance/price. (www.verkkokauppa.com 8.1.2018)

i3-8100 4× 3.6 GHz	124 €
i3-8350K 4× 4.0 GHz	191 €
i5-8600K 6× 3.6 GHz	277 €
i7-8700K 6× 3.7 GHz	401 €
i5-7640X 4× 4.0 GHz	261 €
i7-7800X 6× 3,5 GHz	394 €
i7-7820X 8× 3,6 GHz	622 €
i9-7900X 10× 3.3 GHz	1 027 €
i9-7940X 14× 3.1 GHz	1 415 €
i9-7980XE 18× 2.6 GHz	2 032 €
Xeon Platinum 8176 28 × 2.8 GHz	≈9 000 €
Xeon Platinum 8180 28 × 3.2 GHz	≈11 000 €

- Not quite as simple as GHz/€ or even GFLOPS/€.
  - We need more than processors (motherboards, network cards, switches).
  - Algorithm may be slightly less efficient with more processing nodes.
  - See exercises.

2 PRAM

A simple model of parallelism

PRAM programming

PRAM physical implementation possibilities

⇒ PRAM is used to avoid dirty details!

2.1 PRAM shortly

How PRAM was born?

- RAM (Random Access Machine)
  - A processor
  - A memory (RAM, Random Access Memory)

⇒ A familiar computer abstraction (for computer architects, algorithm designers, programmers, etc.):

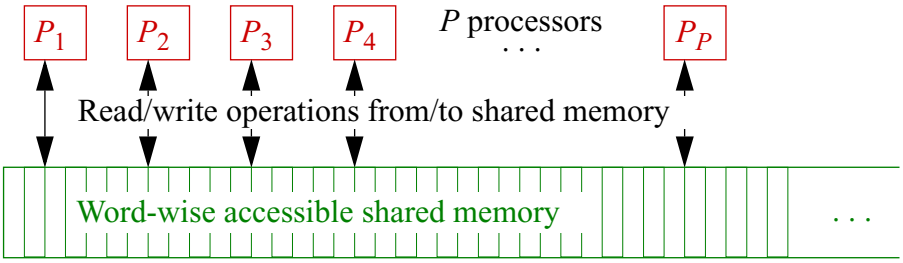
- Procedural (or OO) programming, especially variables.
- Not quite accurate any more, but good enough.

A natural extension:

- PRAM (Parallel Random Access Machine)
  - Fortune and Wyllie 1978, many others

⇒ Increase the number of processors.

- All processors can equally access the shared memory.
  - Shared memory is (also) the communication medium of the processors!



⇒ Programming like RAM, except memory (variables) is shared.

- All processors have to be programmed.
- Memory access conflicts have to be avoided.

Why PRAM is good:

- Simple and strong model.
  - If a parallel algorithm can be done, it can be done for PRAM.
- Resembles real parallel computers (as RAM resembles sequential computers).
- Flexible: Tens of different variations (good or not?).
- Generally used.
  - Most parallel algorithms are designed for PRAM.
  - Rich existing set of algorithms and other theory.

Why PRAM is bad:

- P-port shared memory cannot be built (easily).
- Real world delays are ignored.
- Does not account for building costs.
  - Does not guide for saving resources.

Still:

- A handy tool (abstraction) for research and teaching.
- PRAM algorithms can be adapted for real computers.

2.2 PRAM (memory) model

Processors are processors, brand does not matter.

- If needed, we can define each processor (processing node) to have local memory and I/O.
- Especially the program can be stored as local copies, but as a plain model, it does not matter.
- Usually we assume the same program but own program counters at every processor (MIMD, multiple instruction stream, multiple data).
  - SIMD (single instruction stream) is an option for cheaper implementation.

The shared memory in PRAM is interesting!

- To operate efficiently, the processors need to be able to use memory.
  - Up to a read/write at every clock cycle by every processor.
- Is it possible/feasible to define/implement a memory that can handle P simultaneous memory accesses every clock cycle?
  - It is easy to define.
  - It is attractive to use.
  - It might be possible to implement (with some tricks).
  - It is not currently feasible to implement, though.
  - For a while we assume that it is possible, and we'll exploit it to achieve easiest possible parallelism.

Processor – memory speed comparison (Random Access Machine):

- 8 bits/DRAM chip, 50 ns random access latency, 3 GHz 64-bit processor:
  - $3 \times 50 \times 64/8 = 1200$ DRAM chips/processor for full random access of one word at every clock cycle!
    (24GB/s, $\approx$32 chips for sequential access!)
- Actually, modern (SD)RAM should not be considered as RAM(emory)...

PRAM memory model

- A single memory, indexed memory locations (e.g., $1..m$).
  - m usually "unlimited" (as in RAM).
- Each memory reference (read/write) is done in unit time ($O(1)$, 1 clock cycle).
- Also, all other machine instructions in 1 clock cycle.

$\Rightarrow$ What about if simultaneous memory references hit the same memory bank or even the same memory location?

- Simultaneous: on the exactly same clock cycle, no timesharing possible within a clock cycle. Also called concurrent in this context.

Same bank, different address:

- For the model, there is no such problem.
- For a real implementation, we need more circuitry and/or tricks (see below) (and delay).

Several simultaneous memory references to the same memory address:

- The references could possibly be combined.
- Write requests: something is written.
- Read requests: the result is copied to all accessing processors.

$\Rightarrow$ In a model, we just define what will happen.

- Several simultaneous reads is a strong operation, but very easy to define.
- Simultaneous read(s) and a write can be defined as, e.g., every write to occur before every read (two stages = $O(1)$).
- Several simultaneous writes is much more difficult to define.
  - Each memory location will always contain only one value.

$\Rightarrow$ In PRAM model, these are considered as model variations.

PRAM variations

- Around the concept of shared memory, several variants have been proposed and used.

  - To make algorithms easier or more efficient.
  - To follow some implementation ideas.
- The memory models differ on restrictions/results on what can happen at single memory location at a single clock cycle.
- If the restrictions are violated, the whole machine halts immediately (in a model), or results are unknown (in real life).

E/C/O × R/W

- EREW (Exclusive Read, Exclusive Write)
  - Both several simultaneous reads and writes are forbidden.
- CREW (Concurrent Read, Exclusive Write)
  - Several processors may read simultaneously, but writing is allowed to one processor at a time.
- CRCW (Concurrent Read, Concurrent Write)
  - Unlimited number of reads and writes are permitted simultaneously.
  - The result of simultaneous writes has to be solved somehow, see below.
- CROW (Concurrent Read, Owner Write)
  - Each memory location is owned by a processor, others may only read it.
- ERCW (Exclusive Read, Concurrent Write)

CW variation examples

- On concurrent access to a single memory location.
  - In ascending (partial) order of strength.

- WEAK
  - Only simultaneous writing of zeroes is allowed.
- COMMON
  - Only simultaneous writing of the same value is allowed.
- TOLERANT
  - Nothing happens if several processors try to write simultaneously.
- COLLISION
  - A special collision symbol is written if several processors try to write simultaneously.
- COLLISION+
  - A special collision symbol is written if several processors try to write simultaneous different values. (see COMMON)
- ARBITRARY
  - Some (random) value survives if several processors try to write simultaneously.
- PRIORITY
  - Processor with lowest PID will success, others fail.
- STRONG
  - A combination of the values is written,
    - e.g., ADD&WRITE, AND&WRITE, PREFIX-SCAN
    - Different variations have been suggested.

Examples on potency differences:

- Spreading a word to every processor (or to P memory locations).

  - CREW: every processor reads the same memory location: $O(1)$ time.
  - EREW: value is doubled (as in a binary tree) until all processors have read it: $O(\log P)$
- Maximum of an array.
  - CREW: $O(\log N)$
  - WEAK CRCW: $O(1)$
- Sorting
  - EREW: $O(\log N)$
  - STRONG CRCW: $O(1)$

2.3 PRAM "programming"

⇒ As in sequential programming, we'll use several abstraction levels.

- Describe the algorithm in a natural language and a picture.
- Describe the algorithm in an algorithm notation.
- Transform the algorithm to adapt with real world (machine and programming environment) restrictions.
- Write the algorithm in a programming language.
- Compile the program into machine language.

(Data)parallel algorithm notation

⇒ As sequential, with an additional statement to express parallelism.

```
for i ∈ 1..N pardo                                    1
        statement;                                    2
end pardo                                             3
```

- statement is executed once for each value of i $(1..N)$ (as in a sequential for-do).
- or

```
for each x ∈ A pardo      // A is an array/list      1
        statement;                                    2
end pardo                                             3
```

- All N executions are done in parallel, if we have at least N processors.
- Time complexity (T_{st} = time complexity of *statement*):
  - $T_{st}+O(1)$ if we have enough processors.
    - $\lceil T_{st} \times N/P \rceil + O(1)$ if we take P into account.
    - Remember Brent's theorem (page 10).

⇒ Different parallel executions may not disturb each other.

```
for i ∈ 1..N pardo                                                    1
        A[A[i]] = A[i];      // result in not clear (unless A is a permutation of 1
end                                                                   3
```

- If we need local variables (memory), we can use keywords *private* and *shared* to clarify the situation.

⇒ Creative freedom is allowed in algorithm notation as long as exactness and comprehensibility are maintained.

```
procedure Odd–even_mergesort(A : array[1..N]);
    if Processors = 1 then
        Sequential_mergesort(A);
    else
        par i = 1 to 2 do
            Odd–even_mergesort(i:th half of A});
        Odd–even_merge(halves of A);

procedure Odd–even_merge(A : array[1..N]);
    if Processors = 1 then
        Sequential_merge(A);
    else
        par i = 0 to 1 do
            Odd–even_merge(halves of odd/even (2n+i) elements of A);
        par i = 2 to N–1 by 2 do
            pipelined_compare–exchange (A[i], A[i+1]);
```

Parallel programming languages

⇒ Variety is huge, few established standards.

- For algorithms, (any) algorithm notation suffices.
- For practical applications, any high performance-oriented language (Fortran, C, C++) augmented with parallelism primitives will do.
- We'll describe some real languages/standards later on.

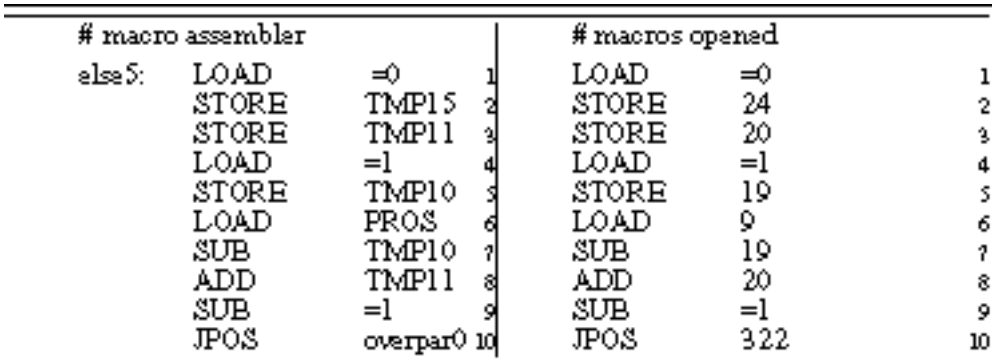
⇒ PRAM programming with paper (or with a PRAM emulator) can be done as easily as moving from sequential algorithms to sequential programs.

- Local and shared variables.
- Processor-ID (PID) to distinguish between processors.
- Synchronization.
- I/O is either forgot, or we'll use parallel I/O.
- Example: (Parallel Modula-2 for F-PRAM)

```
procedure oemerge(sharedvar S : array of word; Start, Length, Stride : word);
var         a, b : word;
            i, j, k, Length2 : register word;
begin
    Length2 := Length / 2;
    par i := 0 to 1 do
            oemerge(S, Start + i * Stride, Length2, Stride * 2);
    end;
    par i := 1 to Length2 - 1 do
            j := i * 2;
            a := S[Start + (j - 1) * Stride];
            b := S[Start + j * Stride];
            if a > b then
                S[Start + (j - 1) * Stride] := b;
                S[Start + j * Stride] := a;
            end;
    end;
    synchronize;
end oemerge;
```

PRAM machine language

- As any RAM machine language, possibly also LOADPID, and separate operations to access local and shared memory.
- Usually one shared program for every processor.
  - The same program is loaded to every processor node, processors will branch according to PID.
- We can use assembler as an intermediate stage.
- E.g., F-PRAM.



# macro assembler				# macros opened		
else5:	LOAD	=0	1	LOAD	=0	1
	STORE	TMP15	2	STORE	24	2
	STORE	TMP11	3	STORE	20	3
	LOAD	=1	4	LOAD	=1	4
	STORE	TMP10	5	STORE	19	5
	LOAD	PROS	6	LOAD	9	6
	SUB	TMP10	7	SUB	19	7
	ADD	TMP11	8	ADD	20	8
	SUB	=1	9	SUB	=1	9
	JPOS	overpar0	10	JPOS	322	10

Figure 2-3: (F)PRAM machine language

2.4 PRAM implementation possibilities

⇒ Using shared memory (memory reference: read or write) in one clock cycle is impossible.

- It has not succeeded even on uniprocessors since 1 MHz times at 80's.
- Today, we could achieve 20 MHz on DRAM, 300 MHz on (non-embedded) SRAM.
- In addition to DRAM latency, physical distances or large computers make access slow.
  - In 0.3 ns (3 GHz), the light will travel 10 cm in free space, electricity ≈7 cm in a coaxial cable, even less on circuit board, only (few/less than) cm on a semiconductor.

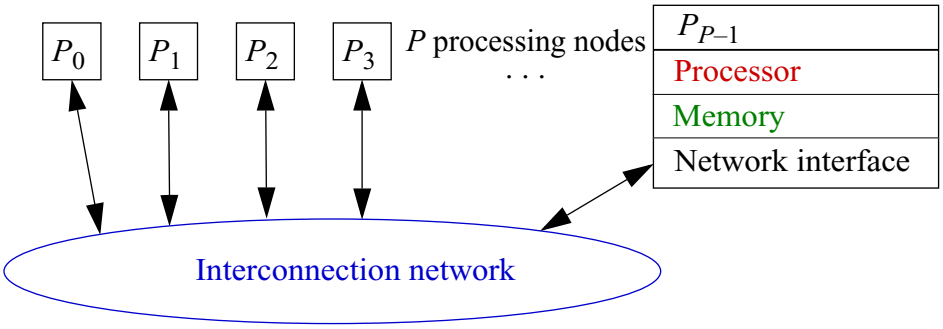
⇒ Moreover, building a P-port memory is prohibitively expensive/impossible if P is large.

Extra cost factor for P-port memory chips is $\Omega(P^2)$ (as VLSI area). [Forsell]

- E.g., let us consider technology for 16 Gbit (2 GB) memory chips.
  - It will yield 64 Mbit (8 MB) memory with 16 ports.
  - Moreover, each of the 16 processors will need 26 address lines and 2 data lines, totalling more than 448 pins for the 64 Mbit (8 MB) memory chip.
  - Packaging costs for a modest 4 GB memory (256 MB/pr) would be 100,000's €.
- At 64 ports, a single 4 Mbit (512 kB) chip would be about as complex (>1900 pins) as a large CPU.
  - 256 GB (4GB/proc) would take 0.5M chips, 1000m^2, and cost > 10^9€.
  - And the access latency would still be long...

PRAM can be implemented more easily via simulating the shared memory by distributed memory.

⇒ P processors, M memory banks.



- Alternatively, separate shared memory nodes and processor nodes.
- We might assume that $M = P$, i.e., each processing node contains a memory module.
  - Good: easier construction, less nodes, less communication connections.
  - Poor: more traffic in each node/connection, in real life, memories are slower than processors.
  - For reasonable performance, $M = c \times P$, where c is the speed difference factor between processors and memory (see example on page 24).
    - still, there can be c memory modules (channels) in a processing node.

Overloading (ylikuormitus)

⇒ Let us assume that a memory reference from/to a (virtual) shared memory takes h clock cycles.

- The computer has P physical processors.
- Each physical processor executes the tasks of h PRAM processors (h virtual processors per a physical processor).
  - The processor executes only one instruction at a time for each PRAM processor it is responsible of.
  - After each clock cycle it changes to the next PRAM processor.
  - After execution an instruction on all h PRAM processors, it starts over by executing the next instructions for each PRAM processor.

⇒ The memory references made by PRAM processors have occurred in h clock cycles.

- In algorithm notation:

```
while not all processors halted do                                    1
    for each thread i do                                              2
        PC_i := PC_i + 1;      // PC = Instruction Pointer           3
        if op = write then                                            4
            send write_reference                                      5
        else if op = read then                                        6
            send read_reference                                       7
        else                                                          8
            execute operation                                         9
    for each thread i do                                              10
        if op = read then                                             11
            receive read_reference                                    12
```

⇒ What shall we benefit?

- For each PRAM processor ("virtual processor") everything occurs in one clock cycle.
- The clock frequency of each PRAM processor is only $1/h$ of the real processor.
- There are $h \times P$ PRAM processors.

- Processing power is $(h \times P) \times (1/h) = P$, i.e., the same as with direct P processors.

$\Rightarrow$ If the program can utilize $h \times P$ processors, it will execute efficiently.

- h is called also parallel slackness.

How large h needs to be?

- Depends on the network and routing protocol.
- At least twice the diameter of the interconnection network.
- Even a bit more as the routing algorithm needs slackness to handle congestions.
- E.g., in a butterfly network: $O(\log P \log\log P)$.
- It has been done (Saarbrücken SB-PRAM, Paraleap XMT, Tera MTA / Cray XMT).
  - High cost, low clock.
- Same technique is used in GPU units, e.g., Nvidia G8x, etc.
- Bonus: no caches needed.

Hardware requirements for overloading

- Multithreading processor (switch after every clock cycle)
  - Implementation similar to superpipelining [Forsell].
  - SIMD helps (no need to handle several instructions streams).
  - Aligned / bank coordinated memory access helps (sequential access and/or access to different memory banks).
    - Vector operations.
- Huge memory bandwidth needed for full performance.
  - E.g., fully populated grids have too narrow bisection bandwidth, see Figure 1.6 on page 19.
  - (Partially) local memory access helps.

Lesson learned

$\Rightarrow$ A parallel algorithm should be designed to use as many processors as (efficiently) possible.

- PRAM is not complete utopia.
  - Especially if we use also local memories to decrease the traffic in the shared memory.

3 Parallel algorithms (in PRAM-notation)

Goals

Techniques

Some algorithms

3.1 Parallel algorithm design goals

Either

- Maximal speedup (and parallelism), or
- maximal speedup while still maintaining work-optimality.

More formally, an algorithm classification

- According to time complexity
  - NC: polylogarithmic time complexity, polynomial number of processors (Nick's class).
  - P: polynomial speedup
  - Note: different P than in sequential algorithms (solvable in polynomial time).
  - NC and P are not disjoint, they may even be same
    - NC $\subseteq$ P
    - NC $\stackrel{?}{=}$ P is one of the great unknowns (as sequential P $\stackrel{?}{=}$ NP).
- According to work optimality
  - E: efficient
  - A: polylogarithmic inefficiency (almost efficient)
  - S: polynomial inefficiency (semi efficient)
- Combining these we'll get six classes of algorithms, ENC, ANC, SNC, EP, AP, SP.
  - ENC would be nice.
  - EP is usually good enough.

3.2 Parallel algorithm design methods

$\Rightarrow$ Concentrate to (operations for) data, not (operations by) processors!
$\Rightarrow$ Remember sequential algorithm design methods.

Parallelizing sequential parts of an existing sequential algorithm.

$\Rightarrow$ This is not a real design method, but in real life this is what we'll face (as ad hoc programmers have sequentialized parallel problems).

- Suits well for linear algebra.
- Analysing for-do loops (and other sequential sections).
- If the sequential parts are independent, we can parallelize them
- Sometimes, inner loops are parallel, sometimes outer loops.
- Loop rearranging may help
- Two independent recursive calls can be executed in parallel
- E.g., matrix multiplication

$$C = A \cdot B \qquad c_{ij} = \sum_{k=1}^{N} a_{ik} \times b_{kj} \tag{4}$$

- Easy sequential algorithm and an easy parallelization.

```
for i := 1 to N do                    // => pardo                    1
    for j := 1 to N do                    // => pardo                2
        for k := 1 to N do                                            3
            C[i,j] := C[i,j] + A[i,k] * B[k,j];                       4
```

- $N \times N$ matrix, $O(N^3)$ sequential algorithm, $O(N)$ parallel algorithm with $O(N^2)$ processors.
- Which PRAM variant is needed? Exercise.
- Parallelizing innermost for-loop is not quite as straightforward (unless we use STRONG CRCW-model).
- However, the innermost product-sum can be evaluated in $O(\log N)$ time using $O(N)$ processors see Parallel tournament (page 33).

- Even with $O(N/\log N)$ processors, see Blocking (page 33).
- Thus, the whole algorithm in $O(\log N)$ time with $O(N^3/\log N)$ processors (exercise).
- For real computers and real input sizes, it is often enough to parallelize only one of the nested loops.
  - Decision first based on iteration dependencies (correctness), then based on memory access patterns (efficiency).
- In algorithms with several stages, we should parallelize all (demanding) stages to achieve full efficiency (processor utilization).

```
for i := 1 to N pardo                    // O(1)
    for j := 1 to N pardo
        statement1;                      // O(1)

for i := 1 to N do                       // O(N)
    for j := 1 to N pardo
        statement2;                      // O(1)
```

- Uneven parallelization: $O(N)$ time, $O(N^2)$ processors (but $O(N)$ with $O(N)$ processors).
- Although, if we have less processors, the implementations of the Brent's theorem may do the load balancing "automatically".

Divide-and-conquer (Hajoita-ja-hallitse)

⇒ Divide input in two parts, solve halves recursively in parallel, combine the results (in parallel).

- Familiar technique in sequential algorithms.
- Parallel recursion is terminated when either
  - input is trivial (as in sequential programming), or
  - there is only 1 processor left, when we can switch to a sequential algorithm (see Blocking (page 33) and algorithm on page 2.3).
- Subresults are combined to larger subresults on returning from recursion.
- Or before the recursion.

- E.g., mergesort

```
1 procedure mergesort(var A : array; first, last : index);
2     if (last-first) > 0 then
3         mergesort(A, first, (last+first)/2);
4         mergesort(A, (last+first)/2+1, last);
5         merge(A, first, (last+first)/2, (last+first)/2+1, last);
```

- Sequential algorithm: $T_s(N)=2\times T_s(N/2)+O(N)=O(N\log N)$
- Recursive calls at lines 3 and 4 can be executed in parallel (as they work on disjoint parts of the array).
- Using sequential merge, $T_p(N)=T_p(N/2)+O(N)=O(N)$, $O(N)$ processors, $O(N^2)$ work, not good.

⇒ Also combining of subresults must be parallelized!

