Parallel matrix computations

(Gentle intro into a part of HPC)

Miroslav Tůma

Faculty of Mathematics and Physics

Charles University

mirektuma@karlin.mff.cuni.cz

Praha, February 12, 2024

Outline

1. Foreword
2. Computers, computing, communication
3. Parallel computing
4. Parallel processing and us - parallel programming
5. Parallel computer architectures: hardware and classification
6. Combining pieces together: computational models
   - Uniprocessor model
   - Vector and SIMD model
   - Multiprocessor model
7. Parallelizing problems
8. Sparse data decomposition: graph partitioning
9. Parallel and parallelized algebraic preconditioning

Introductory notes

- Created as a material supporting online lectures of NMNV532.
- Assuming basic knowledge of principles of numerical mathematics:
  - matrix-matrix and matrix-vector multiplication, factorizations,
  - algebraic iterative (Krylov space) and
  - direct (dense) solvers (elimination/factorization/solve)

Outline

1. Foreword
2. Computers, computing, communication
3. Parallel computing
4. Parallel processing and us - parallel programming
5. Parallel computer architectures: hardware and classification
6. Combining pieces together: computational models
   - Uniprocessor model
   - Vector and SIMD model
   - Multiprocessor model
7. Parallelizing problems
8. Sparse data decomposition: graph partitioning
9. Parallel and parallelized algebraic preconditioning

Basic terminology related to this text

Tools used to compute (process data, perform computations)

- **Computer**: device able to perform (process) automatically sequences of arithmetic or logical **instructions**.
- **Computation (data processing)**. Controlled by **program** or **code**.
- Modern computers consist of one or more of **data/program/code processing units** called also **processors**. If only one processor: **CPU (central processing unit)**.
- **Chip**: physical unit on which one or more processors reside.
- Processors typically **structured**: contain one, but rather more units called **cores**.
- **Computer architecture**: more detailed specification of the considered computer system.

Basic terminology related to this text II

Tools used to compute (process data, perform computations) II

- **Clock** or clock generator: signal used to coordinate actions of processors and digital circuits.
- **Clock rate / frequency**: frequency on which the computer clock is running. Another way to describe processor/processors speed: **cycle time**.
- Data and programs are stored in **memory**.
- Memory is typically **structured** and **hierarchically organized**

Basic terminology related to this text III

Splitting computation into smaller pieces

- Computation typically **decomposed** into smaller items **tasks, subtasks, instructions, stages**.
- Approximate **size of tasks**: **granularity**. We distinguish **large grain**, **medium grain** or **small grain** size of tasks.
- The tasks assigned to computational units: **processes** or **threads**.
- Threads; programmed instructions, bundled with data, to be managed by a **scheduler**.
- Mapping between the processes/threads and the computational environment is called **scheduling**.
- In order to achieve a correct **data flow**: processes need to be **synchronized**.

Basic terminology related to this text IV

Communication

Any computation process needs to communicate. At least internally.

- Communication: **internal** (as communication with memory) or **external** (as **I/O (input/output**).
- Communication based on an **embedded** communication network (links) (hardware) and programs (software).
- Communication characteristics:
  - **Bandwidth**: rate at which a link (links) can transmit data.
  - **Bisection bandwidth** (capacity across the narrowest bisector): measure of connection quality.
  - **Latency** describes a delay (amount of time) from input to the desired outcome/output. There are **more different types of latencies**: (processor-memory, network, internet, router, storage etc.) and we will mention some of them separately.

The most important characteristics of communication (from our point of view): **latency** and **bandwidth**.

Basic terminology related to this text V

Measuring computational performance

- **flop** (number of floating-point operations),
- **flops** (number of floating point operations per second; also plural of flop)
- **communication latencies**
- Everything together: **performance**

- All important parameters should be involved in **timing models**
- Our timing models are extremely **simplified**.

Outline

1. Foreword
2. Computers, computing, communication
3. Parallel computing
4. Parallel processing and us - parallel programming
5. Parallel computer architectures: hardware and classification
6. Combining pieces together: computational models
    - Uniprocessor model
    - Vector and SIMD model
    - Multiprocessor model
7. Parallelizing problems
8. Sparse data decomposition: graph partitioning
9. Parallel and parallelized algebraic preconditioning

Parallel computing

What is parallel computing and parallel computer

- **Parallel computer**: a set of processing, cooperating and communicating elements
- **Potential parallelism**: property of an application and of an algorithm to solve the computational problems.
- **Parallel computing**: ability of **concurrent** (**simultaneous**) computation/ data processing on more **computational units**. These units can be represented by more CPUs. Also **other** centralized or detached computational resources can perform the computational tasks simultaneously.
- Traditional **serial** or **sequential** computation is based on data processing on a single machine (computer/chip etc.) using either a single Central Processing Unit (CPU) or a single computational element. Now extinct. Still called here **uniprocessor**.

Parallel computing

Why parallel processing is of of interest: three questions?

Q 1: Why single processors are not enough?

- Consider the **computational power** measured by the **number of transistors** on a chip (this was often used approximate method to measure the computational power).
- 1971: chip 4004 : 2.3k transistors
- 1978: chip 8086 : 31k transistors (2 micron technology)
- 1982: chip 80286: 110k transistors (HMOS technology)
- 1985: chip 80386: 280k transistors (0.8 micron CMOS)
- 1989: chip 80486: 1.2M transistors
- 1993: Pentium: 3.1M transistors (0.8 micron biCMOS)
- 1995: Pentium Pro: 5.5M (0.6 micron)
- 1997: Pentium II: 7.5M transistors
- 1999: Pentium III: 24M transistors
- 2000: Pentium 4: 42M transistors
- 2002: Itanium: 220M transistors
- 2003: Itanium 2: 410M transistors

Parallel computing

Why parallel processing is of of interest: three questions?

- Performance improvement of **single processors** since 2002 only 20% per year,
- It has been approximately 50% per year between 1986 and 2002.
- Earlier chips: performance improvement strongly correlated to the **number of transistors** and to the **clock frequency.**
- Since 2003: difficulties to increase transistor density on a **single computational unit** on a chip. The clock frequency increase has started to stagnate.
- Since around 2003 (as AMD 64 Athlon X2, Intel Core Duo): **more "independent" computational units (cores)**
- Explanations are **technologically oriented**.
- This trend goes on.
- 2023: Apple A17: 19G transistors (3nm technology)
- 2023: Apple M2 Ultra: 134G transistors (5nm technology)
- 2023: AMD Instinct: 146G transistors (5nm technology)

Parallel computing

Q1 - answer: sequential processing has **inherent physical limitations**. What are these physical limitations?