- Combining is often more difficult than dividing.
  - Sometimes combining is trivial, though.
    - E.g., in search algorithms (only discoverer acts), especially using CRCW.
- In mergesort, combining is the merging phase, which is more difficult to parallelize.
  - If we could merge in $O(1)$ time using $O(P)$ processors, the sorting time would be in $T_p(N)=T_p(N/2)+O(1)=O(\log N)$ time, $O(N)$ processors, $O(N\log N)$ work.

  - Unfortunately merging in $O(1)$ time is impossible (using realistic memory models).
  - $O(1)$ amortized time is possible, but unfeasibly complex.
  - Merging in $O(\log N)$ or $O(\log\log N)$ time is much easier, but does not offer work optimality, unless we use less processors, see "Odd-even merge" (page 45).
- In quicksort, dividing (partition) is difficult to parallelize, no combining needed at all.
- Division can be made in more than two parts to reduce the number of stages.
  - E.g., division in $\sqrt{N}$ parts, combining in $O(1)$ time: $T(N)=T(\sqrt{N})+O(1)=O(\log\log N)$.
  - Obviously, combining might not be as easy any more, see raw power and waterfall techniques below.

Parallel tournament (turnaustekniikka)

- Also called balanced tree.
- If divide-and-conquer is a top-down approach, we can also apply a similar technique also bottom-up.
- We'll skip (recursive/parallel) dividing in parts, instead we'll start from ready "sequences" of length one element.
- Compare input elements pairwise, winner continues to the next round.
  - Definition of winner depends on application, e.g., a combination can be used.
- A stage can be done in $O(1)$ time using $N/2$ processors.
- Same is repeated again and again among the winners ($N/4$, $N/8$, ... pairs) until the ultimate winner is left.
- $\log N$ stages, each $O(1)$ time $\Rightarrow O(\log N)$ time, $O(N)$ processors.
- As in divide-and-conquer, more than two elements can be handled at each stage, see below.

Raw power (raaka voima)

- As fast as possible.
- "Overkill".
- Almost: using as many processors as possible.

$\Rightarrow$ We'll try to evaluate all possibilities at once.

- E.g., we'll compare all pairs simultaneously.
  - $O(N^2)$ comparisons in $O(1)$ time using $O(N^2)$ processors.
  - N input elements will transform to N^2 subresults!
- Combining may be hard to do fast, requires usually CRCW.
- Goal is $O(1)$ or logarithmic time algorithm.
- Rarely work-optimal.
- Is often used as a final stage of an algorithm, see below.

Blocking (lohkominen)

- Previous methods often result in unbalanced processor utilization, which implies non-optimal work.
  - E.g., at the beginning of a tournament, $N/2$ processors are used, but the number of active processors reduces on every round, last comparison is made by one processor only.
- We'll restrict parallelism appropriately to achieve work-optimality.
- Idea:

  - Less processors.
  - More work to do for each processor.
  - At the beginning, each processor (in parallel) evaluates its own block sequentially.
  - Switch to the fast parallel algorithm only when each processor has a single intermediate result.
  - In the end, the processors might have to do some local work for their blocks also.
- Usually used with other techniques, e.g., divide-and-conquer.

- E.g., in a tournament of $O(N)$ sequential work:
  - Actual tournament stage will take $\Omega(\log P)$ time.
  - To maintain work-efficiency, we can use at most $O(N/\log P)$ processors (if also the block part can be done in $O(\log P)$ time, less if it takes more time).
  - We'll choose $P = N/\log N$.
  - Each processor will have a $\log N$ -element block, sequential algorithm is used, $O(\log N)$ time.
  - Remaining $N/\log N$ elements will be processed using parallel tournament in $O(\log N)$ time using $N/\log N$ processors.

$\Rightarrow$ Whole algorithm in $O(\log N)$ time with $N/\log N$ processors, $O(N)$ work.

- If the sequential part with blocks is more than $O(N)$ time, smaller blocks are enough.

Waterfall technique (vesiputoustekniikka)

- Also called accelerated cascading.
- Combine the best parts of the previous methods.
- Switch to a faster algorithm after the size of input has shrunk enough to be executed faster using the given P.

Other methods

- Some basic algorithms, e.g., prefix sums (see page 39), binary search, and tree/path compaction, are useful as parts of larger algorithms. They often help at the combining stage of the algorithm.
- Randomization (breaking patterns), useful for real-world EREW-like variant to avoid memory congestion.
- Parallel Monte Carlo / genetic methods (all processors try (random) solutions).
- Sampling.
  - Take a (smallish, but as large as possible without disturbing the efficiency) sample on the whole data, analyse it using a fast algorithm (raw power).
  - Divide input according to the distribution of the sample.
  - Input will hopefully be divided more evenly to processors.
  - Helps on real data with inconvenient or unknown patterns.

3.3 Maximum finding

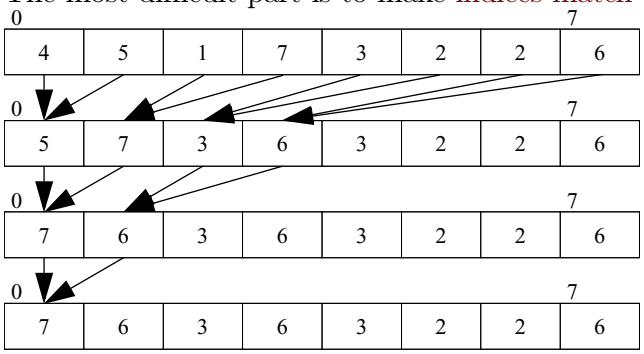
$\Rightarrow$ A very simple problem, examples on each technique.

- Input: a shared array $A[0..N-1]$
- Output: largest element or/and its index.
  - Sometimes (often) all processors need a copy of the result.
- Sequential algorithm $O(N)$.

Standard tournament

⇒ Compare elements pairwise, winner continues to the next iteration.

- After $\lceil \log N \rceil$ iterations, only one element is left.
- Intermediate results have to be stored somewhere.
- For each comparison, we need two values which are compared on previous iteration by different processors.
- If we want to leave original array intact, we'll use an auxiliary array.
  - Here we'll use the original for simplicity.
- Winner placement can be done in many ways, see below.
- Here we'll restore all winners to the beginning part of the array. The part reduces to half on every iteration.
- The most difficult part is to make indices match on every iteration.



- Iterations have to be executed in strict synchrony
  - We can assume this in PRAM algorithm notation (we can mention it, though). In real machines we need to have an explicit synchronization.
  - By some clever organization, the synchronization requirement can be easied, even removed (with auxiliary data structures).

```
function tournament–max(var A : array[0..N–1]);                    1
    for i := ⌈logN⌉–1 to 0 do                                      2
        for j := 0 to 2^i–1 pardo                                  3
            A[j] := max(A[j*2], A[j*2+1]);                         4
    return A[0];                                                   5
```

- If/when the input size N is not of form 2^k, we'll have to refine line 4 to, e.g.,

```
A[j] := max( ((j*2<N) ? A[j*2] : A[j]),                            1
             (j*2+1<N ? A[j*2+1] : A[j])) )                        2
```

- Time: $\log N$ (line 2) $\times O(1)$ (lines 3-4) $+O(1)$ (lines 1 and 5) $= O(\log N)$.
- Number of processors: $N/2 = O(N)$.
- Work: $O(N \log N)$, not work-optimal (inefficient by factor $O(\log N)$).
- EREW PRAM is sufficient.
- The same set of indices can be written in different ways:

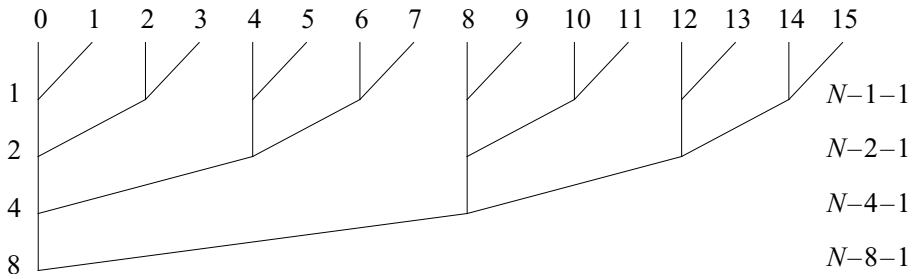
```
function tournament–max2(var A : array[0..N–1]);                   1
    i = N;                                                         2
    while i > 0 do                                                 3
        i := i/2;                                                  4
        for j := 0 to i pardo                                      5
            if j*2 < N–1 then                                      6
                A[j] := max(A[j*2], A[j*2+1]);                     7
            else if j*2 = N–1 then                                 8
```

```
                A[j] := A[j*2];
        end while;
        return A[0];
```

- Also, you may use any indices, or a new array to store the intermediate results.
- If counting twice does not hurt, modulo helps on boundaries.

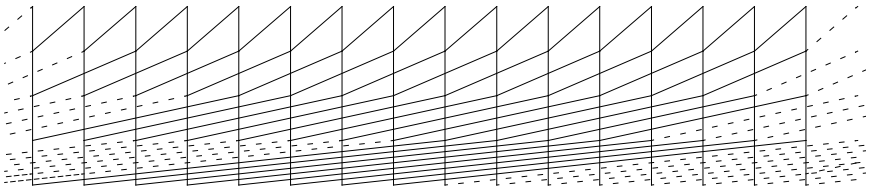
- E.g., using doubling/halving stride works well.

```
function tournament-max3(var A : array[0..N-1]);
    s := 1; // stride
    while s < N do
        for j := 0 to N-s-1 by s*2 pardo
            A[i] := max(A[i], A[i+s]);
        s := s * 2;
    end while;
    return A[0];
```



A variation: maximum for every processor

- Often, the maximum has to be spread to all processors (or indices of the array).
  - This is useful especially on EREW PRAM.
  - We could make the spreading by using another log$N$ "tree".
- But, on previous algorithm, most processors are idle most of time. They can be utilized in "concurrent spreading".
- Each processor evaluates its "local" maximum tree.



- Even if all processors make useful work during the whole execution, this is not work-optimal.

Divide-and-conquer

- Works actually like tournament, slightly different notation.
- Divide recursively until input is trivial.
- On returning from recursion, compare and return the larger one.

```
function divide_conquer-max(var A : array[0..N-1]; low, high : index);
    if (low = high) then
        return A[low];
    else
```

```
        pardo                                                          5
            x := divide_conquer-max(A, low, (high+low)/2);             6
            y := divide_conquer-max(A, (high+low)/2+1, high);          7
    return max(x, y);                                                  8
```

- Managing array boundaries and synchrony is easier.
- Parallelism representation possibly more difficult / inefficient.
- Time: $T(N)=T(N/2)+O(1)=O(\log N)$, $O(N)$ processors, $O(N\log N)$ work.

Blocking and tournament

- None of the previous algorithms is work-optimal.
- Without Concurrent Write, we cannot achieve $O(1)$ time with $O(N)$ processors, thus, we'll have to reduce the number of processors for work-optimality.

$\Rightarrow$ We'll first use $N/\log N$ processors, goal for $O(\log N)$ time.

- Idea: reduce the input to $N/\log N$, after which we'll use tournament in $O(\log N)$ time using $N/\log N$ processors.
- Each processor finds first the maximum of its own block of size $\log N$ sequentially (but all processors in parallel).

```
function blocking_tournament-max(var A : array[0..N-1]);               1
    for i := 0 to N/logN-1 pardo                                       2
        B[i] := A[i*logN];                                             3
        for j := 1 to logN-1 do                                        4
            B[i] := max(B[i], A[i*logN+j]);                            5
    return tournament-max(B[0..N/logN-1]);                             6
```

- After $O(\log N)$ time, we'll have an intermediate input of size $N/\log N$.
- Then we'll do tournament for the smaller input.
- Total time $O(\log N)$, $N/\log N$ processors $\Rightarrow O(N)$ work!
- EREW is still enough.

Raw power (raaka voima)

- Let us assume that any element could be the maximum.
- We'll prove other elements not to be maximum, only maximum is left.
  - Initialize an array of 1's of size N (a bit for every element of the input).
- Compare all pairs simultaneously (about $N^2/2$ pairs).
  - The smaller of a pair cannot be the maximum, thus mark it with 0 to the boolean array.
  - Draws are decided according to the index (below, the one with smaller index wins).
- Only the maximum value retained the 1.

```
function raw-max(var A : array[0..N-1]);                               1
    for i := 0 to N-1 pardo                                            2
        V[i] := 1;                                                     3
    for i := 0 to N-1 pardo                                            4
        for j := i+1 to N-1 pardo                                      5
            if A[i] < A[j] then                                        6
                V[i] := 0;                                             7
            else                                                       8
                V[j] := 0;                                             9
```

```
        for i := 0 to N–1 pardo                                    10
            if V[i] != 0 then                                      11
                return A[i];                                       12
```

- All stages in $O(1)$ time, $N^2/2$ processors, $O(N^2)$ work.
- Concurrent read is needed at line 4, concurrent write at lines 7 and 9.
- Only zeros are written concurrently, thus WEAK CRCW suffices.

Divide-and-conquer & raw power

- Divide-and-conquer can be used with division in more than 2 parts.
- Combining fast enough is harder, though.
- Using raw-power maximum, we can combine (find maximum of) M results with M^2 processors in unit time.
- If we have N processors, we can combine $\sqrt{N}$ subresults by raw-maximum.
- Divide input in $\sqrt{N}$ parts, solve them recursively, find maximum with raw-max.

```
function root–max(var A : array[0..N–1]; low, high : index);          1
    if (low = high) then                                              2
        return A[low];                                                3
    else                                                              4
        k := high–low+1;                                              5
        for i := 0 to sqrt(k)–1 pardo                                 6
            B[i] := root–max(A, low + i*sqrt(k),                      7
                                low + (i+1)*sqrt(k)–1);               8
        return raw–max(B[0..sqrt(k)–1]);                              9
```

- If N is not of form 2^{2^n}, we have to refine the algorithm a bit (exercise).
- Time $T(N)=T(\sqrt{N})+O(1) = O(\log\log N)$, N processors, $O(N\log\log N)$ work.

Waterfall = blocking & divide-and-conquer & raw-power

- Reduce N elements to $N/\log\log N$ elements sequentially in $O(\log\log N)$ time using $N/\log\log N$ processors (blocking).
- Solve the remaining $N/\log\log N$ elements with $N/\log\log N$ processors using divide-and-conquer & raw-power.

$\Rightarrow$ A work-optimal $O(\log\log N)$ time (WEAK) CRCW algorithm.

Using stronger CRCW models

- STRONG CW has a ready operation for maximum.
- PRIORITY CW can solve maximum easily in $O(1)$ time using $O(N+M)$ processors (M is the maximum possible value).
  - Requires very strict synchrony (or some other hard/software support).

```
function crcw_priority_max(sharedvar A : array[0..N–1] of [0..M]);    1
    sharedvar maxvalue, winnerindex;                                  2
    sharedvar counts : array[0..M] of index;                          3
    for i := 0 to M pardo                                             4
        counts[i] := –1;                                              5
    for i := 0 to N–1 pardo                                           6
```

```
        counts[A[i]] := i;                                          7
for i = M to 0 by -1 pardo    // process with largest i will win   8
        if counts[i] >= 0 then begin                                9
            maxvalue := i;                  // Priority CW          10
            winnerindex := counts[i];       // Priority CW          11
        end                                                         12
return (maxvalue, winnerindex);                                     13
```

Other similar problems

- Most previous algorithms can be used (with small changes) for many similar tasks.
- Especially all problems where the result is atomic, and combining is easy.
  - Finding, selecting, counting, sum, and, or, etc.
- Or, the algorithms can be used in opposite direction to spread data.

3.4 Prefix sum (alkusumma)

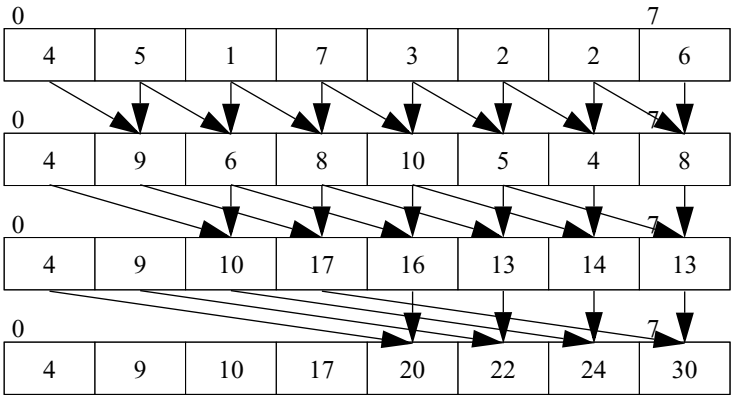
- Input: array $A[0..N-1]$ (or $[1..N]$).
- Result: array

$$(A[0],\ A[0]+A[1],\ \ldots,\ \sum_{j=0}^{i} A[j]\ ,\ldots,\ \sum_{j=0}^{N-1} A[j]) \quad \text{(inclusive prefix sum)}\ \text{, or} \tag{5}$$

$$(0,\ A[0],\ A[0]+A[1],\ \ldots,\ \sum_{j=0}^{N-2} A[j]) \quad \text{(exclusive, "0-prefix" sum)} \tag{6}$$

- Instead of $+$, any associative binary operation $\oplus$ can be used (max, and, mul, ...).
- E.g., (4 5 2 5 6) $\Rightarrow$ (4 9 11 16 22).
- E.g., (1 0 1 1 0 0 1) $\Rightarrow$ (1 1 2 3 3 3 4).
- Sometimes we need segmented prefix scan.
  - Additionally an N-element binary array, each 1 on the binary array resets the prefix. See below.
- Applications: counting, array/list compaction (removing empty elements), load balancing, radix sort, graph algorithms, etc.
- Algorithm similar to maximum for all (p. 36).
  - Every sum is formed by "own" tree of sums (branches are, however, shared).
- $O(\log N)$ time with N processors.
- EREW is enough (read/write in two steps, if needed).
- Use blocking to make it work optimal (exercise).
  - Not quite as straightforward as blocking in maximum.

```
procedure prefix_sum(var A : array[0..N-1]);                        1
    for i := 0 to ⌈logN⌉-1 do                                       2
        for j := 2^i to N-1 pardo                                   3
            A[j] := A[j-2^i] + A[j];                                4
```



- Again, synchrony is crucial, updating of $A[j]$ must proceed in stages.
  - In practical implementation, temporary local variables should be used on line 4.
- Array boundaries are more difficult if N is not a power of 2 (exercise).

Segmented prefix scan [Blelloch]

- Associative binary operator $\oplus$.
- Input array A, binary segment array S.
- Output: prefix scan of A for all segments having continous 0s in S.
  - Every 1 of the segment array S resets the prefix scan.
- Define a new binary operator $\oplus_2$ for value-binary pairs $\binom{x}{b}$.
- Operator $\binom{x}{a} \oplus_2 \binom{y}{b}$ is defined as follows:

$\oplus_2$	$\binom{y}{0}$	$\binom{y}{1}$
$\binom{x}{0}$	$\binom{x \oplus y}{0}$	$\binom{y}{1}$
$\binom{x}{1}$	$\binom{x \oplus y}{1}$	$\binom{y}{1}$

- Resulting $\oplus_2$ is an associative binary operator for value-binary pairs $\binom{x}{b}$.
- Thus, we can use the same (parallel) prefix scan algorithm.

3.5 Merging and sorting algorithms

$\Rightarrow$ Parallel sorting can be approached in several ways (as sequential sorting).

- We'll present:
  - Raw power
  - Mergesort (with a couple of possible approaches to merging in parallel).
  - Sampling bucket sort.
  - Parallel Quicksort.
  - Radix sort.
- Some of the algorithms can be applied also for message-passing environment.

Parallel "bubblesort" (odd-even transposition)

- Compare-exchange odd and even pairs N times.

- $N/2$ processors, $2N = O(N)$ time, $O(N^2)$ work.

"Parallel" quicksort

- Quicksort is divide-and-conquer, thus the independent recursive calls can be executed in parallel.
- Partition before the recursive calls is more difficult to parallelize.
- Thus only $\log N$ speedup is achievable easily.
- For large data sets and a couple of cores it works, although the memory bandwidth can be a limiting factor.
- Exercise in OpenMP / Java threads.
- Partition of distributed data actually can be parallelized
  - If we can fairly reliable take a pseudomedian as a global pivot value, we can do the partition among the processors recursively in $\log P$ stages, each requiring N/P time and $O(N)$ copying of data.
  - Overheads are quite large (pseudomedian, transferring data)
  - See page 48.
- Combinations of quicksort, sampling, bucketsort, and mergesort work.

Raw power sort (by ranking)

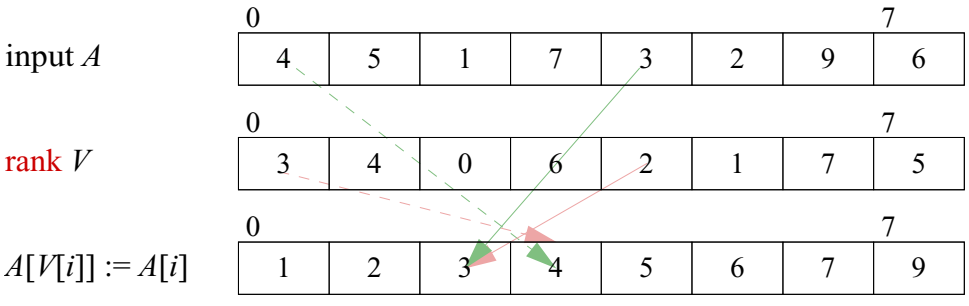
⇒ Presents PRAM at its best and worst!

- Takes advantage of STRONG ADD CRCW.

⇒Compute the correct location of each element at once:

- Count how many smaller elements there are in the array.
- I.e., the rank (rankkaus, sijoitus) of each element.
- Ranks are evaluated as in raw-max: compare all pairs, increase the rank of the larger element by one (vs. zero the smaller in raw-max).
- Several increasings of the same element at once (STRONG ADD CRCW needed).
- After ranking, we'll know the number of smaller elements for each element, i.e., the location of each element.
  - If order is not total, the draws would have to be solved.
- $O(1)$ time, $O(N^2)$ processors, $O(N^2)$ work.

⇒ We'll see later that ranks can be counted also more efficiently.




```
procedure raw-sort(var A : array[0..N-1]);                          1
    for i := 0 to N-1 pardo                                         2
        V[i] := 0;                                                  3
    for i := 0 to N-1 pardo                  // ranking the elements 4
        for j := 0 to N-1 pardo                                     5
            if A[i] precedes A[j] then                              6
                V[j] := V[j] + 1;            // STRONG ADD CRCW     7
```

```
                                                                    8
    for i := 0 to N-1 pardo                  // sorting the rankings   9
        A[V[i]] := A[i];                                            10
```

Parallel mergesort (lomituslajittelu)

- Actual sort is trivial:

```
procedure mergesort(var A : array; first, last : index);            1
    if (last-first) > 0 then                                        2
        pardo                                                       3
            mergesort(A, first, (last+first)/2);                    4
            mergesort(A, (last+first)/2+1, last);                   5
        merge(A, first, (last+first)/2, (last+first)/2+1, last);    6
```

- Merging in parallel is interesting, we'll present a few example algorithms.
  - Merging in $O(N)$ time (sequentially): $O(N)$ time full sort (N processors: $O(N^2)$ work).
  - Merging in $O(\log N)$ time: $O(\log^2 N)$ time sort.
  - Merging in $O(\log\log N)$ time: $O(\log N \log\log N)$ time sort.
  - Merging in $O(1)$ (amortized) time: $O(\log N)$ time, N processors: $O(N \log N)$ work.

Merging by ranking

- We assume elements to be distinct (use index to resolve draws).
- Let us define the rank of an element x in an array $A[0..N-1]$ as the number of smaller elements in array A.

$\Rightarrow$ Computing of the rank is much easier if A is in increasing order (sorted).

$$\text{rank}(x,A) := \max\{i \mid A[i] \leq x\} \tag{7}$$

- Using one processor: using binary search in time $O(\log N)$.
- With P processors, we can divide into $P+1$ parts (P division points) instead of two.
- One processor finds the correct interval, others do not. On next iteration all processors concentrate on the new (correct) interval. Exercise.
- Thus parallel "binary" search ("$P+1$ -ary search") in time

$$T(N,P) = T\left(\frac{N}{P+1}\right) + O(1) = O(\log_{P+1} N) = O\left(\frac{\log N}{\log P}\right) \tag{8}$$

- Using raw power, we can find one rank in $O(1)$ time using $O(N)$ processors.

```
function raw-rank(x : element; var A : array[0..N-1]);              1
    if x < A[0] then                                                2
        return 0;                                                   3
    else if x >= A[N-1] then                                        4
        return N;                                                   5
    else                                                            6
        for i := 0 to N-2 pardo                                     7
            if A[i] <= x and x <= A[i+1] then                       8
                return i+1;                                         9
```

- If needed, we can refine this with one processor writing (instead of return) and the rest of processors reading the result.
- CREW suffices.
- Later we'll show how to do this more efficiently.
- Asymptotically, any $P = N^{1-\epsilon}$ is enough, e.g. $P = \sqrt{N}$.

$$\frac{\log N}{\log(N^{1-\epsilon})} = \frac{\log N}{(1-\epsilon)\log N} = \frac{1}{1-\epsilon} = O(1) \tag{9}$$

Merging with ranking

- Input: readily sorted arrays A and B (often halves of the same array).
- Output: Sorted array C with all the elements of A and B.

Using rank to merge:

- In a sorted array, rank and index are the same!
- Rank of element $A[i]$ in array A is i.
- Rank of element $A[i]$ in array B is rank($A[i]$,B).
- Rank of element $A[i]$ in the result array C is i+rank($A[i]$,B).
  - We can place every element to the final array independently!

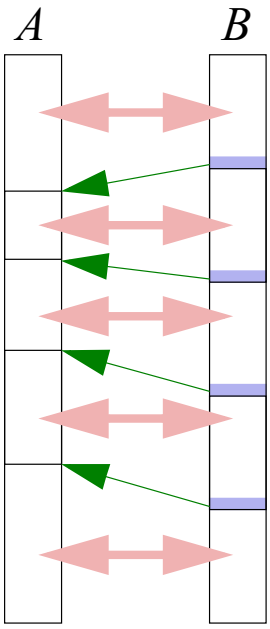
⇒ For the whole merge, we'll need the rank of each element of A in B, and the rank of each element of B in A.

```
function rank-merge(A, B : array[0..N-1]) : array[0..N*2-1];
    for i := 0 to N-1 pardo
        C[i + rank(A[i], B)] := A[i];
        C[i + rank(B[i], A)] := B[i];
    return C;
```

- *Rank-merge* can be easily converted to restore elements back to A and B and/or to merge halves of a single array (synchrony needed).
- We need CREW PRAM since N simultaneous ranking processes read the same array (using binary search) in parallel (though only constant penalty on EREW).
- If parallelization and synchronization are made carefully, the merging can be done in place.
  - But we need N processors, all of which use $O(1)$ helper space, thus it actually uses $O(N)$ extra space.
  - Later, with less processors, we need anyway $O(N)$ extra space and have to move elements to/from a helper array.
- $O(N \log N)$ work for merging, not work-optimal.
- More accurate analysis of rank-merge-sort with $P = N$, $P = N^2$, arbitrary P as an exercise.

Merging in $O(\log N)$ time, $O(N)$ work

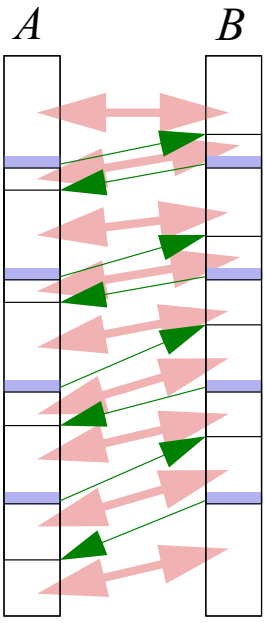
- Input: sorted arrays A and B (of length N)
- Choose regularly $N/\log N$ elements of B.
- Rank each of these (with sequential binary search $O(\log N)$ time) in A (one sample for each processor, total $N/\log N$ processors).
- Now we have $N/\log N$ pairs of sections, each of which can be merged sequentially.
  - $a_1..a_{j_1}$ and $b_1..b_{\log n}$ | $j_i = \text{rank}(b_{i\times\log n}, A)$
  - $a_{j_1+1}..a_{j_2}$ and $b_{\log n+1}..b_{2\log n}$
  - ...
  - $a_{j_{\log n-1}+1}..a_n$ and $b_{(n-1)\log n+1}..b_n$
- From the section boundaries, we know the location of the merged section in the new array – merging tasks are independent.
- On average, the lengths are $O(\log N)$, thus the whole merge in $O(\log N)$ time.



- Unfortunately, the sections of A can be longer than $\log N$ if data is uneven.
- Either:
  - Symmetric ranking & partitioning:
  - Choose $N/\log N$ elements of both A and B.
  - Rank each of these (with binary search) on the other array.
  - Now we have to merge $2\times N/\log N$ pairs of sections of length at most $\log N$ (average $\frac{\log N}{2}$).
- Or:
  - Repartition the (few) too large sequences.

$\Rightarrow$ Merging $O(\log N)$ time, $O(N)$ work.

- Sort: $O(\log^2 N)$ time, $O(N \log N)$ work.



Merging in $O(\log\log N)$ time, $O(N)$ processors, $O(N \log\log N)$ work

- Utilizes faster 2-step ranking algorithm *root-raw-rank*.
- Take regularly $\sqrt{N}$ samples of each array A and B.
- Rank samples of A in samples of B (not in whole B!).
  - $\sqrt{N}$ ranks on $\sqrt{N}$ elements with N proc in $O(1)$ time (*raw-rank* (p. 42)).
- Same for samples of B in A (as in symmetric ranking above).
- Now we have $2\sqrt{N}$ subsequences, but boundaries are still inaccurate (we only know in which block of the other array the samples belong to).
- Rank each sample of A in the subsequence of B it belongs to.
  - $2\sqrt{N}$ ranks on $\sqrt{N}$ elements with N procs in $O(1)$ time (raw-rank).
- Same for samples of B in A.
- Now we have $2\sqrt{N}$ subsequences with accurate boundaries in $O(1)$ time.

- Apply the algorithm recursively to each of $2\sqrt{N}$ subsequences (of average length $\sqrt{N}/2$ with $\sqrt{N}/2$ processors for each subsequence.
- At the bottom of recursion, we have single elements to merge ($O(1)$ time with N processors.
- $T(N)=T(\sqrt{N})+O(1) = O(\text{loglog}N)$.

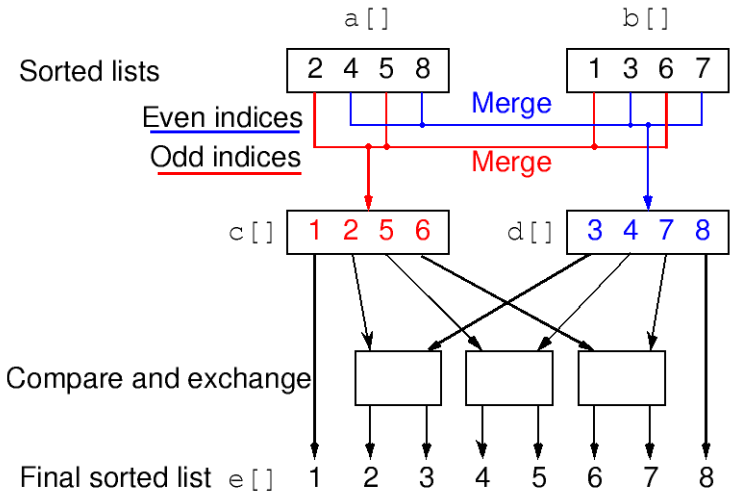
```
function root-raw-rank(x : element; var A : array[0..N-1]) : index;
    if x < A[0] then
        return 0;
    else if x >= A[N-1] then
        return N;
    else
        for i := 0 to sqrt(N) -1 pardo
            B[i] := A[i* sqrt(N)]
        block = raw-rank(x, B);                    // O(1) time with sqrt(N) proc
        brank = raw-rank(x, A[block * sqrt(N) .. (block+1) * sqrt(N) ]); // O(1)
        return block*sqrt(N) + brank;
```

Merging in $O(\text{loglog}N)$ time, $O(N)$ work

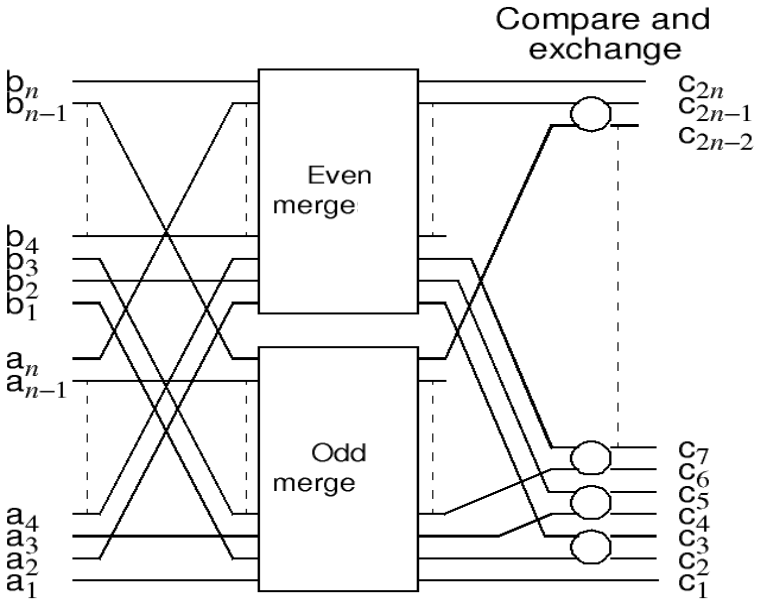
- $N/\text{loglog}N$ processors.
- Partition A and B to blocks of size $\text{loglog}N$.
- Rank the (arrays of) boundaries ($N/\log\log N$) in each other with the previous algorithm ($O(\text{loglog}N)$ time).
- Rank each of the boundaries with one processor within the corresponding block of the other array (of length $O(\text{loglog}N)$). ($O(\text{logloglog}N)$ time with binary search).
- Now we have accurate boundaries (ranks) of $2\times N/\text{loglog}N$ pairs of sections of length at most $\text{loglog}N$.
- Merge each pair of sections independently by one processor ($O(\text{loglog}N)$ time).
- Results a $O(\log N\text{loglog}N)$ time, $O(N\log N)$ work sorting algorithm.
- Work and time optimal merge, work optimal sort!

Odd-even merge

- Batcher 1968: odd-even merge and bitonic merge.
- Recursive divide-and-conquer approach for merging.
- Input array halves A and B.
  - In practice, halves of the same array are named A and B for easier reference.
- Merge (recursively) odd elements of A and odd elements of B; and
  merge (recursively) even elements of A and even elements of B.
  - Merging is done in place.
- After these merges, consecutive pairs may be out of order, we'll check order of each pair, swap if needed.



- Merge in time: $T(N)=T(N/2)+O(1)=O(\log N)$, $O(\log N)$ space (recursion).



Parallel odd-even merge informally.

```
procedure Odd–even_merge (A : array[0..N−1]);
    pardo
        Odd–even_merge(halves of odd elements of A);
        Odd–even_merge(halves of even elements of A);
    par i = 1 to N–2 by 2 do
        compare–exchange (A[i], A[i+1]);
```

Parallel in place odd-even merge procedure using stride (FPM):

```
procedure oemerge(var S : array; First, Length, Stride : index);
  par i := 0 to 1 do
     oemerge(S, First + i * Stride, Length/2, Stride * 2);
  par i := 1 to Length/2−1 do
     j := i * 2;          // j := 2 to Length−2 by 2
     if S[First + (j−1) * Stride] > S[First + j * Stride] then
        swap(S[First+(j−1)*Stride], S[First+j*Stride]);
```

OEM-sort performance

- Mergesort with odd-even merge utilises at most $N/2$ processors, executes in $O(\log^2 N)$ time, and thus uses $O(N\log^2 N)$ work, which is inefficient by factor of $O(\log N)$.

⇒ We can improve the efficiency by reducing the number of processors.

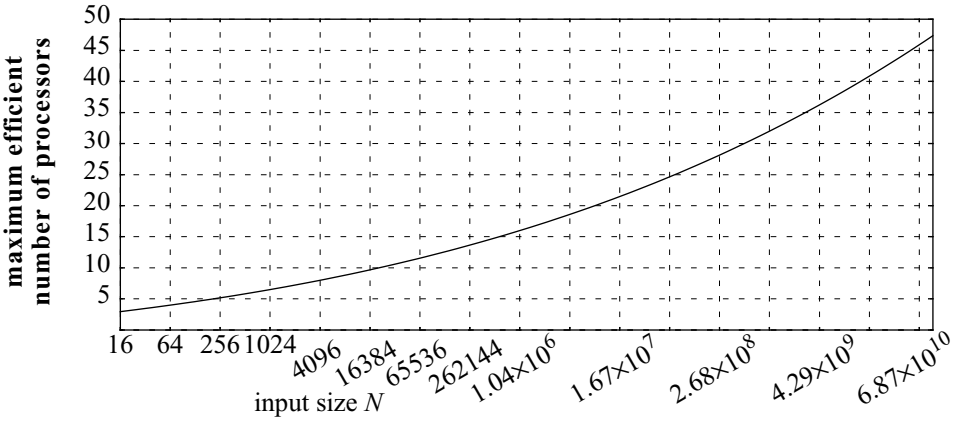
- If there are less than $N/2$ processors, we can switch to sequential sort/merge as soon as we run out of processors.
- The recursive sort branches according to P.
- Also, merging can run out of processors, thus also the merge will branch according to P.
- Time complexity will be

$$T(N,P) = O((N/P) \times (\log^2 P + \log(N/P))). \tag{10}$$

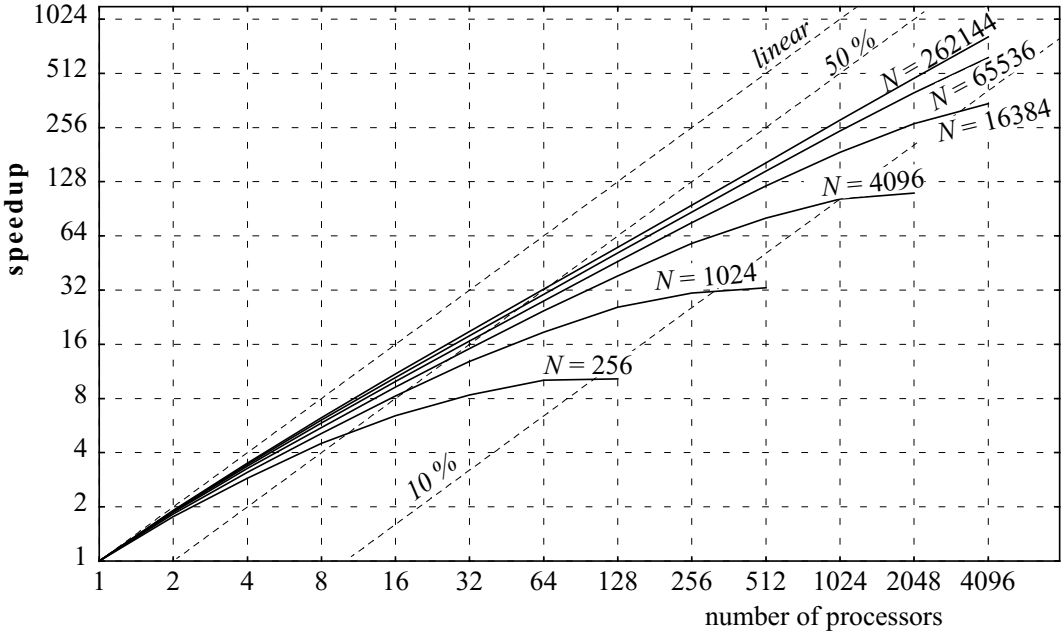
- In theory, we cannot utilize very many processors efficiently.
- E.g., to ensure 50% efficiency, we would have to settle for

$$P \leq 2^{\sqrt{\log N}} \tag{11}$$

- Plotted:



- In practice, though, we can efficiently use slightly more processors, as the slow recursion tails are removed if N is much larger than P.
- Measured performance on F-PRAM:



Cole's optimal parallel mergesort (1986)

- The first almost practical time and work-optimal $O(\log N)$ sort.
  - First asymptotically optimal was Ajtai, Komlós, Szemerédi (AKS) 1983.

⇒ We do not need a $O(1)$ time merging, a merging with $O(1)$ amortized cost for each phase is sufficient.

- The merge operations in different stages of sort can be pipelined.
  - We collect samples (border values, "cover") in different stages.
  - We collect the ranks of the samples in halves of data.
  - According to ranks of samples we can do the next stage faster.
- Because of large constants, this Cole's sort is faster than odd-even mergesort (or bitonic) only if $N > 10^{21}$ [Natvik].
- See, e.g., Jájá or Akl.

Sampling parallel bucket&mergesort

- Let us assume that $N \gg P$, each processor holds a block of data.
- Each processor (quick)sorts local block of data.
- Each processor samples its sorted block of data (fixed intervals, e.g., P samples).
- Samples are sorted in some fast (parallel) way, or sequentially by one processor.
- According to the sorted samples, the (master) processor(s) decide(s) $P-1$ pivot points (values).
- Each processor partitions its part of input to other processors according to the pivot points.
- Each processor receives one subsection of input from all others.
- Each processor merges the P segments of its own section.
- In shared memory model, we need some amount of additional space.
- In message passing model, we need all-to-all communication.

Parallel partition in quicksort

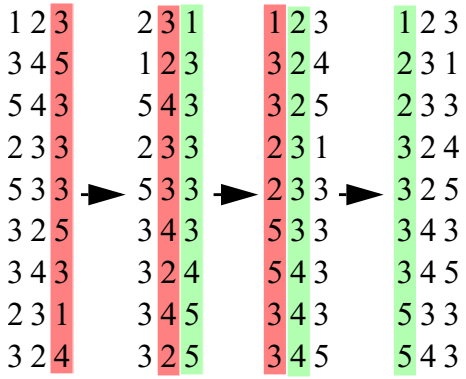
- Let us assume that $N \gg P$, each processor holds a block of data.
- Select (by sampling) a pivot element/value, broadcast it to all processors (of the group).
- Each processor divides its block of input according to the pivot.
- Split the processors to equal "low" and "high" parts.
- Low half of processors send (store) their high part of data to a corresponding "partner" processor of the high half. High half of processors "send" their low part of data to corresponding processor of the low half.
  - Now low and high halves of the machine have low and high values, respectively.
  - In data parallel PRAM, there is no need to "send" anything, we just use parallel loops and indexing. But "sending" might be easier to understand.
- Apply the algorithm recursively on both halves.
- After $\log P$ recursive stages we run out of processors and each processor has a disjoint section of data, which it can sort locally.
- Each poor selection of pivot point may waste up to half of processors, up to double the execution time.
  - Median of a largish sample is needed, it may be useful to sort locally first. On later recursion calls, merging is enough.
- $O(\frac{N}{P}\log P + \frac{N}{P}\log\frac{N}{P})$
- Using multiple pivots and multiple groups (instead of 1 pivot and 2 groups) reduces the number of stages and thus amount of communication needed.

Radix sort in parallel (kantalukulajittelu)

⇒ Possibly the fastest sequential sort if keys are reasonably short and input is large.

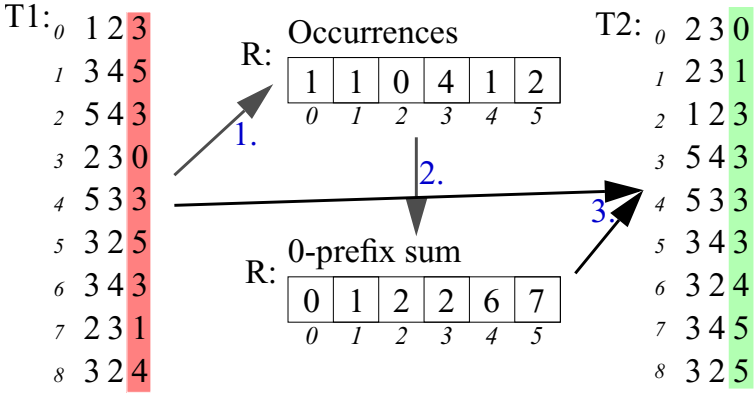
- Sequential time $O\left(\lceil \frac{m}{r} \rceil (n+2^r)\right)$, where m is key size (in bits) and r is radix size (bits).

- Sorting in stages:
  - Divide key in parts.
  - Sort according to the least significant part.
  - Sort according next least significant part.
  - ...
  - Sort according to the most significant part.



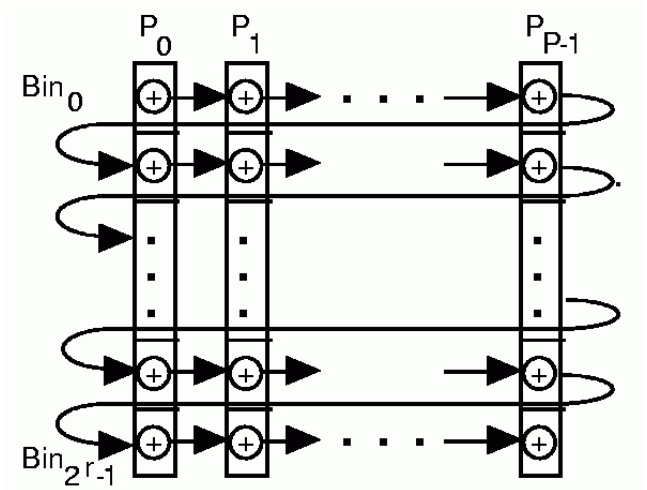
- Sorts have to be stable, i.e., the order of elements with the same subkey has the be sustained.
- If/when keys are not integers, we'll use the bit representation of keys. r bits at a time yields 2^r buckets, r is typically 12-20.