1. Finite signal speed

Consider the **cycle time** (time per computer clock tick) and the **clock rate** (frequency of clock ticks). Then, e.g., the frequency

$$100\ MHz \quad \text{corresponds to} \quad 10\ ns \tag{1}$$

- The frequency of $2\ GHz$ then correspond to 0.5 ns. The **finite signal speed** (speed of light of $3.10^8\ ms^{-1}$), implies:
- With the cycle time 1 ns (frequency $1\ GHz$) **signal can pass** at most cca $30\ cm$ per the cycle time.
- With the cycle time 1 ps (frequency $1\ Tflops$) signal can reach at most the radius < c/rate ≈ 0.3 mm

Parallel computing

1. Finite signal speed

- An example of **wiring in computers** shows that **the speed can be critical**:
  - Early **and very mildly parallel** computer Cray 2 (1981) had about 10 km of interconnecting wires.
  - **“Small” physical size** of processors does not decrease the role of **insufficient signal speed in practice** at high frequencies:

Parallel computing

1. Finite signal speed

- Historical demonstration of **increased clock rates**:
  - 1941: Z3 (Konrad Zuse) 5 - 10 Hz
  - 1958: First integrated circuit: flip-flow with two transistors (built by Jack Kilby, Texas Instruments)
  - 1969: CDC 7600: 36.4 MHz (27.5 ns cycle time) (considered as the fastest computer until 1975)
  - 1976: Cray-1: 80 MHz (12.5 ns cycle time) (but throughput faster more than 4 times than for CDC 7600)
  - 1981: IBM PC: 4.77 MHz
  - 2011: AMD FX-8150 (Bulldozer) chips: cca 8.81 GHz (cca 0.12 ns)
  - 2022: Intel Core i9-13900K (not much less ☺)

Parallel computing

2. Limits in memory density

- Consider $1\ TB$ of memory. This means that on a chip of circular shape and area of πr^2, where $r = 0.3\ mm$ (from above).
- This circular area is of an approximate size 3.5 Ångstr$\ddot{o}$m$^2 \equiv 3.5 \times 0.01\ nm^2$ for one bit of information. And remind that
  - A typical **protein** is about 5 nm in diameter,
  - a molecule of **glucose** is just about 0.1 nm in diameter [**?**].
- We are **close to absolute limits of affordable density** to store information.

3. Technology and lithography limits

- **Production limits** arising from the possibilities of the electron-beam lithography.
- Early lithography resolution has been for Intel 4004 chips $10\ \mu m$.
- Later Intel processors as: Xeon Phi ($22nm$ lithography resolution), SPARC M7 (20 nm), contemporary GPUs (28nm), More new chips around 2020: 5nm lithography (Apple A14 Bionic, Apple M1, etc.), getting even more below (see above)
- Changing technology: **SSI** (1964), **MSI** (1968), **LSI** (1971), **VLSI** (1980), **ULSI**, **WSI**, **SoC**, **3D-IC** etc. But the pace of advancements **slows down.**
- In any case, **size of atoms and quantum effects as quantum tunelling** seem to ultimately limit this progress.

Parallel computing

Why parallel processing is of of interest: three questions?
4. Power and heat dissipation

- Transistor **speed** increases: new features need more transistors
- The corresponding **power** increases.

$$P_{CPU} = P_{dyn} + P_{short\ circuits} + P_{leakage}$$

logic gates, toggle gate current, leak – among differently doped parts

- Global overall guess
  P_{CPU}: switching-const × area-const × frequency × $voltage^2$
- **Power (heat dissipation)** density has been growing exponentially because of increasing clock frequency and doubling of transistor count.
- **Consequently**, processors with a clock rate significantly beyond the approximately 3.3 GHz are difficult and costly to be cooled using contemporary cooling techniques.

Parallel computing

5. Some early related predictions:

- Prediction: if scaling continues at present pace, by 2005, high speed processors would have power density of nuclear reactor, by 2010, a rocket nozzle, and by 2015, surface of sun. (Patrick Gelsinger, 2001)
  - **Cooling** is needed.
  - **Hot integrated circuits become unreliable**.
- **Dennard (MOSFET) scaling:** - scaling law roughly stating that chip power requirements are **proportional to area of the chip** (R. Dennard, 1974)
- Since cca 2005 this rule **seems to be not valid** anymore. Strong motivation to develop **multicore** processors.
- We start to get **dark silicon** – part of circuitry of an integrated circuit that cannot be powered at the nominal operating voltage given a thermal dissipation constraint. Successful research topic now.

Parallel computing

Q2: is it technologically possible to build the new and still more powerful parallel computational systems?

- earlier: some optimistic predictions, but possibly even now
- might seem that **processor technologies** are getting better steadily and very fast,
- might seem that **computers based on these technologies** are getting much faster.
- The power of processors expressed via **number of transistors on (chips / microprocessors / integrated circuits)** is expressed via an empirical observation and prediction called **Moore's law**:

Parallel computing

Q2: is it technologically possible to build the new and still more powerful parallel computational systems?

Observation

Moore's law: The number of transistors per square inch on integrated circuits doubles approximately from one to two years since the integrated circuit was invented (1965, Gordon E. Moore, co-founder of Intel recalibrated to two years in 1975)

- The law is sometimes restated that **chip performance doubles every 18 months** (David House, Intel executive, 1975) which combines the effect of more transistors on chip and having the transistors faster
- Dennard's scaling −− > power per Joule increases in this way; many more "laws".

Corollary

Number of cores will double every 18 months (A. Agrawal, MIT, 2009)

Parallel computing

Q2: Sketch of subsequent development

- 2005: Pentium D:230M+ transistors
- 2007: AMD K2 quad core - 2M L3: 463M transistors
- 2007: IBM POWER6: 789M transistors
- 2008: Intel Core i7 quad: 731M transistors
- 2008: AMD K10 quad core - 6M L3: 758M transistors
- 2009: AMD Six core Opteron 2400: 904M transistors
- 2010: Intel Six-Core Core i7 (Gulftown): 1170M transistors
- 2011: Six-Core Core i7/8-Core Xeon E5 (Sandy Bridge-E/EP): 2270M transistors
- 2012: Intel 8-Core Itanium Poulson: 3100M transistors
- 2013: Microsoft/AMD Xbox One Main SoC: 5000M transistors
- 2015: Sparc M7 (64-bit, SIMD, caches), Oracle, 10G transistors
- 2019: AWS Graviton2 (64-bit, 64-cores, SIMD, caches), Amazon, 30G transistors
- 2023: Apple A17: 19G transistors (3nm technology)
- 2023: Apple M2 Ultra: 134G transistors (5nm technology)
- 2023: AMD Instinct: 146G transistors (5nm technology)

The Moore's law even after 2003. But **Far beyond uniprocessor status.**

Parallel computing

Q2: Graphic processing units

- 1997: Nvidia NV3: 3.5M transistors
- 1999: AMD Rage 128: 8M transistors
- 2000: Nvidia NV11: 20M transistors
- 2000: Nvidia NV20: 57M transistors
- 2001: AMD R200: 60M transistors
- 2002: AMD R300: 107M transistors
- 2004: Nvidia NV40: 222M transistors
- etc.
- 2012: Nvidia GK110 Kepler: 7080M transistors
- 2013: AMD RV1090 or RV1170 Hawai: 6300M transistors
- 2015: Nvidia GM200 Maxwell: 8100M transistors
- 2018: TU106 Turing, Nvidia, 10.8G transistors
- 2019: Navi10, AMD, 10.3G transistors
- FPGA (field-programmable gate array) up to 20G transistors in 2014

Parallel computing

Q2: is it technologically possible to build the new and still more powerful parallel computational systems?

- Despite enormous technological progress there are more predictions that the Moore's law will **cease to be valid** around 2025.
- Technological point of view is more positive: Using more processing units can actually result in high gains. It comes with some **economy of scaling**. Using more processing units (processors, cores) can overcome the problems summarized above, can be efficient for problem solving and can be rather **cost efficient**.

Parallel computing

Q2: partial answers

Observation

Grosch's law: To do a calculation 10 times as cheaply you must do it 100 times as fast (H. Grosch, 1965; H.A. Grosch. High speed arithmetic: The digital computer as a research tool. (1953); H.A. Grosch. Grosch's law revisited. (1975)). Another formulation: **The power of computer systems increases as the square of their cost.**

Consequently, computers should obey the square law:

Observation

When the price doubles, you should get at least four times as much speed (similar observation by Seymour Cray, 1963).