- First count the number of each of the subkeys.
- Compute a 0-prefix sum (exclusive prefix sum) of the count array.
- Prefix sum tells us into which position each "bucket" will be stored.
- Contents of each "bucket" will be stored in original order.
- Radix sum location is increased after each assignment.



Parallelization

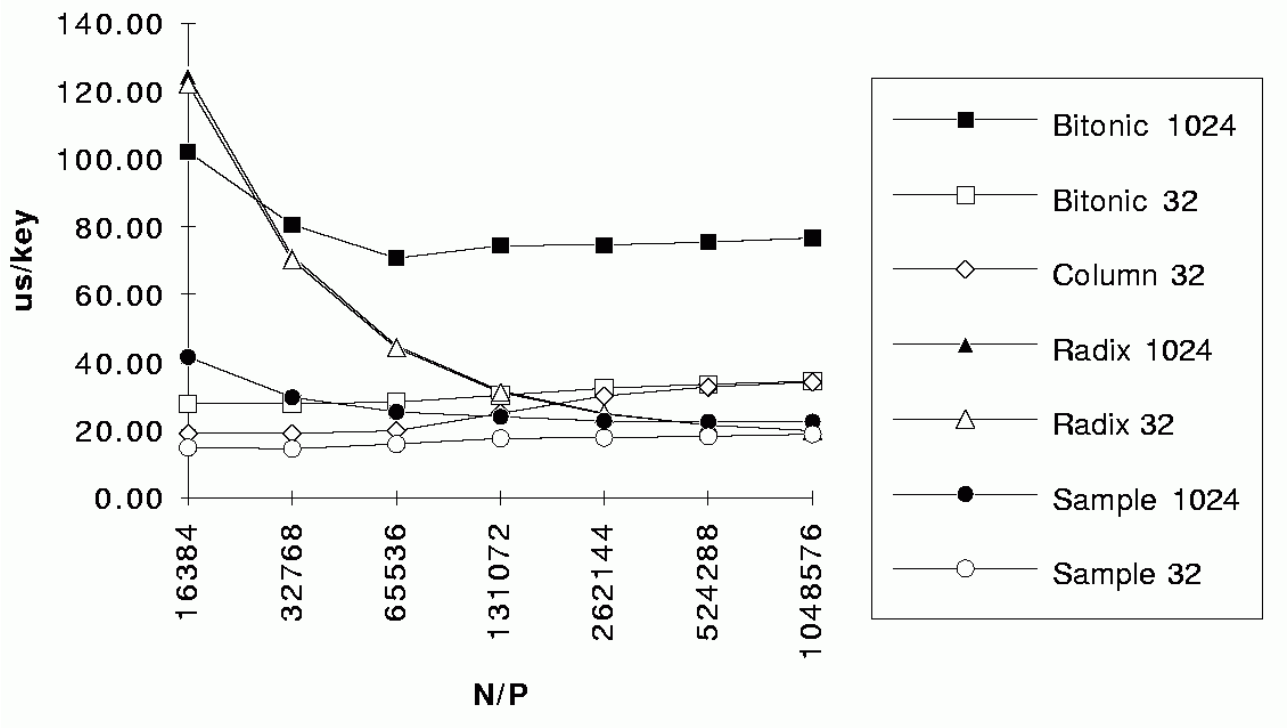
- If several processors count occurrences in parallel, the prefix sum needs to be counted for every $P \times 2^r$ buckets.
- Result like below, but linear (sequential) scan is too slow.



Prefix in three stages

- Prefix sum each row to last column ($2^r \times P/P = 2^r$ time).
- Broadcast all values of the last column to all processors (2^r (or skip in CREW)).
- Prefix sum the last column (for a moderate P, use sequential algorithm).
- Evaluate final prefix sums by adding also the previous row sums. (2^r)
- Assignment stage of the local input as in sequential version.
  - Processes can work independently.

⇒ Comparison of different sorts on CM-5 [Culler&al] (32 or 1024 processor CM-5, 128 MOPS each, 20 MB/s/link data connection (fat tree)).



3.6 String matching

- Search a string M (of length m) from another string T (of length n).

⇒ Relatively easy work compared to the size of the input.

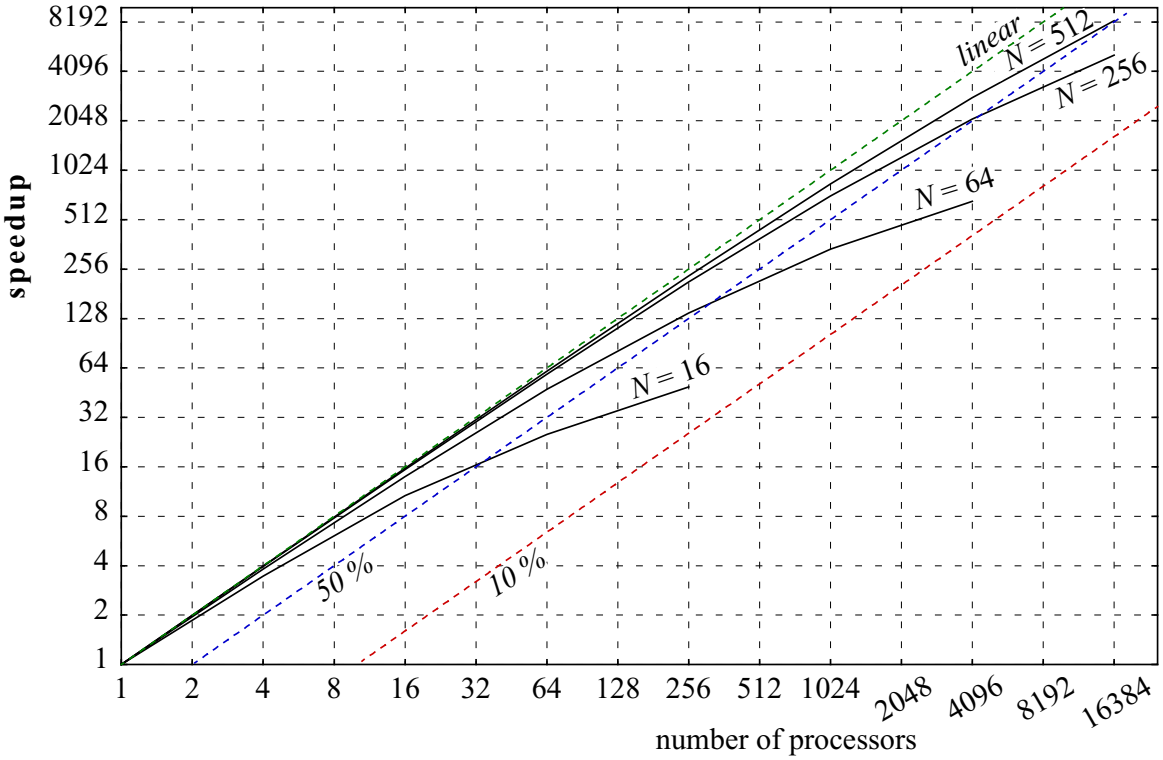
- Extensively studied sequential algorithms.
  - $O(n)$ or even $O(n/m)$ time, small constant factor.
  - Sequential fast algorithms are already memory bandwidth limited.
- Thus, parallelism is needed only if n is huge and we are in a hurry.
- Then we can partition T into P parts, and do the search in parallel on each part of T at a time.
- There is no interprocessor communication, thus searching can be done in a network of workstations.
- By using more sophisticated methods and $O(n)$ processors we can do in $O(\log m)$ time.
- Approximate matching is much more laborious than exact matching, thus parallel computing is more feasible then.
  - E.g., genome projects.
  - Still partition of text usually suffices.

3.7 Numerical problems

- In practical implementations, the memory access pattern is the most important factor for best performance.
  - Retrieve the data from memory in sequential order (row-wise in C, column-wise in Fortran).
  - Reuse the elements in cache.

Matrix multiplication

- Sequential basic algorithm is easy to parallelize to a time $O(N)$ algorithm with N^2 processors, see page 31.
- In a real computer, input spreading will take a bit more time.
- There will be more and more data to spread the more processors we have.
- However, there is no communication during the execution.
  - F-PRAM performance, see below.
- Innermost dot-product can be parallelized with tournament technique to a time $O(\log N)$ algorithm for N^3 (or $N^3/\log N$) processors.
- Also, the more sophisticated recursive Strassen-based matrix multiplications can be parallelized relatively easily.



Solving a set of linear equations, matrix inversion, LU decomposition, etc.

- These are more difficult even as sequential algorithms (but time complexity is the same $O(n^3)$ as matrix multiplication).
- Computing different results is not independent, instead each element of the result depends on results of the previous row and column.
- There are several SNC algorithms, see, e.g., Jájá: Section 8.8.
- Work-optimally (50%) we can easily utilize N processors.
- Iterative methods parallelize even better than direct algorithms.
- E.g., LU-decomposition

- Decompose matrix A to a lower-diagonal matrix L and an upper diagonal matrix U such that $L \times U = A$.
- Useful stage to solve set of linear equations, matrix inversion, etc.
- Computation proceeds by columns of U and L, all columns/rows depend on the previous ones.
- Only loops that can be parallelized are of form $j..n$, i.e., shorten during the computation.
- See example in OpenMP at course www-page.

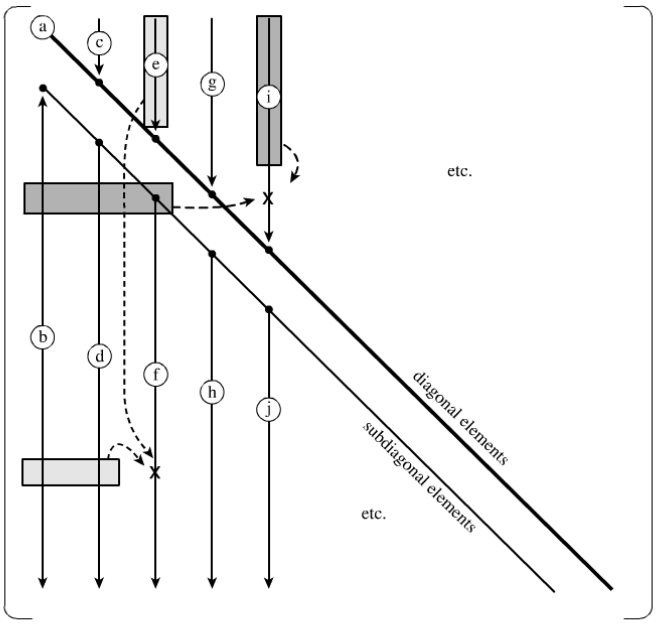


Figure 2.3.1. Crout's algorithm for LU decomposition of a matrix. Elements of the original matrix are modified in the order indicated by lowercase letters: a, b, c, etc. Shaded boxes show the previously modified elements that are used in modifying two typical elements, each indicated by an "x".

[Numerical Recipes]

3.8 Graph, tree, and list algorithms

- See any book on parallel algorithms

⇒ As the linked structures are more sequential by nature, we need some techniques to shortcut long lists.

- Pointer jumping
  - to find ranks of list nodes
  - to find roots of trees
    - Update successor of each node to successor's successor.
    - Repeat $\log N$ times.
- Algorithms working with matrix representation of graphs often appear to be straightforward to parallelize, but there may be dependencies between loops.
  - E.g., Floyd/Warshall for All-to-all shortest paths:
    - Only the second loop can be parallelized easily - each row of each stage is computed in parallel, up to $O(N)$ parallelization.
    - The inner loop can also be parallelized (at a cost), but the threads need communication.
    - Kŭcera: $O(\log^2 N)$ time, $O(N^3)$ processors.
  - All-pairs = N independent Dijkstra's algorithms.
  - For moderate P, parallelizing one loop is enough.
- Divide-and-conquer works well if combining of subresults can be done efficiently, e.g., planar Convex hull
  - Sort points $\{p_1, p_2, \ldots, p_n\}$ according to the x-coordinate.
  - Divide to two subproblems according to the x-coordinate $\{p_1, p_2, \ldots, p_{n/2}\}$ and $\{p_{n/2+1}, \ldots, p_n\}$ and solve those recursively in parallel.
  - In combining, we need to find the upper and lower point pairs that combine the two convex hulls (upper and lower common tangent).

- At least one of the points of a pair is the uppermost of one subhull (maximum in log$N$ time). The other point can be found in log$N$ time sequentially (like binary "search") by checking if the point is correct or not. Select the correct from two candidate pairs. Same for bottom.
- $O(\log^2 N)$ time, $O(N/\log N)$ processors (using blocking), work optimal.
- Aggarwal: $O(\log N)$ time, $O(N)$ processors
- Goodrich: $O(\log N)$ time, $O(N/\log N)$ processors if the points are readily in sorted order.

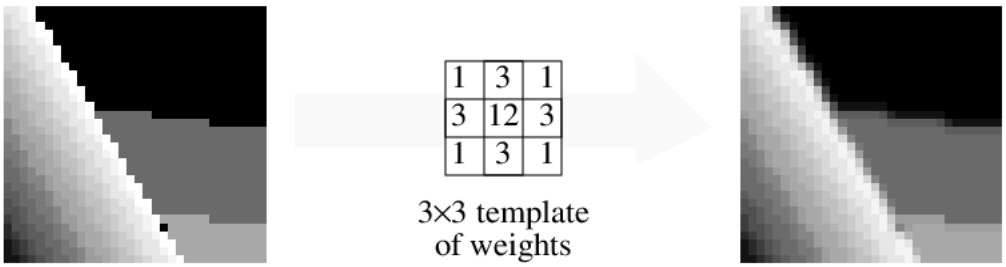
3.9 Other problems to solve in parallel

Image/video processing

- Most image processing tasks are embarrassingly parallel.
- Often the limit is memory bandwidth.
- Image processing often needs relatively little computing time compared to input size (linear time complexity).
  - There are exceptions, though.
- Even mass-market image/video/audio processing software (VLC, GIMP, Photoshop, MATLAB, etc.) support SMP computers and/or GPU computing.

Example: image smoothing

- A simple method to reduce noise and soften hard edges.
- The new value of each pixel is a weighted average of the pixel itself and neighbouring pixels.
- Typical size for neighbourhood is 3×3 or 5×5 pixels (or Manhattan-distance of, e.g., 2). For some applications (e.g., heatmap generation it can be much larger).



- Smoothing does not really improve the picture.
- By doubling resolution and/or number of colours, image appears better.

Parallelization

- Very easy: each resulting pixel can be computed independently, very easily in CREW, quite easily also in EREW.
- For real computers, loading of input may take longer than actual operation.
  - Especially if only one processor does the input.
- We need parallel I/O.
  - In GPGPU graphics, if the image under manipulation can be displayed directly from device memory, we can skip the I/O stage altogether. (See below).
- For reasonable efficiency, each processor reads its own part of the input.
- Memory bandwidth is an expensive resource in real computers.
- We should use shared memory at most to border areas (or use message passing).
- In the shared memory we need only those parts of the image that are used by several processors.

- Whole image is divided blocks.
  - Stripes could be easier to parallelize, but for narrow stripes, the common border area needed by several processors would increase (compared to blocks), [stripes_vs_blocks.ods].
- Only border areas of the blocks are needed by several processors.
- The larger the block, the larger the relative size of the (private) insides of the blocks.
  - $N \times N$ pixel picture, P processors ($\lfloor\sqrt{P}\rfloor \times \lfloor\sqrt{P}\rfloor$), each responsible of $b \times b = \lceil \frac{N}{\lfloor\sqrt{P}\rfloor} \rceil \times \lceil \frac{N}{\lfloor\sqrt{P}\rfloor} \rceil$ pixels.
- Border areas a $M-1$ pixels wide, combined area is $2N(M-1)\sqrt{P} - (\sqrt{P}-1)^2(M-1)^2$.

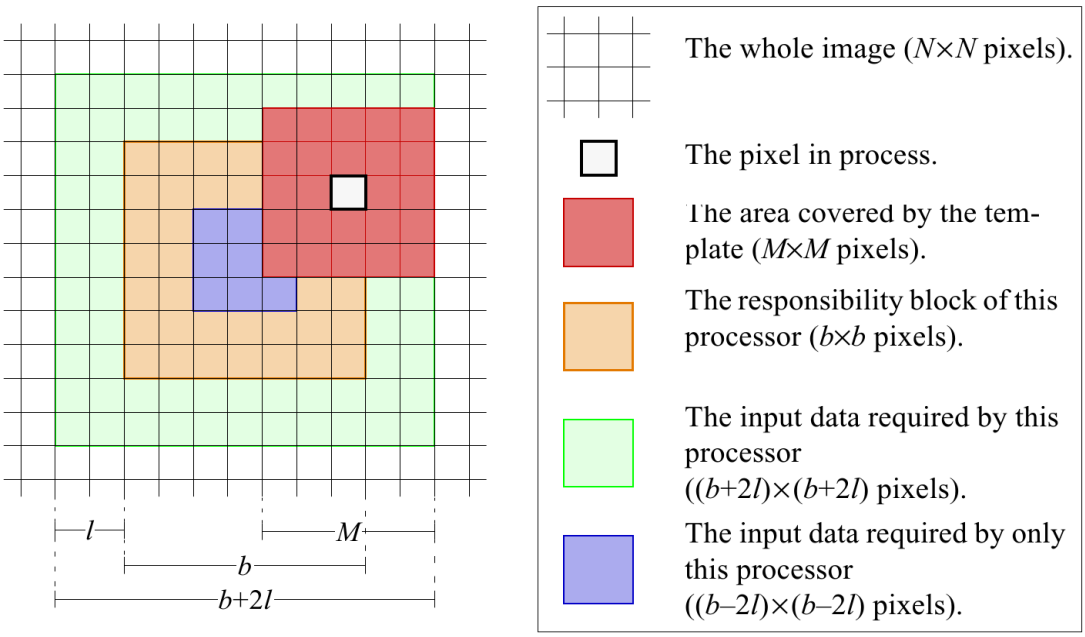


Figure 3-15: Examples of the variables in the image smoothing, 5×5 template, 7×7 pixel block for each processor.

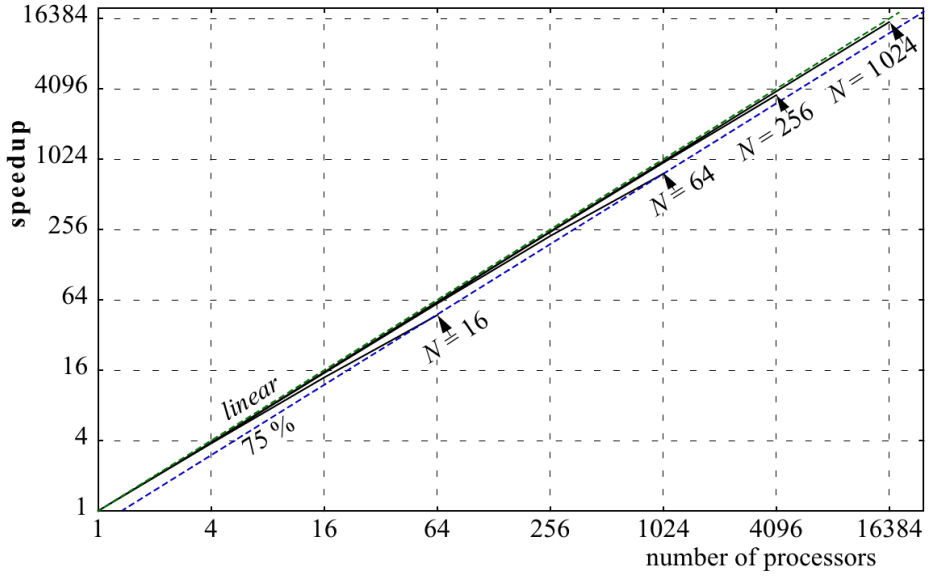


Figure 3-16: Speedup of the image smoothing as a function of the number of processors for different input sizes, $M = 5$.

4 Practical shared memory programming: OpenMP

- Committee result (as HPF, MPI-2).
- Standardization effort for SMP computers.
- Goals to support also fine-grained parallelization (especially parallelization of loops).
- Is implemented by OS threads, e.g., pthreads.
- Most features as directive/comments as in HPF.
- `!$OMP` in Fortran
- `#pragma omp` in C/C++
- Specify parallel sections of a program, coordinate access of shared data.
- I.e., guide compiler to parallelize loops (or other).

```
!$omp parallel do private(i)
      do i = 1,n
            ...
```

- Goal is to make (initial) parallelization easy.
- Incremental parallelization, keep compatibility to sequential (non-OpenMP) compilers.
- Responsibility of correct function is left to programmer (as always).
- Fortran, C, C++
- Windows, Linux, MacOSX, Solaris, HP-UX, ...
- http://www.openmp.org/
- gcc, g++, gfortran 4.2 or later: -fopenmp
- Intel C/C++ Compiler: -Qopenmp
- Visual C++: /openmp
- http://openmp.org/wp/resources/
- http://openmp.org/mp-documents/Intro_To_OpenMP_Mattson.pdf
  - Youtube videos, slides, and source code available.
- http://software.intel.com/en-us/articles/getting-started-with-openmp
- http://software.intel.com/en-us/articles/more-work-sharing-with-openmp
- At laskuri3: `gcc -fopenmp`

```
// trivial parallelization of for loop
// i is private automatically
#pragma omp parallel for
for (i=0; i < N; i++) {
  int temp = 0;         // local variable declared here
  array[i] = ...        // shared array declared outside par loop
} // parallel block ends here
```

```
// You can specify each "outside" variable to be private
// or shared among execution threads
#pragma omp parallel private(i) shared(k)
{
// several threads execute here
        int tid = omp_get_thread_num();
        int p = omp_get_num_threads();
        while (k < n) {
                // computation here
                // update of shared variable k has to be protected
               #pragma omp critical
               {
                        // only one process executes this block at a time
                       k++;
}       }       }
```

- Number of threads is taken either from
  - set with `num_threads()` option for `omp parallel`, or
  - `omp_set_num_threads()` function
    - which would break compatibility with non OpenMP compilers unless `#ifdef`:ed.
  - user/system `OMP_NUM_THREADS` environment variable, or
  - system number of processors.
- See also atomic, single, master, reduction, schedule.

5 Taking the real world into account

5.1 Problems of PRAM

⇒ Unit time memory references of the PRAM are very nice to use, but too expensive/impossible to implement.

- We can forget everything beyond this point if someone invents (and sells) a "PRAM" that competes with MPP in price/performance.
  - Cray/Tera MTA/XMT is (was) close to PRAM (but too expensive).
  - GPGPUs may approach this.
    - Currently still hierarchical.
    - Programming model might mask out part of the problems.

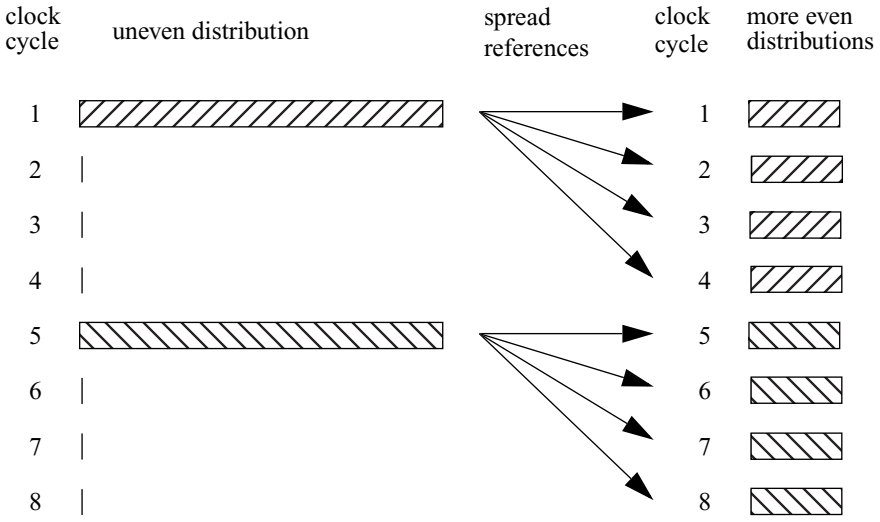
Hard assumptions of PRAM

⇒ Memory reference in one clock cycle!

- This is not possible even in uniprocessor computers.
- Additionally, multiplexing wires from several processors takes more time.
- Several processors take more physical space, thus wire lengths and transmission times grow.
- Caches are problematic in parallel computers.
  - Bus snooping works only in simple bus (or increases traffic).
- By overloading and deep pipelining we can hide the latency.
  - Number of virtual processors increases, which may cause extra work for algorithm designer.
  - Current commercial processors are not suitable (except GPUs).
  - Volume of memory traffic may increase.
  - Still, the most promising technique.
  - Current GPUs use this already!

Unlimited volume of memory references

- If/when all processors refer to the shared memory simultaneously, there is too much traffic.
- In a typical PRAM algorithm:
  - Each processor refers to shared memory in the same clock cycle,
  - then does some local computation for a few clock cycles, and
  - then again all processors use shared memory.
- Distribution of references is uneven: 0,0,P,0,0,0,0,0,P,0, etc. references/clock cycle.
- We could assume that a memory channel of width P/c would suffice (c = average time between references):



- Problems:
  - Some processors will encounter additional delays.
  - References do not occur simultaneously, thus the stronger (> COMMON) CRCW -models do not work.
  - We have to be careful that the next references are not mixed with earlier ones: $A[V[i]] := A[i]$;
- Solutions:
  - Delay is accepted: memory references occur only after memory latency (against PRAM ideology).
  - Do not guarantee simultaneous memory references (which is against PRAM ideology).
    - Require programmer to add explicit synchronization.
  - Allow even congestion and additional delays if processors send too many references.

Random references to any memory location

- In real computers, the memory is implemented in banks.
- Bandwidth to each memory bank is about $1/M$ of the whole memory bandwidth.
  - If all processors refer to the same bank, there will be a long queue.
- Two possible solutions:
  - Hash the memory locations randomly to memory banks, bad congestions are then unlikely.
    - bank(m) = $h(m)/M$, addr(m) = $h(m) \mod M$
    - Locality and sequentiality of memory references are lost!
  - Leave the problems for programmer (to be optimized).
    - Programmer can (has to) specify the allocation of arrays in memory.
    - E.g., cyclic, clocks, random.
    - This is used in MPP (HPF) and GPU
    - In OpenMP, the programmer can specify allocation of tasks (data) to threads.

Concurrent read and write (CR and CW)

- If/when the references to the same memory location/bank are serialized, the concept of concurrent access can be forgotten.
- All CW-tricks can be forgotten, unless special hardware is added to support specifically them.
- Packet combining at intermediate nodes and/or memory modules could help, but is has not been implemented (outside laboratories).
  - Especially reliable and sub-clock cycle accurate synchronization cannot be guaranteed.
- Special operations (implemented in network nodes (hardware)) for CW are available in many MPP machines.
  - Barrier synchronization, broadcast, gather, logical and, maximum, prefix scan, etc.
  - MPI library calls implemented by proprietary libraries (or direct proprietary library calls).
  - Implicit barrier synchronization.

Synchrony up to a clock cycle

- In PRAM-algorithms we exploit the fact that processors execute in lock-step up to accuracy of a clock cycle, e.g.,

```
for i := 1 to N/2 pardo                    1
      A[i] := A[i] + A[2*i];               2
```

- This is possible in SIMD (Single Instruction Stream, Multiple Data Streams)-approach.
  - Large scale SIMD is almost extinct nowadays.
- In real MIMD (Multiple Instruction Streams, Multiple Data Streams)-computers, this accuracy is impossible, especially if we have less processors.
- We can, however, add barrier synchronizations.

```
for i := 1 to N/2 pardo                    1
        localvar tmp := A[2*i];            2
        synchronize;                       3
        A[i] := A[i] + tmp;                4
end pardo                                  5
```

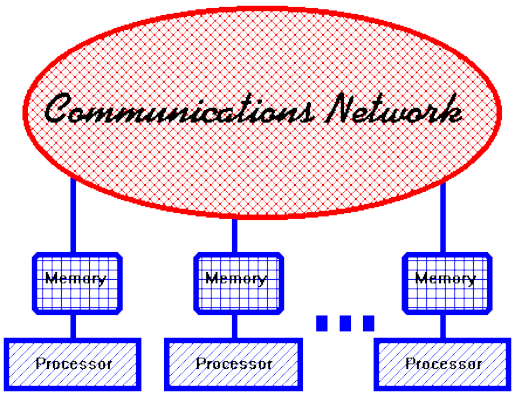
- Synchronization cost (time) is machine-dependent, but rarely negligible.

5.2 BSP (Bulk Synchronous Parallel Model)

- http://www.bsp-worldwide.org/
- Valiant (CACM 33,8)
- McColl, Bisseling

A BSP computer consists of

- A set of processor-memory pairs
- A communications network that delivers messages (or remote memory references) point-to-point.
- A mechanism for the efficient synchronization of all, or a subset, of the processors.



The parameters that define performance

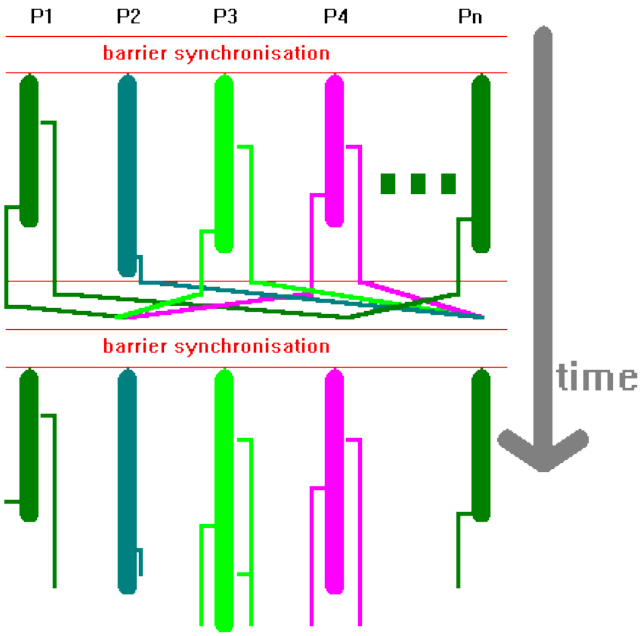
- p = number of processors
- l = the cost, in steps, of achieving barrier synchronization (depends on network).

- g = the cost, in steps per word, of delivering message data
- s = processor speed (number of steps per second)

Computation on a BSP Computer

⇒ Computation and communication are divided into supersteps.

- In each superstep, the processors do local computation and request (non-blocking) for non-local data.
- References are guaranteed to be ready only at the next superstep.
- The length of supersteps is at least l.



Cost Model

- The time for a superstep S is determined as follows.
- Let the work w be the maximum number of local computation steps executed by any processor during S.
- Let h_s be the maximum number of messages sent by any processor during S, and h_r be the maximum number of messages received by any processor during S.

⇒ The time for S is then at most $w+\max(h_s,h_r)\times g+l$ steps.

- The total time required for a BSP computation is obtained by adding the times for each superstep to obtain an expression of the form $W+H\times g+S\times l$, where W, H, and S will typically be functions of n and p.

A BSP Programming Environment

- BSLlib, see http://www.bsp-worldwide.org/
- Paderborn University BSP-Library, see http://wwwcs.upb.de/~bsp/
- BSGP: Bulk-Synchronous GPU Programming (BSP on CUDA): http://www.kunzhou.net/#BSGP
- MulticoreBSP: http://www.multicorebsp.com/

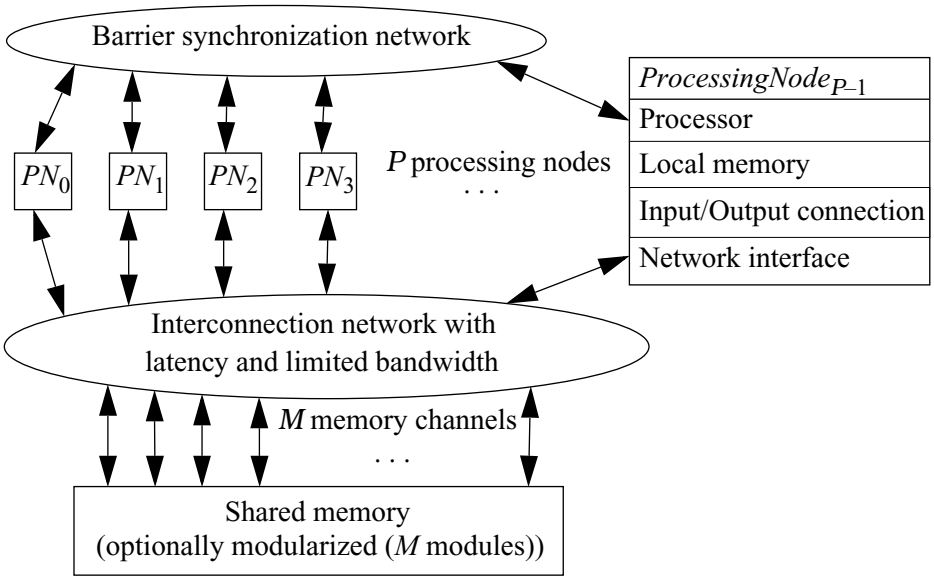
5.3 F-PRAM

⇒ Based on PRAM, but:

- No exact synchrony, nor immediate shared memory access, no CR/CW.
- Volume and speed of memory references are restricted.
- All restrictions are visible for the programmer/user.

F-PRAM logical components

- A real implementation may differ, e.g.,
- shared memory distributed to processing nodes,
- central I/O,
- no separate synchronization network.



Memory references

- An F-PRAM processor cannot directly refer (LOAD, STORE, ADD, etc.) to a shared memory.
- The processor has to fetch data from shared memory by a special future operation.

Motivation of the future reference:

⇒ Instant memory reference is not possible.

- If a memory reference takes L clock cycles, could processor do something useful meanwhile?
  - Yes, even make more memory references!
  - Especially interleave computation and communication.
- How to express such delayed (background "process") memory reference?
  - How the result is given back to the referring processor?

Future -mechanism

- Originally Halstead (1985) for Multilisp language:
  - A Compromise between lazy and aggressive evaluation: background (parallel) evaluation.
  - A functional (list) programming is conveniently parallelized as list components can be evaluated in parallel.
- In F-PRAM, instead of a direct memory reference, we'll do

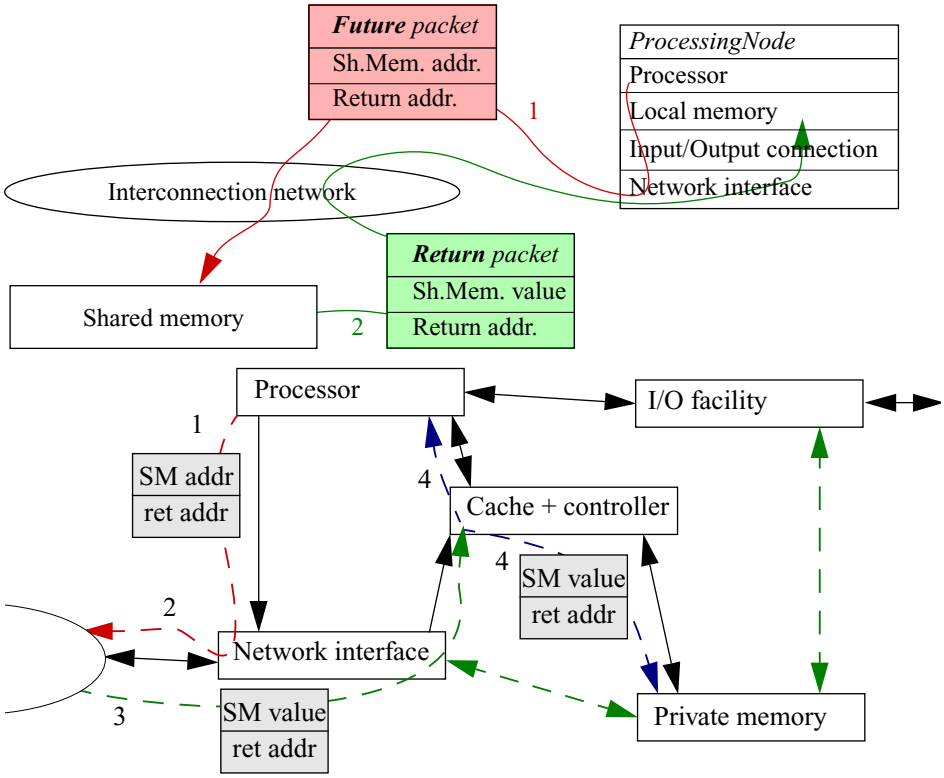
```
future local_variable := shared_variable;
```

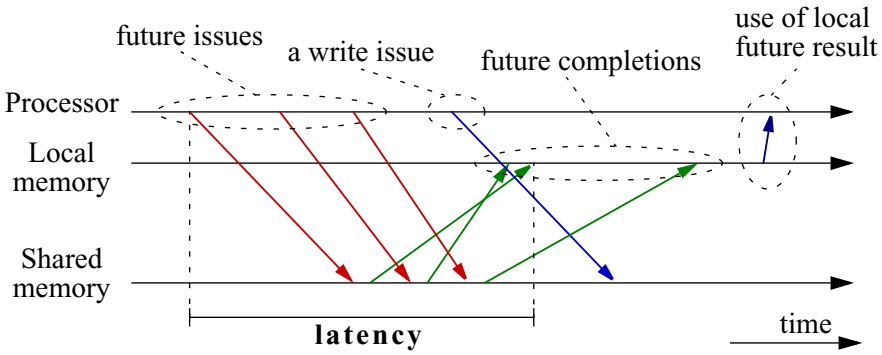
1

⇒ Operation only initiates a transfer from shared memory to local memory.

- The future -operation itself takes only unit time.
  - Processor continues on next instruction (statement).
- To the local variable is written a special not-available value, which cannot be used (see below).
- Communication hardware and shared memory take care to actually implement the memory reference.

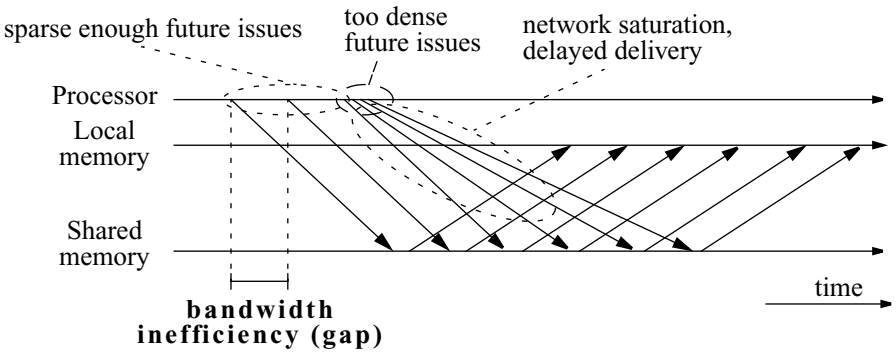
- After about (at most) L clock cycles, the value should finally arrive to originating processing node.
- Communication hardware places the value directly to the local memory by replacing the not-available value by the result.
  - Now the local variable can be read as any variable.
  - Processor does not get any acknowledgment about completion of the future.
    * The programmer should find enough work for the processor to fill up the time until the reference is ready.
    * Processor can send as many concurrent memory references it needs to.
    * If/when the processor tries to read the unfulfilled reference (local "not-available value"), it stops until the result arrives.
  - See figures below.
- Write requests are easier, no return value is needed.





Restricted memory channel

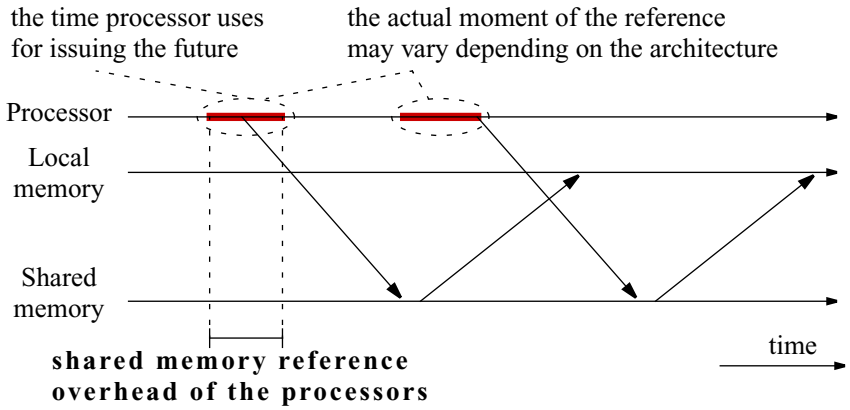
- The interconnection network has a restricted capacity so that at most P/B memory references can be handled at shared memory on a clock cycle (or at any bisection of network).
  - B is called bandwidth inefficiency, or gap.
  - If every processor makes references at every B clock cycles, everything should go on smoothly.
  - If there are too many references, the network will be congested and references will be delayed.

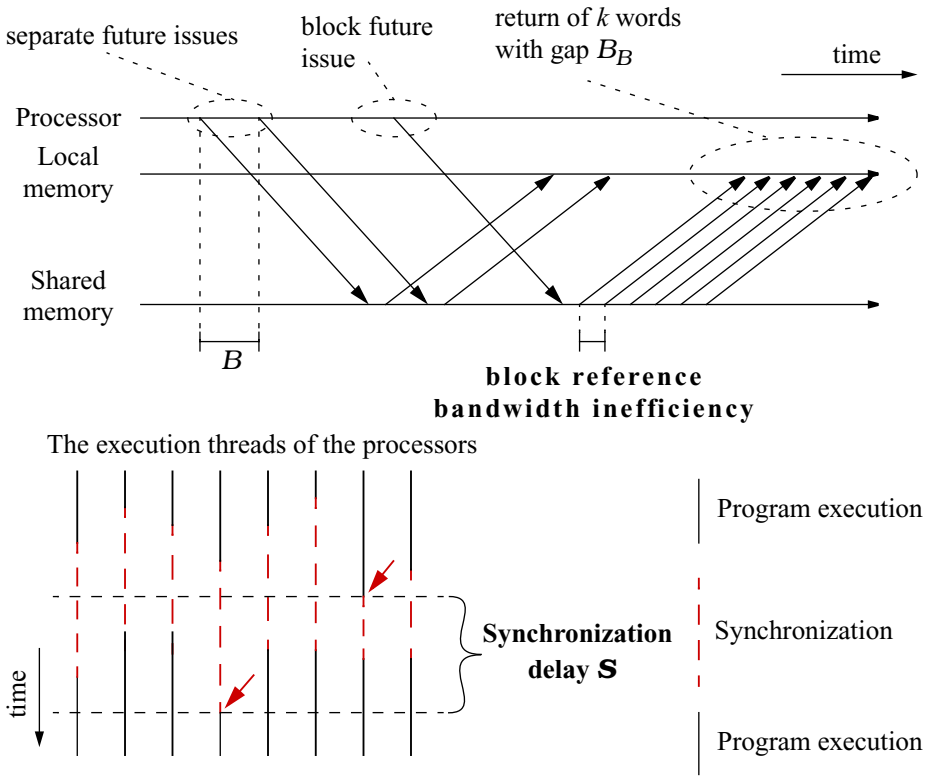


Other parameters

Primary parameters		**Typical/default values**
P	The number of processors	
L	The latency of shared memory references	$L_0 + 2\times Ø$ [a]
B	The bandwidth inefficiency	$O(1)$ ... $O(P)$
Secondary parameters		
B_P	The shared memory reference overhead	$O(1)$
B_B	The block reference bandwidth inefficiency	$\leq B$
B_V	The single variable bandwidth	1
B_M	The single memory module bandwidth	1
M	The number of shared memory modules	P
B_{IO}	The input/output bandwidth inefficiency	1
S	The synchronization delay	$\geq L$

[a] the diameter of the network





Miscellaneous

- If a processor tries to read an unfinished future, it will wait.
  - It could poll the future, but it is rarely useful.
- Processors execute asynchronously.
- The order in which the shared memory references occur is not guaranteed (unless there is a synchronization between).
- In addition to all processor barrier, we can do a "user space synchronization" for any set of processors by a short procedure.

Programming language FPM

⇒ Modula-2 subset (like Pascal) with additional:

- par..do (as in PRAM).
- future, fwrite
- synchronize
- var and sharedvar
- read, write (input/output of local variables).

⇒ All F-PRAM parameters are available as read-only variables, values are assigned at program load time.

- Enables possibility to make programs that adapt to different environments.

```
if L < 50 then                                                    1
        fast version;                                             2
else                                                              3
        version for slow communication;                           4
                                                                  5
if B < 20 then                                                    6
        fast version;                                             7
else                                                              8
        version that minimizes the volume of communication;       9
```

```
for i := 1 to L do // not good                    1
        unit time work                            2
for i := 1 to L/LBL do // better                  3
        unit time work                            4
```

- Compiler will calculate the value for LBL (Loop Body Length)

F-PRAM emulator

- A simple machine language with added future and synchronization.
- Integer and floating point arithmetics.
- Each processor executes the same machine language program.
- On initialization, all F-PRAM parameters are set, emulation is done according to these.
- Program is executed instruction by instruction for each processor.
- Communication network is simulated clock cycle by clock cycle.
- Either simple queue according to B and L, or a specified network and routing algorithm.
- We can specify any communication graph topology.
- We can specify any routing algorithm for each network node.
- Parallel input/output from/to a file.
- We can print used clock cycles for performance statistics.
- Written in C.