Parallel computing

Why parallel processing is of of interest: three questions?

Q3: Are the parallel systems really needed?

- Computation of **climate models** (systems of differential equations simulating interactions of atmosphere, oceans, ice, land surface: far more accurate models needed; global 3D models needed
- Computation of re-entry corridor to get back to the terrestrial atmosphere: supersonic flow, Boltzmann equations
- **Protein folding**: misfolded proteins: Parkinson and Alzheimer
- **Energy research**: combustion, solar cells, batteries, wind turbines ⇒ large ODE systems
- **Crash-tests:** the need to solve large systems of nonlinear equations
- Computation of **turbulent flows**: large systems of PDEs in 3D.
- Big data analysis: LHC, medical imaging etc.
- Summarizing: **3D space/ time eats up the increase in power of today's computers**

Outline

1. Foreword
2. Computers, computing, communication
3. Parallel computing
4. Parallel processing and us - parallel programming
5. Parallel computer architectures: hardware and classification
6. Combining pieces together: computational models
   - Uniprocessor model
   - Vector and SIMD model
   - Multiprocessor model
7. Parallelizing problems
8. Sparse data decomposition: graph partitioning
9. Parallel and parallelized algebraic preconditioning

Parallel computing

- Why we cannot write codes that **automatically parallelize**? Could we rely on high-quality software technologies to convert the programs for parallel computations?
- Why this is of **interest for mathematicians**?

An answer to such questions: **parallel programming (coding)**.

- **There is a very limited success** in converting programs in serial languages like C and C++ into parallel programs
    - For example, multiplication of two square matrices can be viewed as a sequence of **linear combinations** or a sequence of **dot products**. Sometimes is better the first, sometimes the second.
    - Dot products may be very time consuming **on some particular parallel computer architectures**
    - In processing **sparse data structures** efficiently automatic techniques cannot be often used at all
    - Codes have to be often tightly coupled with particular applications in mind

Parallel computing

- **Parallelization may not be obtained by parallelizing individual steps.** Instead, new algorithms should be devised.
  - This is a strictly mathematical step and it is very difficult to automatize.
- Often **very different techniques** needed for **moderate number of cores** on one side and **large number of cores** on the other side.
  - Parallelism can be very different, as **task-driven** or even **data-driven** as we will see later.
- Automatizing processes may help, but often not sufficiently enough. Still new hardware/software concepts being developed.

Outline

1. Foreword
2. Computers, computing, communication
3. Parallel computing
4. Parallel processing and us - parallel programming
5. Parallel computer architectures: hardware and classification
6. Combining pieces together: computational models
    - Uniprocessor model
    - Vector and SIMD model
    - Multiprocessor model
7. Parallelizing problems
8. Sparse data decomposition: graph partitioning
9. Parallel and parallelized algebraic preconditioning

Parallel computer architectures

1. **Levels of parallelism - historical examples - a very brief sketch**

- Long time ago recognized that:
  - Parallelism **saves power** (electricity+power+cooling → less than 50 percent of operating costs (apc.com, 2003)),
  - improves **chip yield**,
  - and **simplifies verification**.
- Nowadays: more motivations
- Let us mention some historical milestones

Parallel computer architectures

1. **Levels of parallelism - historical examples - a very brief sketch**

1. **running jobs in parallel for reliability**
   IBM AN/FSQ-31 (1958) – some of them were purely duplex machine (time for operations 2.5μ s – 63.5 μ s; history of the word **byte**)
2. **running parts of jobs on independent specialized units**
   UNIVAC LARC (1960) – first I/O processor, world most powerful computer 1960-1961; interleaved access to memory banks
3. **running different jobs in parallel for speed**
   Burroughs D-825 (1962) – more modules, job scheduler; multiple computer systems
4. **running parts of programs in parallel**
   Bendix G-21 (1963), CDC 6600 (1964) – nonsymmetric multiprocessor; silicon-based transistors; first RISC; predecessor of I/O multithreading, 10 parallel functional units

Parallel computer architectures

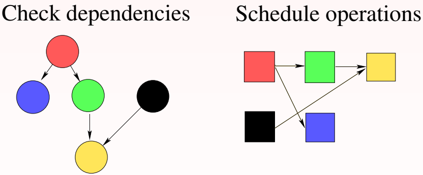
1. Levels of parallelism - historical examples - a very brief sketch

5. **development of multitasking with fast switching**: threads: 'light-weight' tasks sharing most of resources, typically inside a process, managed by the operating system scheduler.
6. **running matrix-intensive stuff separately**
   development of complex IBM 704x/709x (1963); facilities STAR 100, ASC TI (1965); 20MFLOPs ALU
7. **parallelizing instructions**
   IBM 709 (1957), IBM 7094 (1963)
   - **data synchronizer units** DSU → channels – enable simultaneously read/write/compute
   - **overlap** computational instructions / loads and stores
   - IBR (instruction backup registers)
   - **instruction pipeline** by splitting instructions in segments (will be explained later)

Parallel computer architectures

1. Levels of parallelism - historical examples - a very brief sketch

8. **parallelizing arithmetics: less and less clocks per instruction**
   - Static scheduling VLIW that can describe a rather complex instructions and data. Nowadays: threads
     - ★ Multiflow Trace (1984), then in IA64 architecture (Intel)
     - ★ sophisticated software optimization, simpler decoding and instruction scheduling
     - ★ difficult to predict dynamic events like missing data in local memories
   - Superscalar in RISC (CDC6000): operations scheduled at run-time



9. **bit-level parallelism**

2. Processing features in contemporary parallel computations

1. **FPU and ALU work in parallel**

   - Cray-1 (1976) had ALU (arithmetic-logical unit) rather weak
   - Strengthening ALU and simplifying processors pushed in 1980's development of **successful RISC workstations.**
   - **The idea behind RISC**: it is better to have less and much more optimized instructions.
   - Technological progress has brought efficient **CISC**s back ☺

Parallel computer architectures

2. Processing features in contemporary parallel computations

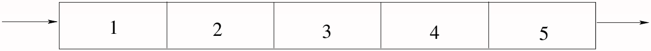
2 Pipeline for instructions

- Partition an instruction into several segments – called a **pipeline**
- pipelining instructions: an efficient form of **parallelism within a single processor**
- Consequently, more instructions can be processed concurrently processing **different segments of different instructions in parallel.**

An example: of standardized **RISC instruction pipeline**:

- Instruction **fetch** (fetches the instruction from memory)
- Instruction **decode** and register **fetch** (decode the fetched instruction)
- **Fetch** operands
- **Execute** operations
- Register **write back**.

This implies a possible overlap



Parallel computer architectures

2. Processing features in contemporary parallel computations

2 Pipeline for instructions - continued

- First use of pipelining: ILLIAC II (1962) project, IBM Stretch project, IBM 7094 (1969). Conceived even earlier: in the Z1 (1938) and the Z3 (1941) computers by Konrad Zuse.
- Contemporary processors can have from a few up to small tens of stages (superpipelined processors).
- Compiler task is to prepare instructions such that they can be efficiently pipelined. Pipeline delay due to waiting for data is called that the **pipeline stalls**. This is what must be avoided.
- Instruction pipelines everywhere.

Parallel computer architectures

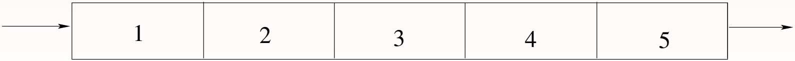
2. Processing features in contemporary parallel computations

3 Pipeline for data

- Pipelining data:
- Instead of segmenting instructions we can **partition operations**

An example: **adding two floating-point numbers**.