Compiler

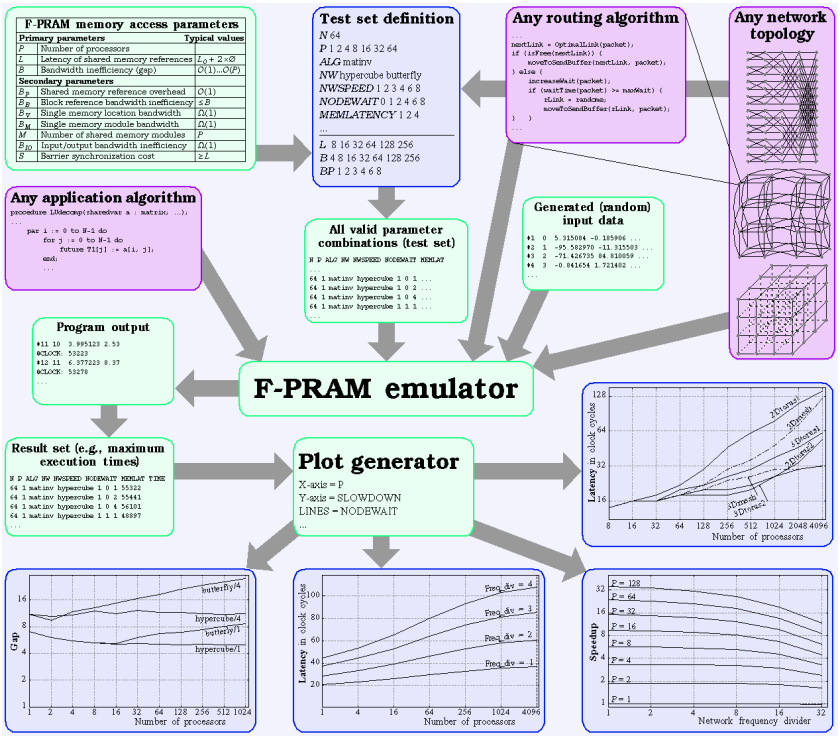
- Compiles from FPM to (macro)assembler.
- Especially takes care on handling the given number of processors and implementation of par..do statements and local/shared variables.
- Original PRAM version was able to keep processors in lock-step execution by filling all it-then-else parts to equal number of instructions.

Automated test system

- The whole F-PRAM is designed to study the impact of shared memory features.
- Thus:

```
foreach N = 1..1000000 do
       foreach P = 1..1000000 do
              foreach  L = 1..1000000 do
                     foreach B = 1..1000000 do
                            A[N,P,L,B] = exec_time(N,P,L,B);
```

- Same for all ten parameters, different programs, network topologies, routing algorithms, prioritizing strategies, etc.
- In each phase we might need regeneration of input, compilation, result post processing, etc.
- Even if we used exponential steps, there was huge number of simulations (and several months of cpu time).



- Examples.

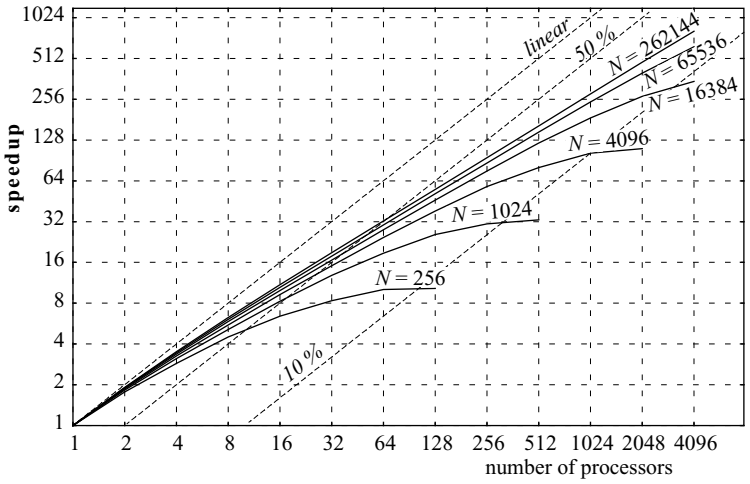


Figure 0-1: Speedup of odd-even mergesort as a function of the number of processors for different input sizes. Both scales are logarithmic.

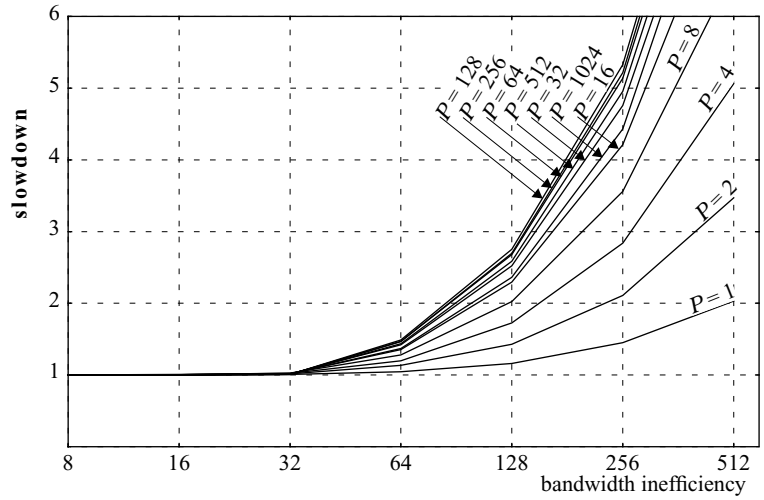


Figure 0-2: Slowdown of the odd-even mergesort when the bandwidth inefficiency (gap) increases. The numbers of processors varies, $N = 16384$.

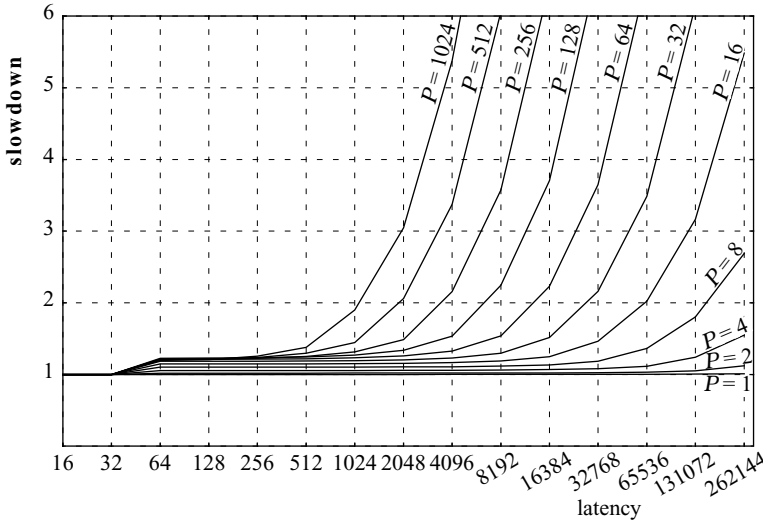


Figure 0-3: Slowdown of the odd-even mergesort when the latency increases. The numbers of processors varies, $N = 16384$.

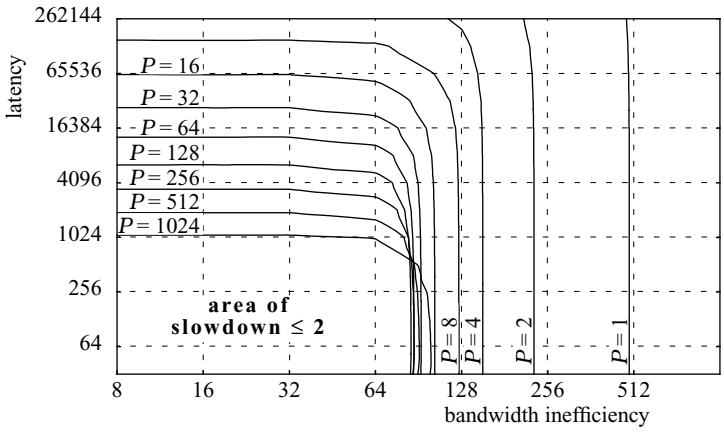


Figure 0-4: Acceptable (at most twice the time of the execution in an optimal machine with the same P) latencies and bandwidth inefficiencies in odd-even mergesort. The numbers of processors varies, $N = 16384$.

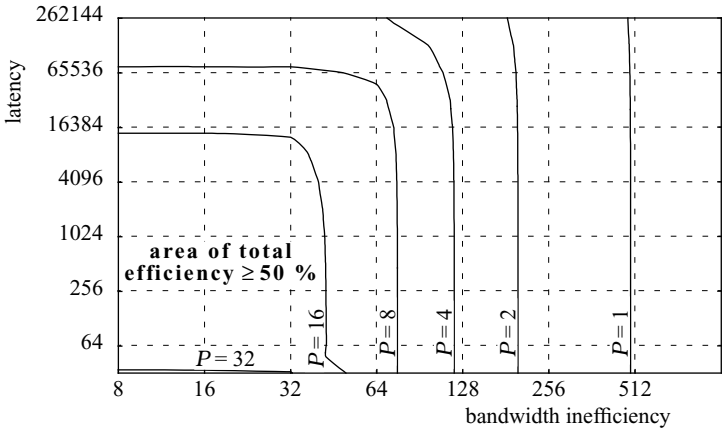


Figure 0-5: Limits on bandwidth inefficiency, latency, and number of processors for efficient (at most twice the work of the sequential execution in an optimal machine) execution of odd-even mergesort, $N = 16384$.

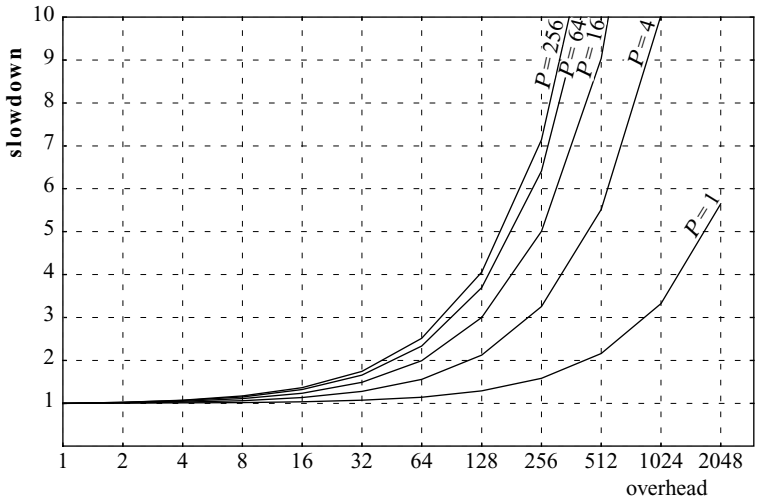


Figure 0-6: Slowdown of the odd-even mergesort when the overhead increases. The numbers of processors varies, $N = 16384$.

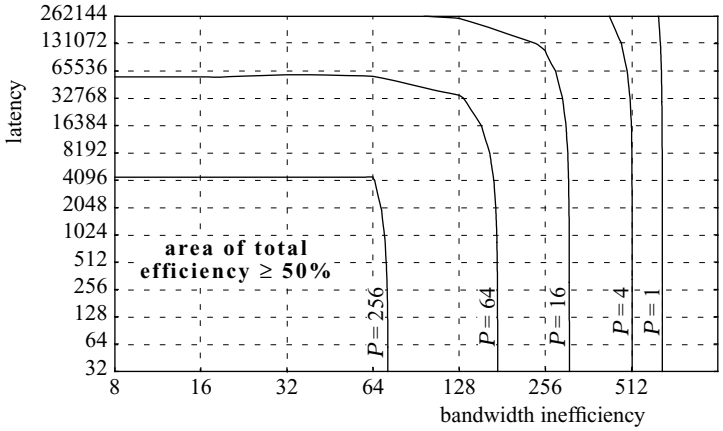


Figure 0-7: Limits on bandwidth inefficiency, latency, and number of processors for efficient (at most twice the work of the sequential execution in an optimal machine) execution of matrix multiplication, $N = 64$.

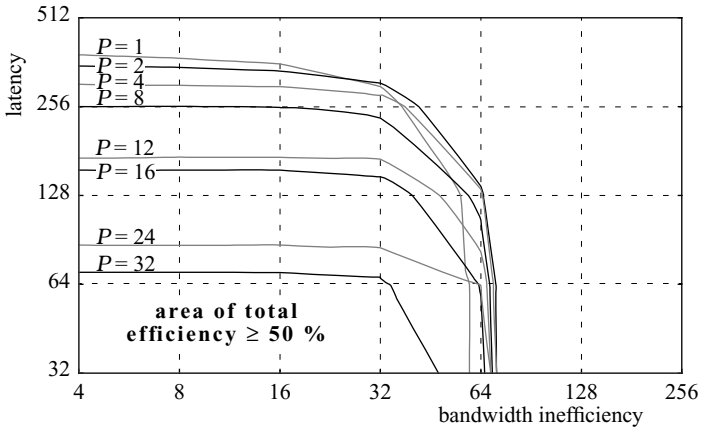


Figure 0-8: Limits on bandwidth inefficiency, latency, and number of processors for efficient (at most twice the work of the sequential execution in an optimal machine) execution of matrix inversion, $N = 64$.

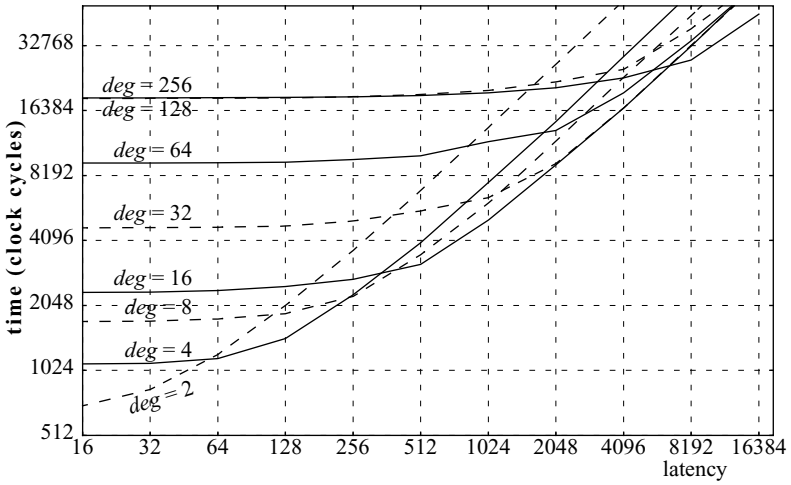


Figure 0-9: Time used for finding the maximum as function of latency for different degrees, $P = 256$, logarithmic scales.

5.4 Message passing models

- Until this far, processors have communicated via reading and writing shared memory.
- Communication has been abstracted as data parallel programming.

⇒ If there is no physical shared memory, and simulating it is not feasible, the processors have to communicate by other means.

- Processors (and processes) communicate directly by exchanging pairwise messages.
  - All input, task division, intermediate results, synchronization, final results, etc. are delivered in these messages.
- In addition to pairwise messages, there may be other messaging patterns.
  - These collective primitives are often data parallel -oriented.
- All processes work independently!

Why message passing?

- There might not be a (virtual) shared memory implementation available at all.

- If the communication pattern of messages of the algorithm follows the physical network structure, the communication is most efficient.
  - Communication pattern can be optimized for the physical network structure.
- In large scale machines, the machine structure is a very significant factor in performance. Also, we can afford more manual work on optimization on multi-million $/€ projects.

Applications

- Message passing techniques are used in several system levels from motherboard buses to email systems.
- We'll first consider senders and receivers in more abstract level.
  - Remember that the participants can be processors, processing nodes, full computer, subsystems of the those, processes, etc.
  - Different systems provide very different message passing mechanisms.

Message passing operations send and receive

- Process(or) sends/receives a message to/from another process(or).
- Basically just:

```
send (dest_pid, start_addr, length);
receive (source_pid, buffer_addr, in length);
```

- Additionally some options and status-results, like, e.g., in *printf*() & *scanf*().
- A lot of variations and options, though.
- Different systems provide different sets of options.

Message passing game

- Sit connected, check your neighbours.
- Use paper notes as messages.
- You may not talk, touch, or otherwise grab others' attention.
- Just show your message quietly towards your neighbour and wait until (s)he receives it.
- When you are planning to receive from someone, wait silent until the neighbour has something.
- Task: maximum-to-all.

Variants

- Message boxes (just leave your message to a agreed position on table).
- Interrupts (touch your neighbour with the message).
- Any-to-any communication.

Specified source (sender) at receiving or not?

- Sender obviously has to specify the destination by one way or another.
- Receiver, on the contrary, may wait for a message either from a specific sender, or accept a message from any possible sender, or from a restricted set of possible senders.

Identifying sender/receiver

- We can either specify an absolute process(or) identification number (PID), or something more abstract address assigned compile/load/run time.
- Can we generate new communication addresses during the execution?
  - These could be delivered as procedure parameters or with messages.
- We might just specify communication channels instead of destination/source process(or)s.

Communication topology

- Can any process(or) send a message to any other process (or) or not?
- In most communication networks, there is only a limited number of physical wires, not complete all-to-all network.
  - The degree of nodes in complete networks is linear, which is also problematic to implement and may slow down the nodes.
- In which topology the communication channels are arranged?
- Can topology be changed?
- Shall we still use global PIDs, or local directions, such as N, E, S, W.
- Is routing automated or do we have to do our own routing to reach non-adjacent nodes?

Buffering, suspension, polling

⇒ What happens if sender and receiver do not execute operations simultaneously?

- If sender is ahead, will it wait the receiver (blocking send), or will it fail, or will the message be buffered somewhere (non-blocking send)?
  - Will the buffering be made in sender's or receiver's buffer?
  - How the sender will know if/when the message was finally sent/received?
  - How about if buffers fill up, or there is congestion in the middle in the interconnection network?
    - Will sender then block, or is the message lost?
  - For long messages (compared to the speed of interconnection network), the changes of congestion are even larger.
- If the receiver is ahead, will it wait for the message to show up, or can it do something else meanwhile?
  - How the receiver will know about an arrived message?
  - How the receiver will know whether there will be a message coming later or not?
  - Can receiver set up a buffer for "when the message finally comes, please store it here".

Message passing time complexity

- Slightly more difficult to analyse/estimate than shared memory programming, see LogP model below.

Intermediate forms of communication

- Remote Memory Access: a request to read/write to the local memory of another processing node. (No synchronization!)
  - Distributed Shared Memory / Distributed Memory.
- Remote Procedure Call (RPC): request to invoke of subroutine in another processing node (parameters are delivered with request, possible return values on termination of the procedure).
- Remote Method Invocation (RMI): request to call a method of a remote object (OO version of RPC).

Collective communication

- Communication primitives involving several/all processors.
- Simplifies programming.
- Can be implemented efficiently (optimized) for a given platform.
- See MPI below.

Parallelism representation in message passing models

⇒ *for..pardo* suits better for dataparallel programming.

- In message passing computing, all processes are usually initiated at once (e.g., *mpi_init*()).
  - At the initialization, each process is assigned a PID and a processor where it is executed in.
  - All the processes execute usually the same program, but branch according to the PID.
  - The lifetime of processes is (nearly&often) the whole execution.
- In some systems, more processes can be created (by *fork*(), of some higher level primitive).
  - Dynamic invocation of new processes in not always possible, thus we'll assume static number of processes.

⇒ Parallelism is usually much more coarse than in PRAM algorithms.

- We try to make blocks of communication and blocks of computation as large as possible.

Synchronization

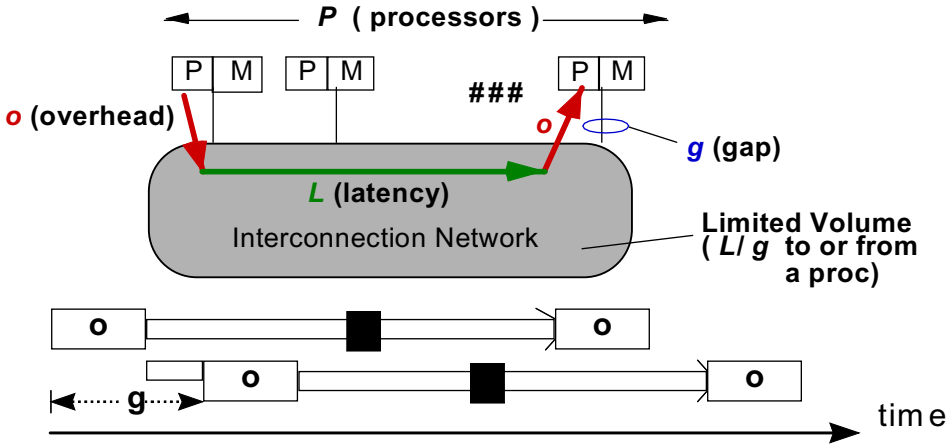
- Successful message passing itself is synchronization, thus no separate synchronization operation is needed.
- There are no race conditions / overlapping / etc. on shared memory usage.
- Collective communication implies group synchronization.
- Deadlock avoidance is most important and difficult problem in message passing programming.
  - But, it is usually easier than guaranteeing shared memory access.

Message passing cost model: LogP [Culler&al. 1993]

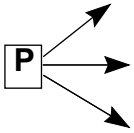
- Processing
  - microprocessors, DRAM, etc P
- Communication
  - significant latency (100's cycles) L
  - limited bandwidth (1-10% of memory bandwidth) g
  - significant overhead (10's - 100's of cycles) o
- Assumptions?
  - no consensus on topology (should not assume structure)
  - no consensus on programming model (should not enforce one)

LogGP [Alexandrov&al 1995]

- Additionally G, byte gap for long messages.

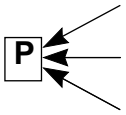


* Send *n* messages from proc to proc in time *2o + L + g(n-1)*
  - each processor does $o \times n$ cycles of overhead
  - has *(o-g)(n-1) + L* available compute cycles



* Send *n* messages from *one to many* in same time

* Send *n* messages from *many to one* in same time
  - all but *L/g* processors block so less available cycles



Other similar models

model	communic. method[b]	synchrony[c]	number of processors	bandwidth restriction[d]	latency measure	overhead measure	block transfers	other parameters
F-PRAM	SM	AB	P	B	L	B_P	B_B	B_V, B_M, M, S
BSP [104]	MP	BS	P	g	L			
LogP [28]	MP	AM	P	g	L	o		
LogGP [6]	MP	AM	P	g	L	o	G	
QSM [39]	SM	BS	P	g				
Y-PRAM [99]	SM	LS	P	B	L			
HPRAM [46]	SM	AB	P		L			S
LPRAM [3]	SM	LS	P		L			
Phase LPRAM [38]	SM	BS	P		L			S
Interval model[a] [71]	SM	BS	P	I	I	I		

BDM [58]	SM	BS	P				Bsz [e]	
PRAM [34]	SM	LS	P					

[b] SM = shared memory, MP = message passing, DM = distributed memory.
[c] LS = lock-step/common clock, BS = bulk-synchrony, AB = asynchronous, barrier synchronization at request, AM = asynchronous, synchronization via message passing.
[d] B = bandwidth inefficiency (soft limit on issue rate, eventual effect on completion of the communication), g = gap (hard limit on issuing rate)
[e] Bsz = block size of the communications

6 Message passing programming (with MPI)

- MPI – Message Passing Interface
- mpi-forum.org
- Standardization effort for portable message passing parallel programs.
- Originally Fortran, C, C++. Later other languages (Java, Python, R, Matlab, C#), MPI itself is language-independent.
  - Even interoperation between languages?
- Committee result (supercomputing industry, users, and academia).
- Replaced earlier proprietary and academic message passing libraries (PVM, P4, PARMACS, etc).
- Both open and proprietary implementations of MPI are available.
  - Supercomputer manufacturer can use special hardware/firmware to implement MPI calls highly optimized, especially collective communication.
- MPI 1.3 (MPI1) is widely used.
- MPI 2 is more complex.
- MPI 3 is a bit more complex.

Why to use MPI?

- To make portable programs for all scales of parallel computers.
- Main goal are (large) machines without shared memory, but MPI programs run also efficiently on shared memory (SMP).
- MPI programs can be also run on a cluster of machines, even on a geographically distributed, heterogeneous cluster.
- Message passing forces programmer to remember the cost of communication, and avoid fine-grained communication.
- Message passing paradigm makes programming less prone for non-deterministic errors than shared memory programs.
  - There are no hard-to-find accidental (programmer error!) shared memory race conditions.
  - Deadlocks are usually more deterministic, and easier to detect.
- MPI(1) does not support multithreaded shared memory programs within processing node, it is all separate processes.
  - You can, however, use OpenMP in each MPI process for 2-level hierarchical parallelism.
- Use OpenMP (or HPF) if you know for sure that the code will be always run on a shared memory machine.

MPI at Linux workstations (also other *nixes, MacOS)

- OpenMPI, https://www.open-mpi.org/
- MPICH, https://www.mpich.org/
- `mpicc program.c -o program`
- `mpirun -np 4 program`

MPI for Windows

- Microsoft MPI (inter-node connection initialized through Active Directory Domain Services)
- MPICH

MPI at server cluster (no more)

- 4 (was: 8) dual-socket Xeon 2.8 GHz & HT servers = 8 (was: 16) processors (16 (was: 32) virtual processors)
- 1Gb/s Ethernet

- `lamboot ~/lamhosts lamnodes mpicc program.c mpirun -np 8 a.out`
  lamhalt
- Instructions for compiling and running programs and how to transfer files to the cluster are displayed when logging in and more can be found via the LAM/MPI manual page (command: man lam, mpirun).
- Move files from other computers with scp/sftp.

MPI for your own PC (Linux, OSX, WinNT)

- OpenMPI (http://www.open-mpi.org/)
- MPICH2 (http://www.mcs.anl.gov/research/projects/mpich2/),

MPI at course server

- OpenMPI
- mpicc -o program.out program.c
- mpirun -np 8 program.out

Lecture material

- University of Notre Dame, Laboratory for Scientific Computing, MPI Tutorial
  - Six-function MPI, asynchronous communication, Collective communication, Communication groups, avoiding serialization, performance.
- Pavan Balaji (Argonne NL), Torsten Hoefler (ETH Zürich): Advanced Parallel Programming with MPI-1, MPI-2, and MPI-3

  - MPI-2 and MPI-3 new features: one-sided communication (remote memory access), hybrid parallel programming, asynchronous collective communication.
- Copies at course www-page with lecture recordings.

7 Other stuff

Other optional parts:

Parallel functional/dataflow programming

Processes, threads, and IPC - C&Java

7.1 Parallelizing sequential programs

⇒ Why we have to write parallel programs?

- Automatic parallelization would be economical and less error-prone!

⇒ Why sequential programs do not parallelize?

⇒ How sequential programs parallelize?

- In a compiled machine language program, it is almost impossible to find anything else than very fine grained parallelism.
  - Processors analyse consecutive (4-64) instructions, and try to execute as many instructions in parallel as there are functional units available.
    - Dependencies limit parallelism.
    - Branches make parallelization very difficult.
    - Out-of order execution can be used, if processor is very clever.
    - Processors collect statistics to help branch prediction.
    - In some architectures compilers can leave hints for processor.

⇒ Larger independent sections are impossible to detect in a compiled program.
⇒ Analysis has to be very fast (≈ ns) and error-free.

```
.L30:                    1
    movl %edi,%eax       2
    movl %esi,%ebx       3
    movb (%eax),%cl      4
    testb %cl,%cl        5
    je .L75              6
    movb (%esi),%dl      7
    jmp .L77             8
    .p2align 4,,7        9
.L33:                    10
    incl %eax            11
    incl %ebx            12
    movb (%eax),%cl      13
    testb %cl,%cl        14
    je .L75              15
    movb (%ebx),%dl      16
```

Parallelization of source code is much more possible

⇒ We could compile an old sequential program for parallel execution automatically (without changing source code).

- Result depends in the original programming style (and algorithm).
- For best results, we may have to clean up the program, possibly also help the compiler a bit.
- Compiler analyses sequential loops.

```
DO I=1,10000                    1
        C(I)=A(I)*B(I)          2
END DO                          3
```

- Statement at line 2 for different values if `I` are independent, thus they can be executed in parallel.
  - Each processors takes $10000/P$ iterations.
  - It is efficient only in shared memory multiprocessors with enough memory bandwidth.
- Nested iterations are often easier to parallelize, as it is enough to parallelize one of them.
- The more complex is the fragment of a program, the more difficult it is for the compiler to make sure that parallelization is safe.
- If each iteration depends on the previous one, direct parallelization will not do.

```
DO I=1,10000                    1
        A(I)=A(I-1)+B(I)        2
END DO                          3
```

- Don't use pointers, loops of unknown length, recursion, etc.

Languages

- Fortran IV .. Fortran 77 (Fortran 90, C).
  - Simple language, numerical computation.
- Compilers complex and expensive.
- There are some (experimental) interactive tools that allow programmer to participate in parallelization.

Machines

- Vector supercomputers
  - Using a vector processor needs parallelization analysis anyway.
  - Even sequential operations are executed deeply pipelined (32-64 consecutive operations in different stages of execution simultaneously).
  - Huge memory bandwidth.
- In some cases also newer SMP-machines.
- Special compiler needed, memory bandwidth limits.

7.2 Fortran 90 / 95 / HPF

- ANSI-standard.
- The next choice (after Fortran 77) for new numerical programs.
- Good compilers (only recently matching F77, though).
- Note: Fortran 2003/8 are way more bloated.

⇒ In addition to scalar operations, it supports operations for vectors and matrices.

```
PROGRAM TEST
INTEGER A(10), B(10), C(10, 10)
A = 1
B = (/ (I,I=1,10) /)
C = RESHAPE( SOURCE = (/ (I,I=1,100) /), SHAPE = (/ 10,10 /) )
C = MATMUL(C, C)
END PROGRAM TEST
```

A set of ready vector and matrix operations are available as library calls.

- Programming much more easier and less error-prone.
- Parallelism is natural.
- Compiler can implement the operations in parallel.
- Library routines can be optimized for each platform.
- Portability of programs still good.

Old cs:/opt/SUNWspro/bin/f90

gfortran for Linux (and others).

Lecture material:

http://web.am.qub.ac.uk/users/d.dundas/msci_workshop/fortran/notes/fortran90.pdf

High Performance Fortran (HPF)

- High Performance Fortran Forum 1993
- An "extension" to Fortran 90
- Most of the features are "`!HPF$`" directives, which an ordinary compiler skips as comments.
  - Only new statement `FORALL`, some additional library routines.
    - `FORALL` is now also in f95
  - Mostly advice for the compiler.
- http://hpff.rice.edu/
- In SMP machines, OpenMP has mostly replaced HPF because of simplicity.
- The remaining benefit of HPF is the data location directives for larger machines.

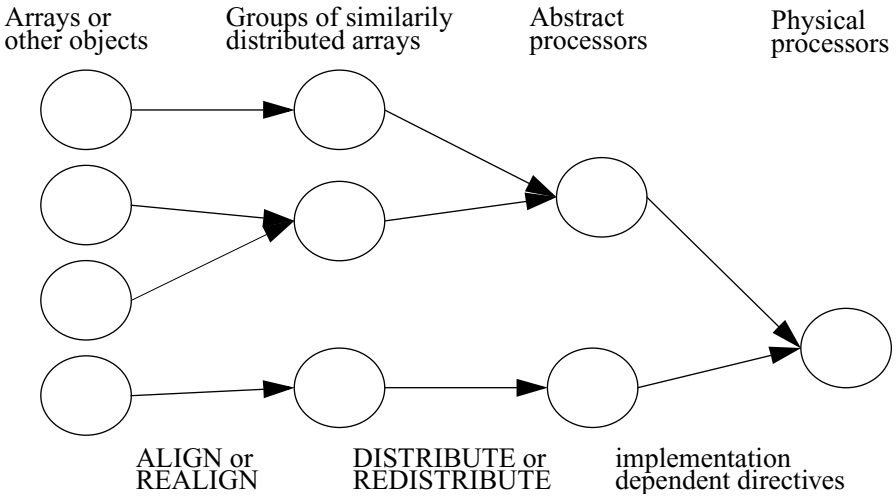
⇒ Programmer guides parallelization.

```
!HPF$ INDEPENDENT
      DO i=1,100
            A(P(i)) = B(i)
      END DO
```

- Correct permutation if `P` contains each integer 1..100 exactly once.
- Programmer can know this, compiler cannot (without programmer to tell so).
- `FORALL` -statement (now also in f95/f03)

```
FORALL (i=2:n:2, k=2:m:2)
      WHERE (x(i,j) .NE. 0.0)
            x(i,j) = 1.0 / x(i,j)
      ELSEWHERE
            x(i,j) = INF
      END WHERE
END FORALL
```

- Prefix and rank operations as library functions.
- Programmer can guide distributing data and computation to processing nodes (and memory).




```
!HPF$ TEMPLATE K(100,100,100)
!HPF$ DISTRIBUTE K(BLOCK, CYCLIC, *)
                    REAL :: A(1000)
!HPF$ DISTRIBUTE A(BLOCK(10))
                    REAL :: B(1000)
!HPF$ DISTRIBUTE B(CYCLIC)
                    REAL :: C(100)
!HPF$ ALIGN C(:) WITH K(*,:,*)
                    REAL :: D(100)
!HPF$ ALIGN D(I) WITH A(N - I+1)
```

7.3 BLAS, PBLAS, LAPACK, ScaLAPACK

- A set of Fortran routines (for numerical computation).
  - Also for C, C++, Java, etc.
- Widely implemented and highly optimized for different platforms.
- (Parallel) Basic Linear Algebra Set.
  - Vector operations (Level 1)
  - Matrix-vector operations (Level 2)
  - Matrix-matrix operations (Level 3)
- (Scalable) LAPACK uses BLAS to provide more complex operations.

⇒ The fasted linear algebra!

- http://www.netlib.org/blas/
- http://www.gnu.org/software/gsl/manual/html_node/BLAS-Support.html
- http://www.netlib.org/scalapack/html/pblas_qref.html
- http://www.netlib.org/lapack/
- http://www.netlib.org/scalapack/

Use libraries!

- 1000 × 1000 floating point matrix multiplication, GFLOPS.
- Intel Core i5-4590 CPU @ 3.30GHz / NVIDIA GeForce GTX 750 Ti
- Gnu C/Fortran Compiler (gcc), Portland Group C/Fortran Compiler (pgcc), Nvidia nvcc, all -O3

Compiler	Single precision	Double precision
School algorithm, gcc	1.5	1.4
With transpose, gcc	2.5	2.4
OpenMP 4-core, gcc	9.3	9.0
With transpose, pgcc	10.1	5.3
OpenMP 4-core, pgcc	38.6	14.5
OpenBLAS (1-core)	69.8	42.5
PG BLAS (1-core)	≈150.0	≈80.0
School algorithm, CUDA	168.1	32.3
CUDABLAS	1210.3	41.5

7.4 Other

Ada

- US DoD -78, ANSI -83
- Goal was robust (bug-free) programming.
- Concurrency is represented as threads.
- Threads communicate trough synchronized rendezvous (kohtaaminen).
  - Ada 9X
  - process replication (data parallel programming)
  - shared memory usage
  - object-oriented support

Data flow programming (tietovirtaohjelmointi)

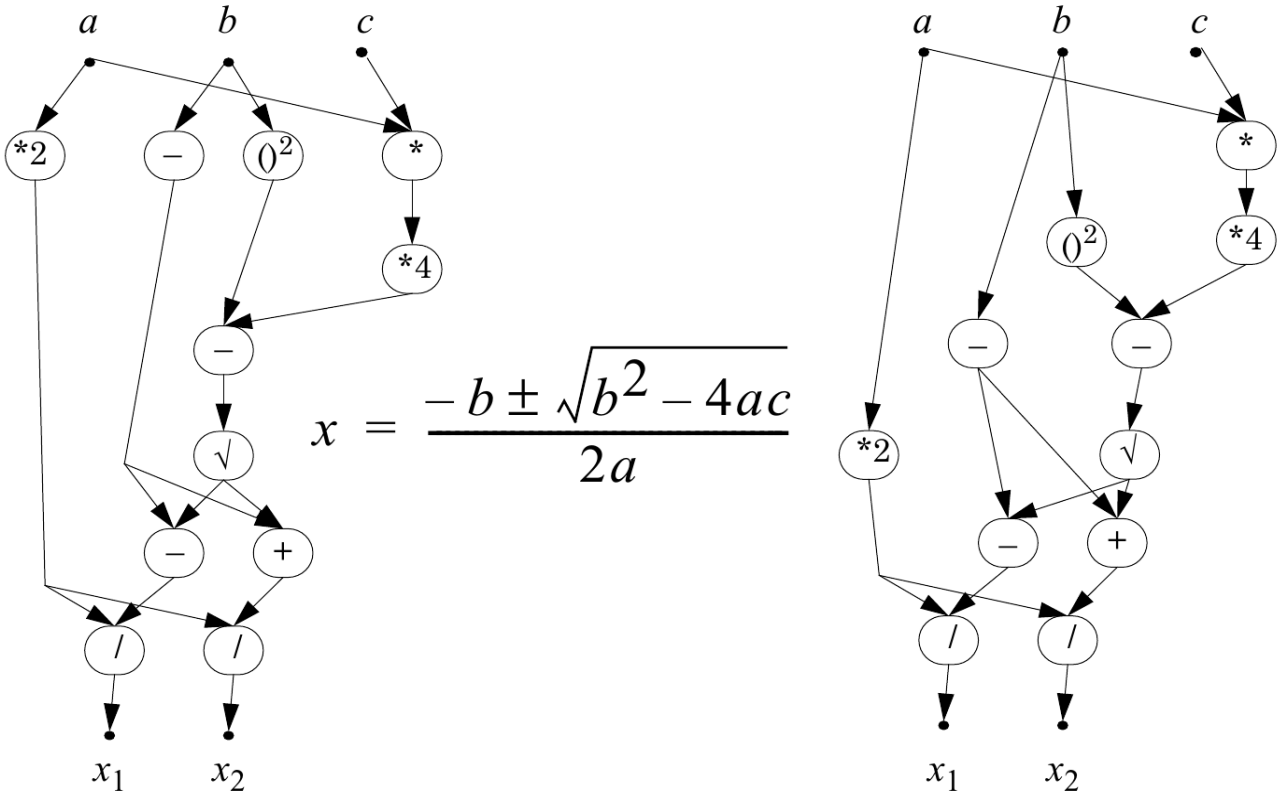
⇒ Computation is represented as a directed (acyclic) graph.

Computation as a graph

- Input as root nodes.
- Output as sink nodes.
- Operations as intermediate nodes.
- Data flow as edges.

Execution

- Each operation is executed at earliest when all operands have arrived from in-edges.



⇒ Computation can be scheduled according to various strategies:

- Greedy: as soon as possible
- Lazy: only if/when someone would need the result.
- Very lazy: only if/when someone would need the result acutely (back-pressure).
- Optimized: schedule each operation so that at most given number of operations are executed simultaneously. Still trying to keep total execution time.
  - Optimize for minimum work ($T \times P$): sequential algorithm.
  - Optimize for minimum time: greedy
  - Optimize 1st for minimum time, 2nd for minimum P.
  - Optimize for mimumum P in given time (e.g., $C \times T_S$).

Measures

- Execution time = length of the longest path (height of the data flow diagram).
- Needed number of processors = maximum parallel operations (width of the data flow diagram).

⇒ The most accurate model of parallelism.

- Used to be popular research subject (in Japan) at late 80's.
- Some machines were built.
  - Implementation efficiency was improved by I-structures, global static tagged memory.

Programming

- Either functional programming, or
- directly using data flow model, stepwise refining (top-down, or bottom-up).

Systolic arrays

- Programmable logics are configured to implement a given operation (e.g., FFT) efficiently.

- Data is pumped in at one end, propagated through the chip, computed on the fly, result output at other end.
- Culler 03-13

Sorting networks

- A network of operations that sort the input.

Parallel functional programming

- Functional programs are parallel by nature

  (* (+ (* 1 2) (* 3 4) (* 5 6)) (+ (* 7 8) (* 9 4) (* 5 4)))
- There are several languages (Sisal, ID, Multilisp, Haskell, NESL, etc.)
- Relatively easy to compile to shared memory computers.
- Message passing would be also very natural, but communication delays destroy efficiency in fine grained programs.
- Glasgow Parallel Haskell: `http://www.macs.hw.ac.uk/~dsg/gph/`
- At their best for data flow computers and systolic arrays.

Parallel logic programming

- A logic program consists of a set of Horn clauses

  $H \leftarrow B_1, B_2, \ldots, B_n$

  $H \leftarrow C_1, C_2, \ldots, C_n$
- H is true if all B_i are true or if all C_i are true.
- In a simple case, everything could be evaluated in parallel!
- All B_i in parallel: AND parallelism.
- B's and C's in parallel: OR parallelism.
- In reality, within a clause, there will be binding of variables, thus the processes have to communicate through the variables (channels).

Linda tuple space

- Carriero&Gelernter
- Shared (associative) memory
- Functional parallelism presentation

Tuples

- A list of elements of any (primitive) type
- Matching tuple has same number of elements of same type each

Just four operations

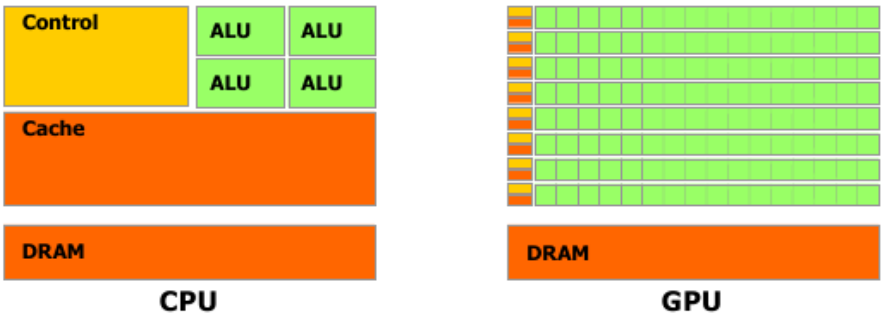
- *out*(tuple): placec a tuple to tuple space
- *rd*(tuple template): read a tuple from tuple space
- *in*(tuple template): read&remove a tuple from tuple space
- *eval*(tuple to be evaluated): create new thread of execution, result will be a tuple

Reading

- *in* and *rd* tuples may contain fields with variables tagged with a '?', those variables will be assigned the values from a matching tuple.

7.5 GPGPU computing

- General Purpose Graphical co-Processor Unit computing.
- Graphical co-Processors have up to thousands of "processors" to accelerate 2/3D image generations. Also, good/excellent memory bandwidth.
- see "Why parallel (once again) [Gordon Moore, ISSCC2003, www.intel.com]" page 15.
- What to do with 21.1 billion transistors in a chip? https://www.nvidia.com/en-us/titan/titan-v/ https://www.intel.com/content/www/us/en/products/processors/core/x-series/i9-7980xe.html



[NVIDIA]

  - Notice: Control and Caches are actually mixed with ALUs.
- SIMD/SIMT vs. MIMD
- Hierarchical (embedded) memory.

- GPGPU: Using GPU to compute something not related to displaying graphics in screen.
- Traditionally/Originally:
  - Transform computation to graphics (texture) problem, solve that in GPU (OpenGL/DirectX primitives), untransform back to the solution.
  - Difficult, restrictive, requires (deep) knowledge of graphics algorithms.
- Now (OpenCL, Nvidia CUDA, (ATI CTM/Stream, Apple VImage/CoreImage):
  - (Almost) plain C to execute any code in GPU processors.
  - Still you need to transfer data between host and device memories.
    - Some libraries can automate data movement.
  - Memory and processor organization in GPUs more complex than in CPU:s. Achieving full performance requires tuning.
  - Special libraries that execute in GPU (Matlab, BLAS).
- In future?
  - (Virtual) unified memory (already in CUDA 6.5 / OpenCL 2.0 (automatic memory transfers))
  - Compiler directives to parallelize loops (as in OpenMP)
    - OpenACC proposed, evolving.
  - These higher level programming tools hide true architecture, and thus may lead to inefficient usage of resources.
- Readily implemented and optimized libraries for applications (and linear algebra).
- Shows the potential of massively parallel architectures.
  - In some cases up to 100× performance compared to same size CPU.
  - Even if GPUs are optimized for 3D graphics!
    - Newer GPUs are compromises between 3D graphics and GPC.

- OpenCL
  - http://www.khronos.org/opencl/
  - http://www.khronos.org/assets/uploads/developers/library/overview/opencl-overview.pdf

  - http://www.khronos.org/developers/library/overview/opencl_overview.pdf
  - https://handsonopencl.github.io/
  - Originally by Apple, now Khronos group (same as OpenGL).
  - Nvidia, AMD/Ati, Intel, Apple, IBM, Nokia, Samsung, Motorola, SE, ARM, TI, SUN...
  - Specification 1.0 at Dec 9th 2008. Products at 2009-10.
  - Specification 2.0 at Nov 2013.
  - Similar structure as CUDA, but more general (dynamic resource allocation, runtime compilations, etc.) and device independent.
  - More complex syntax than CUDA.
  - Also embedded profile (for mobile devices)
- DirectCompute (DirectX 11 Compute) by Microsoft, 2009.
- CUDA, NVIDIA only, still more mature than OpenCL
  - GeForce 8000 onwards, Tesla, Quadro.
  - https://developer.nvidia.com/cuda-zone
  - https://developer.nvidia.com/cuda-education-training
- CUDA vs OpenCL terminology:

CUDA	OpenCL
host&device	host&device
multiprocessor	compute unit
core	processing element
kernel	program
thread	work-item
thread block	work-group
grid	computation
warp	wavefront
global memory	global memory
shared memory	local memory
local memory	private memory
-	context
gridDim, blockDim	get_num_groups(), get_local_size()
threadIdx, blockIdx	get_local_id(), get_group_id()

- OpenACC
  - OpenMP -like directives to offload computation from CPU to GPU.
  - www.openacc.org
    - Originally Cray, Portland Group, NVIDIA, later also AMD
    - gcc-7
    - PGI compiler community edition is free.
  - As offloading to GPU requires data transfer, separately parallelizing each loop results transferring data back-and-forth several times.
    - Loops must be grouped and data handling explicitly defined.