- **check** exponents
- possibly **swap** operands
- possibly **shift** one of mantissas by the number of bits determined by differences in exponents
- compute the **new mantissa**
- **normalize** the result



- Pipelining operations we get to the concept of program **vectorization**

Parallel computer architectures

2. **Processing features in contemporary parallel computations**

4 **Overlapping operations**

- Generalization of instruction pipelining is the concept of **overlapping** operations.
- Processors may have tools to find possible **dependencies** among different evaluations and overlap instructions even when they possibly have **different number of stages** with **differing amounts of time** to perform the operations.

Two specific cases:

- **Superscalar** processors are designed to schedule instructions at runtime, typically without a compiler support. That means that the scheduling is **dynamic**. See above, parallelizing arithmetic
- In contrast to this, the **VLIW processors** (very long instruction word with explicit descriptions what to do) mentioned also above schedule the instructions by the compiler preprocessing at compile time.

Parallel computer architectures

2. Processing features in contemporary parallel computations

5. **Multiple functional units**
   - advantages (parallelism) versus disadvantages (difficult to exploit)
   - Standard on chips, in cores
6. **Processor arrays**
   - ILLIAC IV (1972) with 64 elementary processors
   - concept of **graphic cards**
7. **Multicore and manycore processing**
   - multicore processors with tens of cores
   - manycore processors with hundreds of cores
   - specific control needed: **simultaneous multithreading (SMT)** (hyperthreading) – more threads and schedule executable instructions from different threads and that can be even from different processes in the same cycle. It is a **thread analogy** of superscalar processing.
   - Considering a chip with more cores as a bundle of logical processors.

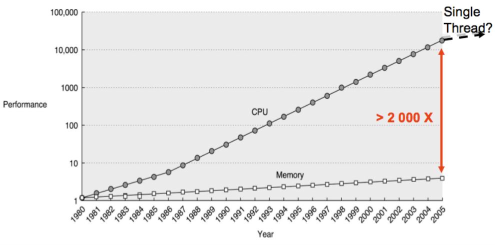
Parallel computer architectures

3. Summarizing recent history of parallel computing in a few slogans

- Seventies of the 20th century were characterized by **data pipelining and vector computations** in general,
- eighties can be considered as a revival of computer architectures with **reduced instruction sets** and **strong integer arithmetic**,
- nineties started with practical use of **multiprocessors** and several very successful **massive parallel systems** and
- later we saw a widespread use of **hybrid and massively parallel** computational tools.

4. Computer memory issues

1. **Memory hierarchy: general view**
   - speed × respond time × cost
   - **registers** (very high-speed memory accessible by computational units)
   - **cache** (locally accessible high-speed memory)
   - **main memory** - gigabytes, access speed around units of GB/s
   - **disc storage** - terabytes, access speed around hundreds of MBytes/s
   - memory available via **network** etc.



4. Computer memory issues

2. **Memory components/functionality**
    - **Physical address**: actual address where data is stored
    - **logical (virtual) address**: address generated by CPU(s). Logical addresses can span **much larger space** called **virtual memory**.
    - Memory management units does the translations between physical and logical addresses using **relocation register**.
    - **Paging** is a scheme to manage exchanges of memory needed by computations caused by different logical and physical memory. It is a way to split data into chunks that can be easily indexed and exchanged.
    - **Segmentation** is another way of logical address allocation alternative to paging. Segments are protected areas of variable sizes that are used to partition the address space according to its contents.
    - **Swap space**: this is a space that substitutes for physical memory. Enable to use much larger logical space. Swap in × swap out.

4. Computer memory issues

2. **Memory components/functionality**
    - **Memory thrashing**: denotes a problem when the computation spends a lot of time to solve problem with data exchanges between physical and logical space. This may strongly slow down computations by this data manipulation overhead.
    - **Page fault** represents the situation when data page is not available for a computation and should be retrieved from other (lower) levels of memory hierarchy.

4. Computer memory issues

3. Very fast memory: cache: **low-latency high-bandwidth storage.** From the hardware point of view: main memory typically composed from **DRAM (dynamic random access memory)** chips, cache uses **SRAM (static random access memory)**: fast access, but smaller capacity per area.
   Sketch of a typical **cache hierarchy**:
   - Level 0 (L0): micro operations cache
   - Level 1 (L1) Instruction cache (kBytes)
   - Level 1 (L1) Data cache (kBytes)
   - Level 2 (L2) Instruction and data cache (MBytes)
   - Level 3 (L3) Shared cache
   - Level 4 (L4) Shared cache

4. Computer memory issues: cache terminology

- **Cache hit**: processor has found data ☺
- **Cache miss**: not the previous case, measured by **miss-ratio**
- Cache **blocks, lines** express the cache structure
- Cache write policy
  - **Write-through**; data are (pseudo)simultaneously updated both in cache and memory.
  - **Write-back (write-deferred)**; data are update only in cache. Later in memory.
  - **cache thrashing**: degradation of performance due to insufficient caches
  - **cache sharing**: sharing data for computational units in the same cache lines

4. Computer memory issues: cache terminology

- **Cache mapping**: similarly as mapping between pages and memory, cache must have some mapping policy
    - **Directly mapped** caches map memory blocks **only to** specific cache locations.
    - **Fully associative** caches can map the memory blocks to any cache position. Asssociative memory used to store content and addresses of the memory word.
    - Compromise solution between directly-mapped and fully associative caches are **set associative caches**. It is an enhanced form of direct mapping.

Parallel computer architectures

4. Other computer memory issues

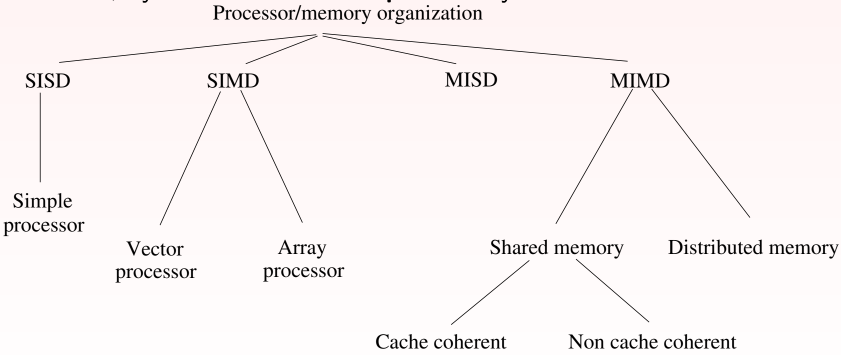
4. Why memory management?
   - minimize fragmentation, keep track of allocated and deallocated data chunks, keep data integrity
   - Difficult for multiprocessors.

5. Interleaving memory using memory banks
   - A way to decrease memory latency
   - The interleaving is based on the concept of **memory banks** of equal size that enable to store **logically contiguous** chunks of memory as incontiguous vectors in different parts of memory using a regular mapping pattern.
   - Mention example of Cray-2

Parallel computer architectures

5. Taxonomy of architectures by Flynn

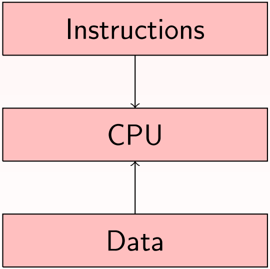
Simple macro classification of parallel computers proposed by Flynn. It considers main features of computers represented by **data and control flows.** Used acronyms represent by **S** the word **single**, by **I** the word **instruction**, by **M** the word **multiple** and by **D** the word **data**.