Libraries on top of CUDA

- https://developer.nvidia.com/gpu-accelerated-libraries
- CUBLAS
  - BLAS calls instead of kernel call, still need to handle memory allocation and transfer.
- Thrust
  - Simplifies memory allocation and memory transfer.
  - Lot of elementary generic vector/matrix operations.
- OpenCV
  - Computer vision, image processing and machine learning.
- ...

NVIDIA GeForce RTX 2080 Ti

-
- 18 billion transistors, 250 W, 1 200€.
- 4 352 cores @1.5 GHz
- 68 multiprocessors, each 64 cores, partitioned into 4 blocks of 16 cores each (see below).
- 11 GB memory, 352 bit interface, 616 GB/s bandwidth
- X MB L2 cache.
- PCIe 3.0 x16 connection to host, 15 GB/s
- 13 TFLOPS (32-bit float)

NVIDIA GeForce RTX 2060 SUPER

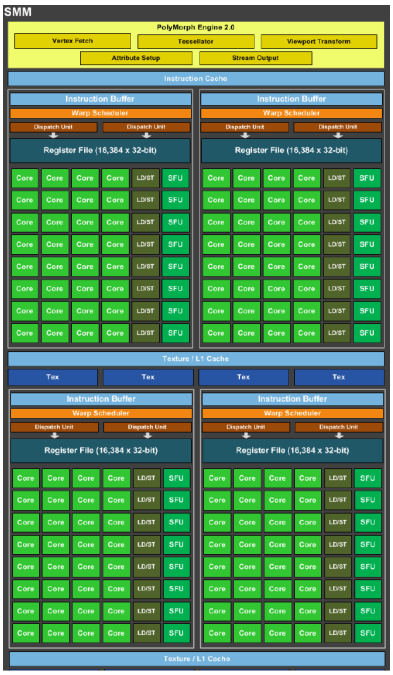
- Installed at laskuri3.uef.fi.
- TU106 Turing, 10 billion transistors
- 2 176 cores @1.5 GHz
- 34 multiprocessors, each 64 cores, partitioned into 4 blocks of 16 cores each (see below).
- 8 GB memory, 256 bit interface, 448 GB/s bandwidth (380 GB/s measured).
- 4 MB L2 cache.
- PCIe X.0 xXX connection to host, 7.5 GB/s (6.7 GB/s measured) (2.0 or x8 because of host??)
- 6 TFLOPS (4 TFLOPS measured) (32-bit float)
- 190 GFLOPS (64-bit float) ??
- ≈450 €
- 175 W.

How to use

- Copy an example from `/usr/local/cuda/samples/` to your own directory.
- Copy or link also /usr/local/cuda/samples/common:
- Edit .c/.cpp/.cu. Edit Makefile for common/inc and possibly file names.
-
- make
- Or: manually: `/usr/local/cuda/bin/nvcc program.cu`
  - If you need to pass parameters for C-compiler use, e.g.:
    `/usr/local/cuda/bin/nvcc -Xcompiler -O3 -Xcompiler -fopenmp program.cu`

Multiprocessor internal structure

- Each multiprocessor shares 64kB shared memory and 24kB L1 cache
- Each block shares instruction buffer, scheduler, &etc., thus executes in SIMD.
  - Except that there are 1-2-4 control units in each multiprocessor.
  - Each control unit can executes 32 threads (a "warp") at once (same instruction).
- Registers (16-64k/MP) are dynamically allocated to threads.
- 8-32 32-bit FP units/MP
- 4-16 64-bit FP units/MP



7.6 Hadoop/MapReduce

- Data-flow computing for scalable data-processing jobs (on large cluster of computers). (p. 12)
- Originally Google internal indexing tool, later several (open-source) implementations.
- https://hadoop.apache.org/
- MapReduce: programming model
  - In intermediate steps, data is stored to files in <key, value> pairs.
  - Computation proceeds in steps, in each step:
    - each piece of input is mapped (the actual computation) to another value(s) on each node (intermediate results)
    - data exchange (shuffle, sort): intermediate results are exchanged between nodes on bases of keys of the output
    - intermediate results are combined and reduced to new output data
  - Data flow between steps is a directed acyclic graph (DAG) (of steps).
  - Processing steps can be done using Java, C++, Python, etc.
- Apache Hadoop: Distributed file system and resource negotiator (tools to run and monitor MapReduce jobs).
- Strengths are only apparent when using huge data sets and a large cluster (or cloud) of machines.
- Resource negotiator handles data distribution, optimizes shuffling and is able to reschedule failed jobs in case of hardware failure).

7.7 Concurrent and parallel, processes and threads

- Concurrent = several threads of execution (apparently) simultaneously, exact order of non-synchronized instructions not known/may be different from execution to execution.
- Parallel = we use several physical processors to do something faster.

⇒ System that is designed to be concurrent can be executed in parallel if we have several processors.

⇒ Parallel is concurrent by definition.

When to use stuff in this part.

- If your goal is to make a new parallel program, use MPI, or OpenMP if possible.
- If you have to parallelize an existing program and someone forbids(?) you to use MPI or OpenMP, then go ahead.
- If you need to just make something concurrent, then use processes or/and threads.
- If you need to do communication between computers, the use IP.
- If you really need shared memory for performance, be careful.

Process (prosessi)

- The base unit of execution in (nearly) all operating systems.
  - Processes execute concurrently and independently.
  - processes are executed in multitasking manner in most OS.
  - In a SMP computer, the processes can be executed in parallel.
- Each process has its own memory, registers, environment, etc.
- New processes can be created by (unix, Posix-compatible)

```
pid_t fork (void);
```

  - which creates an identical copy of the calling process, and returns the process-id of the child process (`-1` on error).
  - The execution of the identical child process starts at returning of `fork()`, the only difference is that the return value of `fork()` is `0`.

- Most OS implement the copying in copy-on-modify semantics.
- Very little is actually copied on `fork()`. Even if each process will get its own virtual memory space, the processes will use the same memory as long as possible .
- Only if either process modifies something, it (page) will be copied.
- Processes can communicate with each other, thus they can be used to create parallel programs.
- Shared memory segments, semaphores, pipes/messages.

Threads (säikeet)

- Each process can contain one or more threads.
- Each thread executes concurrently (independently) as processes, but the threads of the same process share all the memory (variables, heap, program, file descriptors, etc.).
- Usage: e.g., one thread handles the user input, one the updating of screen, one computes on background, one transfers data to/from network, etc.
- Most OS provide threads, JavaVM provides threads, there are separate platform independent thread packages, etc.
- C11 & C++11 have standardized `threads`-library.
- Synchronization: mutexes.
- Even if OS (kernel-level) threads may execute in parallel in SMP computers, it is not guaranteed. Use processes instead if you are want to ensure parallel execution.
- http://users.actcom.co.il/~choo/lupg/tutorials/multi-thread/multi-thread.html

Thread API

POSIX threads (phtread) / C11 threads

- Almost compatible
- Parameters for new threads are delivered as `void*`, return value a `void*` (pthread) or `int` (C11).
- Several parameters can/should be collected to a `struct`.

```c
#include <pthread.h>
pthread_create(pthread_t *thrd,
    attr, func, void *prms);
pthread_join(thrd, &ret);

pthread_mutex_init(
    pthread_mutex_t *lock, attr);
pthread_mutex_lock(lock);
pthread_mutex_unlock(lock);
pthread_mutex_destroy(lock);
```

```c
#include <threads.h>
thrd_create(thrd_t *thrd,
    func, void *prms);
thrd_join(thrd, &ret);

mtx_init(mtx_t *lock, attr);

mtx_lock(lock);
mtx_unlock(lock);
mtx_destroy(lock);
```

Mutex (mutual exclusion)

- As threads use shared memory to co-operate, they need to guard the access to shared resources.
- Mutexes provide a tool for this.
- Programmer is always responsible for using the mutexes right.
- only one thread can `lock` a mutex at a time.
  - other processes trying to `lock` will block until mutex is free.
  - `trylock` can be used to test a mutex (remember to check return value!).
  - `unlock` will free the mutex and yield access to another thread waiting in a `lock`.
  - Recursive mutexes can be locked again by the same thread – they need to be unlocked as many times before other threads can access them.
  - Timed mutexes time-out on `lock` (remember to check return value!).

7.8 How processes communicate?

- Internet Protocol (IP)
  - TCP/IP (reliable stream), UDP/IP (unreliable (faster) datagrams)
  - Provides a communication channel between any processes in any computer (in the whole network).
  - Can be used also within localhost
  - Programmer abstraction: sockets
- Pipes
  - Byte stream between processes at the same computer (OS)
  - Use like a file
  - We have to make protocol ourself
- Semaphores (mutexes)
  - Provide atomic test/wait&modify operations
  - Used for interprocess synchronization
  - Ensuring exclusive access to (modify) resources
- Shared memory segments
  - Allocate a memory block to be accessed by several processes.
  - The shared block is mapped (attached) to virtual memory spaces of several processes.
  - Efficient, useful for SMP computers, requires careful synchronization.
- Remote Procedure Call / Method Invocation (RPC/RMI)
  - Calling a remote (registered) procedure/OO interface.
  - Goal is to make distribution transparent (call remote procesure/method like a local one).

Interprocess communication within a host

⇒ TCP/IP is nice because it portable, and works also between hosts, but it has some additional overhead (because of the above).

- The following methods are less portable, but more efficient.
- Howto (e.g.): Unix Multi-Process Programming and Inter-Process Communications:
  - http://users.actcom.co.il/~choo/lupg/tutorials/multi-process/multi-process.html (link down)
  - http://beej.us/guide/bgipc/output/html/multipage/index.html
  - http://www.haifux.org/guy_keren.html

Pipes (SysV, Posix, WinNT, and others)

- One-directional FIFO (queue) of bytes.
- Create by

```
int mypipe[2];
pipe(mypipe); // check return value too...
```

- Now `mypipe[0]` is a file descriptor for reading from the pipe, `mypipe[1]` is file descriptor for writing to the pipe.

⇒ How to use pipes for IPC?

- Create the pipe before `fork()`, then mypipe is copied to both processes.
  - The writing process must close `mypipe[0]` and write to `mypipe[1]`.
  - The reading process must close `mypipe[1]` and read `mypipe[0]`.

Features

- OS bufferes the communication (up to some limit).
- `read()` is blocking, `write()` blocks only if buffers are full.
- To check readability/writeability, you can use select().
- For two-way communication between processes you can use two pipes, although it is not elegant anymore...

Named pipes

- A persistent pipe visible in the file system.
- Usable (nearly) like any file.
- Make a new one by

```
int mkfifo ( const char *pathname, mode_t mode );
```

- or

```
mkfifo [OPTION] NAME...
```

- One (any) process should open it for reading, one (any) process should open it for writing.
- Works only after both `open()`:s are done.

Shared memory (POSIX)

- System-wide persistent (until reboot) shared memory segments can be allocated using `shmget()`
  - Allocate in bytes, Key = identifier, flags as in files (once is enough).
  - Attach a local pointer to the allocated segment using `shmat()`.
- Use through the local pointer.
- Detach from segment using `shmdt()` (all processes).
- Free the segment using `shmctl()` (only once).
- Guy Keren: Unix Multi-Process Programming and Inter-Process Communications (IPC): http://users.actcom.co.il/~choo/lupg/tutorials/multi-process/multi-process.html

Semaphores (POSIX)

Semaphores are the access guard tool for processes as mutexes are for threads.

- System-wide persistent (until reboot) semaphores can be allocated using `semget()`
  - Actually a semaphore set (array).
- Semaphores can be initializes using `semctl()`.
  - Commands `SETVAL`, `GETVAL`, `SETALL`, `GETALL`,`IPC_RMID`,`IPC_STAT`.
- Semaphores can be manipulated using `semop()`.
  - Atomic operations: Set, Get, or TestAndSet.
  - `semop(semid, semop, n)`, `semop` is a struct that defines which semaphore and what operation.
  - $-k$ reduce value by k when semaphore is greater than k (i.e., wait-and-lock).
  - $+k$ increase value by k (i.e., unlock).
  - 0 wait until value is 0.
  - Using flags you can get error instead of blocking.

Shared memory segments and semaphore sets are shared within the whole system!

- They are not automatically freed.
- Any process can attach to them (given permissions) if it has the key!
- There is a limit (usually modest) how many segments/semaphore sets there is in a system. Leak will consume this.
- List them with `ipcs`. Remove with `ipcrm`. (InterProcessCommStatus/Remove).

Threads

- Guy Keren: Multi-Threaded Programming With POSIX Threads:
  http://users.actcom.co.il/~choo/lupg/tutorials/multi-thread/multi-thread.html

7.9 Java threads

- A thread is a sequence of execution in a process (program).
- In a multithreaded process (program) there are several concurrent threads of execution (program counters).
- Each thread has it's own stack, and thus local variables (of those program segments that it has started itself).

Programming a thread

- A thread is implemented as a class with method *run*(*void*).
  - *run*() is the life-cycle of the thread (like *main*() in a program).
  - Possible parameters must be given beforehand to the constructor.
- Class either extends class *Thread*, or implements *interface Runnable*.
- Extending *Thread* is a bit easier, if it is possible.
  - If our class already extends (inherits) some other class, extending *Thread* is not possible, as Java does not allow multiple inheritance.
    - Thus, you need to implement *Runnable*.

Extending Thread

- The functionality is built to method *run*() (and to whatever it calls).
- Start the thread in the constructor (*start*() method inherited from *Thread*).
  - *start*() can be called at constructor, or separately by the creating code.

```
public class MyThread extends Thread {
    public MyThread() {
        start();
    }
    public void run() {
            // thread main starts
            // here is the life cycle of the thread ...
            // end of thread
    }
}
```

Implementing Runnable

- The functionality is built to method *run*() (and to whatever it calls).
- In the constructor, create a *Thread*, and give it *this* (*Runnable*) to run.
  - Start the *Thread* with method *start*().

```
public class OwnThread2 implements Runnable {
    public OwnThread2() {
        Thread s = new Thread(this);
        s.start();
    }
    public void run() {
        // thread main starts
        // here is the life cycle of the thread ...
        // end of thread
    }
}
```

Life cycle of a thread

- A new thread is started with method *start*() - it then starts executing method *run*().
- Thread executes concurrently with other treads (time-sharing, or parallel).
- Thread stops temporarily (not runnable) if
  - it sleeps (*sleep*())
  - it waits for an event (*wait*())
  - it waits for a input/output stream (*read*(), etc.)
- Thread terminates when it reaches the end of *run*() method (see below).
- You can test the state of a thread using method *Thread.getState*().
- The object that implemented the thread will remain available (to other threads) even after termination.
  - Methods can be called from other threads.
  - Public fields can be used.
- Naturally, the object (methods, public fields) were available also during the execution of the tread.

Thread execution

- When *run*() method is started, a new execution thread begins (a new program counter is initiated) (like in *fork*()).
- Execution of the thread can visit any other part of program (methods of other classes, methods of other instances of the same class, etc.).
- In the same object, even in the same method of the same object may be several concurrently executing threads.
  - Own copies of local variables.
  - Fields of the object are shared.

⇒ We have to separate classes/objects/methods and their execution!

Which thread is executing when?

⇒ Usually only one thread is executing at a time.

- Each thread has a priority (1-10), the runnable thread with the highest priority executes.
- In most platforms, the runnable threads with the same priority execute concurrently.
- By using method *yield*(), a thread can (should) give execution turn to another runnable thread.
  - If a thread is selfish, and does not block for waiting something, it will choke other threads in a non-timesharing system.

Stopping treads

- A tread terminates when the *run*() method exits.

⇒ The thread must do it itself!

- If other threads need to be able to tell the thread that "it's time to quit", they can tell it using, e.g., a field, which the tread monitors regularly. (Still, the thread must do it itself!)
- We can use also a separate flag class.
- See also interrupts below.
- We can define a thread to be a background thread (demon), which will automatically quit whenever all other threads have terminated.
  - *setDaemon*(*true*) before starting the thread.
- In previous versions of Java there was an unsafe *Thread.stop*() -method, but it is now obsolete.

Interrupts

- A *sleep*():ing, *wait*():ng, or queue/channel -waiting thread can be woken with method *interrupt*(), in which case the thread will get an exception.

```
try {                                                          1
    sleep(1000);                                               2
} catch (InterruptedException ie) {                            3
        // waken in middle of sleep ...                        4
}                                                              5
try {                                                          6
    x = blockingQueue.take(); // waiting for element           7
} catch (InterruptedException ie) {                            8
        // waken in waiting for data ...                       9
}                                                              10
while(true) ...                                                11
        // cannot be stopped with in interrupt                 12
```

- A thread that is in execution is not interrupted, but the interrupt -status is changed. Then the thread can check it itself using *isInterrupted*().

Synchronizing threads

⇒ Usually threads share resources.

⇒ Usually threads have to cooperate in certain order.

Every Object in Java has a lock

- Thus, every thread has a lock.
- Thus, every data structure has a lock.
- The locks can be used for synchronization in many ways.
  - The best way depends on application.
- The easiest way is the include modifier *synchronized* in the definition of method(s).
  - There can be several synchronized methods in an object (class). If one of them is in execution (locked), none of them can be started (except by the thread that started the first execution (reentrant synchronization)).
  - Whenever the thread exits the synchonized method, a waiting thread can enter.

```
public class increaseDecrease {
      private int data;
   public synchronized int increase() {
      return ++data;
   }
   public synchronized int decrease() {
      return --data;
   }
}
```

- At most one of the methods is executing at any given time.

Synchronizing a block (with respect to a lock of an object)

```
synchronized( obj ) {
   // only one thread can execute these at a time
}
```

- Method synchronization is actually just a simpler version of the actual synchronization of a block (with respect of an object lock).

```
void synchronizedMethod() {
   synchronized(this) {
      // actual body
   }
}
```

- Block synchronization is more flexible and makes possible more fine grained synchronization.

Lock Objects

- *synchronized* -locking is not quite flexible if a single lock needed is for duration of several methods, in different objects, or otherwise not bound to source code blocks.
- By using *Lock* -interface (*ReentrantLock*), we can lock critical sections even more flexibly.

Atomic access

- Some classes in *java.util.concurrent* packages provide atomic methods.
- Atomic operation ensure that it is executed at once with no other thread disturbing.
- By using atomic classes, we can reduce synchronizations:

```
public class increaseDecrease2 {
    private AtomicInteger data = new AtomicInteger(0);
    public synchronized int increase() {
        return data.incrementAndGet();
    }
    public synchronized int decrease() {
        return data.decrementAndGet();
    }
}
```

- See also, e.g., *java.util.concurrent.ConcurrentHashMap<K,V>*

Inter-thread notifications

- *Object.wait*() method can be used to yield a lock and wait until another thread notifies it with *notify*() method (or timer expires, or thread is interrupted).
- When one of these happens, the thread waits until it regains the lock, and continues execution.
- A thread can wait for another thread to terminate using *join*() method of the other thread.

Repeatedly timed tasks

- By using instances of *Timer*-class, we can set instances of *TimerClass* (subclass) to be executed (*run*() method) after given time, repeatedly, if needed.
  - If repeating, the object is persistent, *run*() is executed several times.
- Actually, there is only one thread (coordinated by *Timer*), thus the *run*() method of the *TimerTask* subclass should be short.
  - If it won't terminate, next executions won't execute.

Data structures for inter-thread communication

- *java.util.concurrent*
- Every collection (and other classes) of Java API is documented as synchronized or not synchronized (Thread-safe, not Thread-safe).

⇒ Be sure to check when using.

- E.g., *ArrayList*: not thread-safe, *Vector*: thread-safe.

Interface Queue<E>, and it's thread-safe implementations.

- *offer*(), *peek*(), *poll*(), *isEmpty*()

Interface *BlockingQueue<E>*

- *E take*() removes element from queue, waits if queue empty.
- *put(E o)* adds an element to queue, waits if (array-implemented) queue is full.
- *boolean offer(E o, long timeout, TimeUnit unit)*
- *E poll(long timeout, TimeUnit unit)*
- Implementations: *ArrayBlockingQueue, DelayQueue, LinkedBlockingQueue, PriorityBlockingQueue, SynchronousQueue*

ConcurrentLinkedQueue<E>

- Unbounded queue.
- $O(1)$ time operations (except *size*()! (why?))

SynchronousQueue<E>

- To synchronize *put* and *get* -operations (rendezvous)
- *put*() blocks until corresponding *get*() is done
- *get*() blocks until corresponding *put*() is done
- no *peek*() operation

ArrayBlockingQueue<E> - finite capacity

PriorityBlockingQueue<E> - priority queue

DelayQueue<E extends Delayed>

- Every element has a delay time, after which it can be taken from the queue.
- The one with most time past the delay are taken first.

Synchronizing other collections

- By default, *Collection*s are not thread-safe (synchronized).
- You can make any collection thread safe by
  `Collection<T> synchronizedCollection(Collection<T> c)`
- Similarly *List*, *Set*, *SortedSet*, *Map*, *SortedMap*.
- You still need to synchronize access:

```
List l = Collections.synchronizedList(new LinkedList());
...
synchronized(l) {
    Iterator i = l.iterator();
    while (i.hasNext())
      ...;
}
```

Java 8: parallel streams

- Streams execute functional operations on all elements of array/Collection.
- Parallel stream gives VM permission to execute them in parallel

Summary on threads

- Draw a picture on threads and their relations.
- Don't create useless threads.
- Be absolutely sure, which threads are executing at each stage.
- Minimize shared resources.
- Minimize cross traffic between threads (keep communication pattern clear).
- Control access to shared resources.
- Ensure that API classes are thread-safe.
- Don't lock for too long time (lock only for usage, and only for single usage at a time).
- Make sure that thread release the resources they lock.
- Make sure no thread starves.
- Test several times (even if it won't prove anything).
- Hand test with picture, check the picture.

References

[1] Jájá: Introduction to Parallel Algorithms.

[2] Akl S. G.: The design and Analysis of Parallel Algorithms. Prentice Hall 1989.

[3] Penttonen: Johdatus algoritmien suunnitteluun ja analysointiin.

[4] Culler: Parallel computer architectures...

[5] Wilkinson B., Allen M.: Parallel Programming: Techniques and Applications Using Networked Workstations. Prentice Hall, 1999.

[6] Natvig: Evaluating Parallel Algorithms. Theoretical and Practical Aspects. PhD theses, Norges Tekniske Høgskule, 1991.