Parallel computer architectures

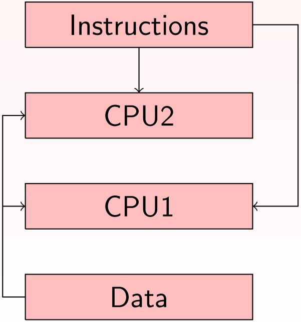
5. Taxonomy of architectures by Flynn

2. SISD: single instruction single data stream
   - traditional (von Neumann) single CPU processor (computer)
   - extinct type of architectures (but useful as a model)
   - here we sometimes call it **uniprocessor** in order to emphasize SISD character (that does not exist in practice)



5. Taxonomy of architectures by Flynn

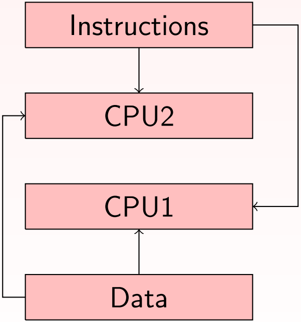
3. MISD: multiple instruction single data stream
   - mostly experimental architectures – difficult to have the whole architecture based on this principle and having it efficient
   - example: single data: **angle**, computing $\sin(angle) + \cos(angle)$.
   - some MISD architectures useful as computers that compute **and** detect and mask errors for the single data stream



Parallel computer architectures

5. Taxonomy of architectures by Flynn

4. SIMD as a prevailing principle
   - vectorization
   - matrix processors
   - supercomputers



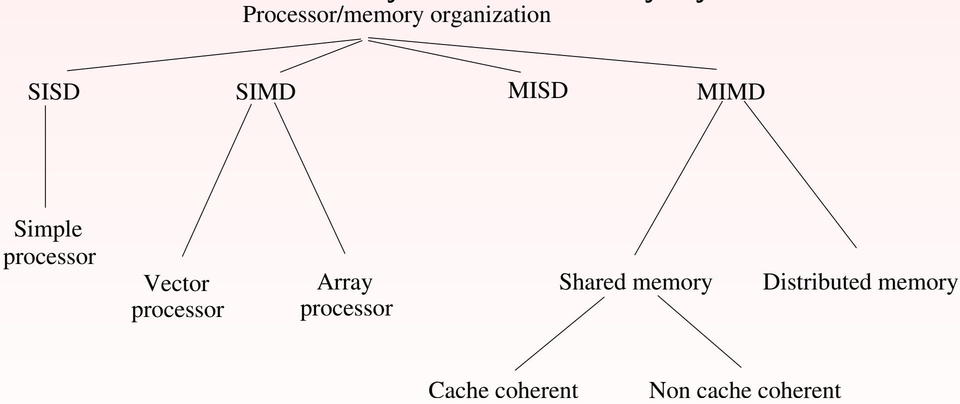
Parallel computer architectures

5. Taxonomy of architectures by Flynn

5. MIMD: multiple instruction – multiple data streams
    - the most general case
    - any interconnection for sending data and instructions, in general
    - Cosmic Cube built at Caltech in 80's, Cray X-MP/2
    - iPSC 860 by Intel
    - problem of **cache coherence**
        - ★ consistency of shared data that can be distributed over more **local** caches

Parallel computer architectures

5. Taxonomy of architectures by Flynn



Parallel computer architectures

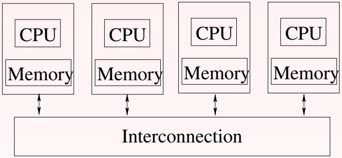
5. Taxonomy of architectures by Flynn

4. (continued) Other possible classification of MIMDs:
    - By **memory access** (local/global caches, shared memory caches, cache only memory, distributed (shared) memory),
    - by **topology and interconnection** (master/slave, crossbar, pipe, ring, array, torus, tree, hypercube, ...).

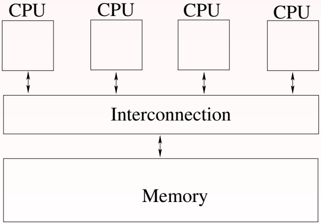
Parallel computer architectures

5. Taxonomy of architectures by Flynn

- Multicomputers - MIMD computers with distributed memory: clusters, grid systems



- Multiprocessor systems - MIMD computers with shared memory



6. Interconnection network and routing

- **Interconnection network** (IN, interconnection, interconnect) physically connects different components of a parallel computer but it can describe an outer network as well.
- Its topology describes the actual way how the modules (nodes, memories etc.) are connected to each other. The topology can be static or dynamic.
- **Routing** describes the way how the modules exchange information. The routing can be described as a union of two components.
  - Routing algorithms determine paths of messages between their sources and sinks.
  - Switches are devices that connect components of the interconnect together. They manage the data flow across the interconnect. Switching strategy determines possible cutting of the messages transferred by an interconnection network into pieces.

Parallel computer architectures

6. Interconnection network and routing

A. Static (and possibly also dynamic) interconnections

The main issues and concerns

- **pure connectivity**: how many links are used to connect nodes, minimum, maximum, weakest points
- connectivity for **communication**: lengths of interconnecting paths.
- **cost, static and dynamic complexity of interconnection**
- **extensibility**: important mainly for reconfigurable architectures (as home-made clusters)

All these items have implications for **bandwidth** and **latency**

6. Interconnection network and routing

A. Static interconnections

Standard model for static interconnection networks is a **graph**, often **undirected** since the interconnecting lines can be typically used in both directions. Its few characteristics:

- **Diameter** is a maximum distance between any pair of graph nodes. Distance of two nodes in a graph is the length of the shortest path between them.
- **Bisection (band)width**: minimum number of edges that should be removed to partition the graph into two parts of equal node counts
- **Degree** of a node is the number of adjacent vertices.
- **Node/edge connectivity** is the number of nodes/edges that have to be removed to increase the number of components of the originally connected graph.

6. Interconnection network and routing

A. Static interconnections

One could prefer, for example:

- **small diameter** of the static interconnection,
- **large bisection bandwidth**,
- **large connectivity** or
- **small average node degree**.

Parallel computer architectures

6. Interconnection network and routing

A. Static interconnections: examples

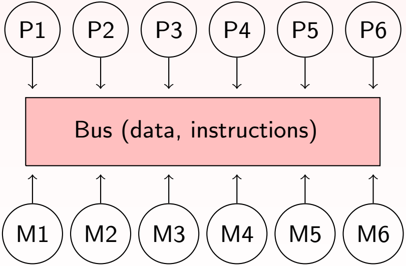
- a complete graph, **linear graph**, **binary tree**, **fat tree** as in CM-5, **cycle**, **2-dimensional mesh**, **2-dimensional torus**. An important case: a **d-dimensional hypercube** with 2^d nodes.

connection	max deg	diameter	edge connect	bisect width
completely connected	$p-1$	1	$p-1$	$p^2/4$
star	$p-1$	2	1	1
binary tree $p=2^d-1$	3	$2\log_2((p+1)/2)$	1	1
d-dimensional mesh	$2d$	$d(\sqrt{d}p-1)$	d	$p^{\frac{d-1}{d}}$
linear array	2	$p-1$	1	1
d-hypercube $p=2^d$	$\log_2 p$	$\log_2 p$	$\log_2 p$	$p/2$

6. Interconnection network and routing

B. Dynamic interconnections

1. Bus
   - Set of **communicating lines** that connect modules.
   - Unidirectional, bi-directional, separate address and data lines
   - Efficient bus should contain at least two communication paths, one for **instructions** and the other one for **computational data**.



Parallel computer architectures

6. Interconnection network and routing

B. Dynamic interconnections

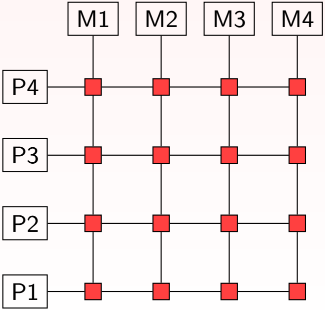
1. Bus

   - **Bounded bandwidth**: limited number of nodes connected in practice.
   - **Constant time** for an item of communication among **limited number of nodes**
   - Consequently, **scalable in cost but not scalable in performance.**

6. Interconnection network and routing

B. Dynamic interconnections

2. **Dynamic networks with switches (crossbar networks)**
   - Interconnection network that **completely interconnects processing elements with other modules** using a set of switches.
   - For simplicity, assume p (P1-Pp) processing elements and m (M1-Mm) memory banks.



Parallel computer architectures

6. Interconnection network and routing

B. Dynamic interconnections

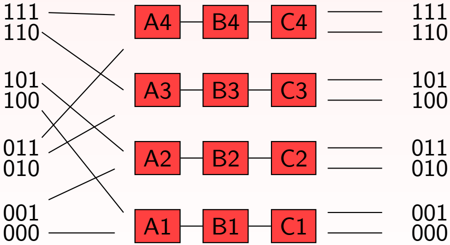
2. **Dynamic networks with switches (crossbar networks)**
   - We need pm switches. Then $min(p, m)$ lines can work in parallel.
   - Useful for **small-scale** parallel computers.
   - Assuming $m \geq p$ (a reasonable practical assumption), the complexity of the interconnection grows at least as $\Omega(p^2)$.
   - Consequently dynamic networks with switches are **not much scalable in cost**.

6. Interconnection network and routing

B. Dynamic interconnections

3 Multistage interconnection networks

- Several stages of switches interconnected by communication lines



6. Interconnection network and routing

B. Dynamic interconnections

3. Multistage interconnection networks
   - Using a limited number k of serially connected blocks of switches called stages can be a reasonable compromise.
   - Considering k stages with w nodes each and n links between two neighboring stages, then, in a regular multistage interconnection the **node degree** is equal to

     $$g = n/w. \tag{2}$$

     Connection can be expressed by a permutation

     $$\pi : \{1, \dots, n\} \to \{1, \dots, n\}. \tag{3}$$

   - **More scalable in cost than crossbar** and **more scalable than bus in performance**

6. Interconnection network and routing

B. Dynamic interconnections

3. More on multistage interconnection networks
   - Basic interconnections
     - ★ **perfect shuffle** (cyclic shift left)
     - ★ $000 \to 000$, $001 \to 010$ etc.
     - ★ **baseline**
     - ★ rotate last $i + 1$ bits right
     - ★ **butterfly**
     - ★ interchange bits at positions 0 and i
   - Easy to code interconnections by binary inputs.
   - **omega network** (our figure)
     - ★ N processing elements, $\log_2(N)$ stages, $N/2$ processing elements per stage
     - ★ perfect shuffle (cyclic shift)
   - blocking versus non-blocking connections (more technical issue).

Parallel computer architectures

6. Interconnection network and routing

B. Dynamic interconnections

4. Fat tree interconnections
   - **Fat tree** network with the topology of full binary tree.
   - The number of actual connections between nodes at different (neighboring) levels increases in order to support "long distance" communications via the root of the network.
   - Problems with mapping unstructured problems to a computer architecture with such interconnect
   - CM5 computer

6. Interconnection network and routing

C. Routing and switching

- **Routing** + **switching**: determine **communication** between **sources** and **sinks (destinations)**, splitting messages and **the manner of sending** the messages from one node to another.
- avoiding **deadlocks**: strategies to resolve **interconnect conflicts**
- **deterministic** and **adaptive** algorithms for routing
- Always **better message aggregation, communication granularity, communication regularity** needed

6. Interconnection network and routing

C. Routing and switching

- **Timing models for routing and switching**
  - **bandwidth of a connection** mentioned above is a maximum frequency at which data can be communicated in **bytes per second**
  - Its inverse is called the **byte transfer time**
  - The **transport latency** (for m bytes)

$$T(m) = T_{startup} + T_{delays\_on\_the\_route} + T_{finish} + t_B\, m. \tag{4}$$

  - : simplified:

$$T(m) = T_{transport\_latency} + t_B\, m. \tag{5}$$

6. Interconnection network and routing
Routing and switching

1. Routing with **circuit switching**
   - Setting up and reserving a **dedicated communicating path (channel, circuit)**. This path is guaranteed for the whole transmission in advance.
   - **Bandwidth** fixed. Circuit switching can be also classified as **connection-oriented**.
   - The path is set up by sending small **control probe messages**.
   - No packets (typically), no buffers, (dedicated) circuit kept all the time. The communication time model for the dedicated communication that uses l independent communication links and sends a message of the size m can be given by

$$T_{circuit}(m,l) = T_{overhead} + t_{control\_message}\, l + t_B\, m. \tag{6}$$

   - Useful for long messages, not communicated often.

5. Interconnection network and routing
Routing and switching

2. Store-and-forward routing/switching (with **packets**)
    - Message split into **packets** – can be transmitted over different paths.
    - Most general way of sending using more links.
    - Intermediate node store received packets before passing them on.
    - The packets carry in its **header** the control information used to determine the path for the packet. In contrast to the routing with circuit switching, the **transfer time increases with the number of switches** to be passed.
    - Errors can be checked on the way.
    - The time for sending a message with the store-and-forward routing (transport latency) is approximately for the message size m, l independent communication links and the byte transfer time t_B.

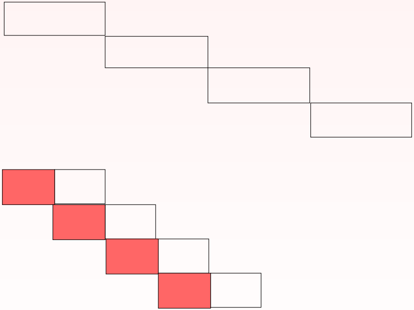
$$T_{store-and-forward} \approx T_{overhead} + l\, t_B\, m. \tag{7}$$

    - Needs to add the **packetize time**

6. Interconnection network and routing
Routing and switching

3. Packet routing
   - Uses **pipelines**

General difference between the store-and-forward routing and packet routing schematically shown here.



Store-and-forward routing (top figure) and packet routing that uses pipelining of the packets (bottom figure).

6. Interconnection network and routing
Routing and switching

4. Cut-through routing with packets: example of optimized routing
    - Optimized packet routing: only one way for a packet
    - Extends the idea of pipelining
    - First a **tracer** establishes the connection
    - Message is broken into fixed size units called **flow control digits (flits)** with much less control information than packets. This implies that the flits can be rather small.
    - Typically in tightly coupled parallel computers with reliable interconnect that enables to make the error control information very compact.
    - In general, it is possible to face a **deadlock**, for example, when sending messages in a circle and if some line is temporarily occupied.

Parallel computer architectures

6. Interconnection network and routing

Delivery schemes: relations sender(s)/receiver(s)

- **Unicast**: message to a specific node.
- **Broadcast**: one sender and multiple receiveres: one-to-all association
- **Multicast**: one-to-many-of-many, many-to-many-of-many
- **Anycast**: ☺☺

6. Interconnection network and routing

More details on communication

- **Blocking operations:**
  Returns control to the calling process **only after all resources (buffers, memory, links)** are ready for next operations.
- **Non-blocking** operations: returns the control to the calling process **after the operation has started** and not necessarily finished. Strategies to avoid deadlocks needed.
- **Synchronous communication**: both sending and receiving process start the operation once the communication is set. (Often for shared-memory/SIMD systems.)
- **Asynchronous communication:** No such rule for the asynchronous communication. A specific way by **message passing** can be both synchronous or asynchronous depending on the algorithms and possibilities of the communicator.

7. Measuring computation and communication

Time models

- A model for **sequential time** to compute n sequential operations

$$T_{seq} = n * (T_{seq\_latency} + T_{flop}), \tag{8}$$

- Simple parallel model

$$T_{par} = T_{par\_latency} + \max_{1 \le i \le p}((T_{flop})_i \tag{9}$$

- Remind: three main timing aspects that should be taken into account on a rough level.
  - **Bandwidth** that limits the speed of communication
  - **Latencies** of various kinds
  - **Time to perform numerical operations** with data

7. Measuring computation and communication

Time models: Speedup S

- The power of parallel processing with respect to purely sequential processing is often measured by the **speedup**.

$$T_{seq}/T_{par} \tag{10}$$

- **Variations** may consider related **latencies**.
- Multiprocessors with p processors typically have

$$0 < S \leq p, \tag{11}$$

- Pipelining:

$$S = n * p/(n+p) \sim p, \tag{12}$$

- More detailed:

$$S = n * p * T_{seq}/(T_{vec\_latency} + (n+p) * T_{vec\_op}). \tag{13}$$

Time models: Speedup S

Graphical demonstration of **data** pipelining speedup for $p = 5$ processing the vector

$$a = \begin{pmatrix} a_1 & a_2 & \dots a_n \end{pmatrix}$$

is

$time$	$segment1$	$segment2$	$segment3$	$segment4$	$segment5$
1	a_1				
2	a_2	a_1			
3	a_3	a_2	a_1		
4	a_4	a_3	a_2	a_1	
5	a_5	a_4	a_3	a_2	a_1
	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$

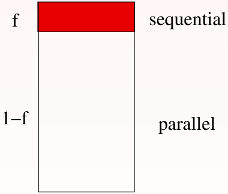
7. Measuring computation and communication

Time models: Speedup S

- $$S_p = T_1/T_{par}, \tag{14}$$
- Efficiency:
  $$S/p \tag{15}$$
- $$0 < E \leq 1, \tag{16}$$

Time models: Amdahl's law

- **Amdahl's law** expresses a natural surprise over the fact that if a process performs part of the work **quickly** and part of the work **slowly** → overall (speedup, efficiency) strongly limited by the slow part.
  - f: fraction of the slow (sequential) part
  - $(1-f)$: the rest (parallelized, vectorized)
  - t: overall time
- $$S = \frac{f*t + (1-f)t}{f*t + (1-f)*(t/p)} \le \frac{1}{f} \qquad (17)$$



- Overall: **significant simplification** (missing dependency on the problem size, actual search space etc.)

Scalability

- Program (code) is scalable if **larger efficiency comes with larger amount of parallelism**
- **linear, sublinear**, **superlinear** efficiency.
- Specialized definitions of **scalability** in specific cases

7. Measuring computation and communication

Scalability

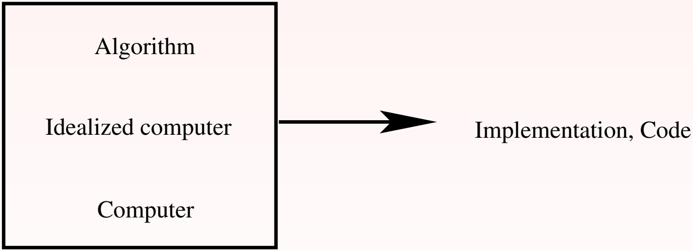
- Consider solution time varying with the number of processors for a problem with a **fixed total size**. The code is (approximately) **strongly scalable** if the speedup is (approximately) equal to p (number of processing elements). That is, the code is strongly scalable, if $T_{par} = T_{seq}/p$. Strong scalability is often difficult to achieve for large p because of the communication.
- Consider solution time varying with the number of processors for a problem having a **fixed size per processor.** The code is (approximately) **weakly scalable** if the code run time stays constant when the workload is increased proportionally to the number of processors. In contrast to the strong scalability, the weak scalability is often easier to be achieved.

Outline

1. Foreword
2. Computers, computing, communication
3. Parallel computing
4. Parallel processing and us - parallel programming
5. Parallel computer architectures: hardware and classification
6. Combining pieces together: computational models
    - Uniprocessor model
    - Vector and SIMD model
    - Multiprocessor model
7. Parallelizing problems
8. Sparse data decomposition: graph partitioning
9. Parallel and parallelized algebraic preconditioning

8. Combining pieces together: computational models

Algorithms → architecture_aware_implementations → Computers



8. Combining pieces together: computational models

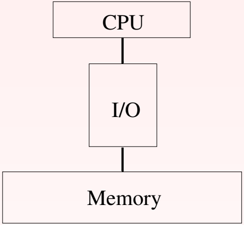
- Idealized **uniprocessor**.
  - latencies processor-memory
  - possible superscalar processing, concurrent data movement
  - threads to be quickly switched
  - instruction pipelining (tacitly assumed)
- Idealized processor with **data pipeline** (vector processor; idealized SIMD computational model)
  - Also data pipelining
  - Sparse data from the point of view of access and pipelining
  - their storage schemes
  - more threads for possible concurrency
- Idealized computers with **more processors** (MIMD)
  - many more concepts, the highest dependence on architecture
  - problemgranularity
  - problem partitioning
  - load balancing
  - multicore, manycore

Uniprocessor model

Uniprocessor model

Uniprocessor model

von Neumann architecture



- There is no pure uniprocessor nowadays. Even a simple Pentium III has on-chip and in its firmware:
  - **instruction level parallelism** (up to 3 instructions)
  - **pipeline** (at least 11 stages for each instruction)
  - **fine-grained data parallelism** (SIMD type) like MMX (64bit) and SSE (128bit)
  - **more threads** at system level for **more cores**
- What can be influenced by us?

Uniprocessor model

What can be influenced by us?: **ways to hide latencies**

- Mostly these are: **data related items**
- Restructuring the code so that
  - caches can be used efficiently
  - dynamic-out-of-order scheduling is enabled (may need tiny register/cache workspace)
  - strong multithreading with more threads that can be fast switched is possible
  - prefetching is easy - (this is preparing data to be understood that they will be used soon)

Uniprocessor model

Outside our control (typically)

: the rest ☺

- **Low-level control** of data pipelines that is an ability to issue more (data processing) instructions at the same time that need
  - Detecting **true data dependencies**: dependencies in processing order
  - Detecting **resource dependencies**: competition of data for computational resources
  - **Reordering instructions**. Note that most microprocessors enable **out-of-original-order** scheduling
  - Solving **branch dependencies** that can be performed by various ways
    - ★ **speculative scheduling** across based on the assumption that typically every 5th-6th instruction is a branch
    - ★ **compile time scheduling**. This is a problem that can be solved by VLIW since the instructions are more complex. Other thread scheduling is often accessed by users.
    - ★ **superscalar scheduling**
- But, all of these should be enabled.

Uniprocessor model

- The following example indicates how the above-mentioned features can be enabled.

Example

Consider a uniprocessor:

- clock frequency 2GHz,
- main memory: DRAM with latency $0.1\mu s$,
- two FMA (floating-point multiply-add) units enabling 4-way superscalar processing (4 instructions in a cycle, e.g., two adds and two multiplies),
- two (double precision) words (each of 8 bytes) are obtained in a fetch, that is within the latency time.
- this means: 2 data fetches for 4 operations.

Uniprocessor model

Case 1: No effort to minimize memory latency

- The **clock cycle time**: $= 1/frequency \equiv 1/(2.10^9)s = 0.5\ ns$
- Since the processor can **theoretically** process $2 \times 10^9 \times 4$ instructions per second then the **maximum processor rate** is 8 GFLOPs.
- The **memory is much slower**: every memory request needs 0.1 μs wait time **memory latency**.

Example: a **dot product** of two vectors (**of infinite length**).

- One multiplication with two numbers and adding the result to the partial product (two operations) need then 1 fetch: **2 operations for a fetch ($0.1\mu s$) of two numbers**.
- That is 2 operations / $0.1\mu s \rightarrow 2 \times 10^7 \text{ operations per second.}$
- This leads to the **rate of 20 MFLOPs** – much smaller than the potential of the processor.

Uniprocessor model

Case 2: Hiding latency using cache

Example: $A * B = C$, hiding latency by cache. Assume **cache of size** 64**kB** with latency of $0.5\ ns$.

- The memory size needed to store one number is 8 bytes. The cache **can store** three matrices A,B and C of dimension 50: $3 \times 50^2 \times 8 = 7500 \times 8 = 60000$ bytes.
- A matrix fetch of A and B: 5000 words, 8 bytes each. Due to the cache that can store the matrices, this needs $5000/2 \times 0.1 = 250\ \mu s$. For the transfer considered **only latency and not the bandwidth**.
- Once the matrices are in cache, operations can be performed.
- Asymptotically, $2n^3$ **operations are needed**. If the computer performs **4 operations per cycle**, we need $2 \times 50^3 \times 0.5\ ns\ (clock\ cycle)\ \times 0.25 = 125000/4\ ns \approx 31\ \mu s$
- This gives $281 \mu s$
- Resulting rate is $2 * 50^3 / 0.000281 \approx 890$ MFLOPs. Close to this if added moving C back to memory.

Uniprocessor model

Case 3: Hiding latency using multithreading

Algorithm (**Matrix-vector multiplication: Standard dot products of rows of $A \in R^{m \times n}$ with $b \in R^{n}$.**)

Input: *Matrix A.*
Output: *Row products.*

```
for $i = 1, \dots, m$ do                    ▷ Loop by rows
    $r_i = A(i,:) * b$
end for
```

Uniprocessor model

Case 3: Hiding latency using multithreading

A multithreaded version of the previous multiplication (symbolically written).

Algorithm (Matrix-vector multiplication: Multithreaded dot products of rows of $A \in R^{m \times n}$ with $b \in R^n$.)

Input: *Matrix A.*
Output: *Row products.*

```
for $i = 1, \dots, m$ do                                        ▷ Loop by rows
    $r_i = new\_thread(dot\_product, double, A(i,:), b)$
end for
```

Uniprocessor model

Case 3: Hiding latency using multithreading: general notes

- In situations like above, more threads than cores/processors is useful as a way to hide slow communication/memory.
- But note that threads consume some amount of memory. And they may **share the same cache**. The optimal number of threads is strongly architecture-dependent.

Uniprocessor model

Case 3: Hiding latency using multithreading (continued)

- Various modifications of multithreading on contemporary computers
- More ways to support more threads processed by one chip
  - **Fine-grain** multithreading (switch between threads on every cycle)
  - **Coarse-grain/block** multithreading (switching among the threads can be based on I/O demands or long/latency operations)
  - **Simultaneous** multithreading (more instructions + more threads; parts of different threads share, for example, a superscalar unit)
  - Combination of the techniques above; combined scheduling for more supescalar units.
- Predecesssor of using therads massively: VLIW as in Tera MTA (2002)

Uniprocessor model

Case 4: Hiding latency using prefetch

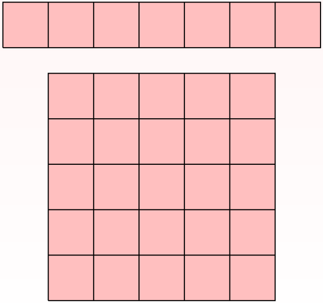
- Boosting performance by advancing fetches from slower parts of memory hierarchy
  - Prefetch of instructions
  - Prefetch of data
- Data for the prefetch need to be prepared/enabled. Possible use of prefetch processor.

Next slides summarize our goals

Uniprocessor model

Case 5: Data preparation: improving memory bandwidth and latency: locality and regularity

- **spatial locality**: data are spatially local if the data items stored close (at logically close positions) to the executed items are highly probable to be executed soon. In this case, the use of **prefetch** with high chance to improve execution.
    - Examples: vectors, matrices



Uniprocessor model

Spatial locality and matrix layout

- Memory layout should be such that **physical access** of memory is compatible with the **logical access** (sometimes forced by the programming language).
- In particular: **column major** versus **row major**

Algorithm (Summing columns of $A \in R^{m \times n}$.)

Input: *Matrix A.*

Output: *Resulting vector sum of the sums.*

```
for $i = 1, \dots, m$ do                ▷ Loop by rows
    $sum_i = 0$
    for $j = 1, \dots, n$ do            ▷ Getting row sum
        $sum_i = sum_i + A_{ij}$
    end for
end for
```

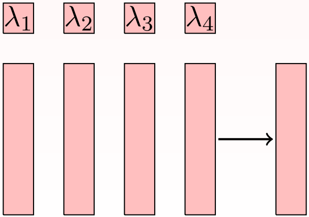
- Here: columnwise A is bad, rowwise A is good

Uniprocessor model

Case 5: Data preparation: improving memory bandwidth and latency: locality and regularity

- **temporal locality**: Data are temporally local if the data items recently executed have a high chance to be executed soon again. Such data can be "hanged" in registers or cache for a long time.
  - example: linear combination of a set of vectors $b_i, i \in S$. The coefficients λ_i should have a high temporal locality being reused a couple of times within a short time

$$\sum_{i \in S} \lambda_i b_i$$



Uniprocessor model

Case 5: Data preparation: improving memory bandwidth and latency: locality and regularity

- **regularity** of processed data: this means that the code is often faster and easier to be processed by the computer software when composed from **similar, and possibly standardized** blocks
- Improving regularity/bandwidth sometimes possible by **tiling**:
  - fragmentation of blocks
  - picking up blocks from a sparse structure

Uniprocessor model

Catching more rabbits at the same time

- Coding to achieve localities and regularity
- Standardization
  - increase **readability of codes** and simplify **software maintenance**,
  - improvements in **code robustness**,
  - better **portability, modularity** and **clarity**,
  - increase in **effective memory bandwidth**,
  - creating a **basis for machine specific implementations** etc.
- Overall: BLAS1 set of subroutines / library (1970's) (AXPY ($\alpha x + y$), dot_product ($x^T y$), vector_norm, plane rotations, etc.)
- Other BLAS for keeping localities and regularity
- Further BLAS-like development: see below

Vector and SIMD model

Vector processor and SIMD models

- **Vectorization**: one of the most simple methods to introduce parallelism into computations. Data pipelining is behind.
- Vector based machines (and not only): **instructions also pipelined** (computer instructions are divided into several **stages**, see above)
- Provided standard support in architectures as
  - **vector registers for instructions**
  - **vector registers for data**
- Contemporary computers: hardware that supports vectorization **on chips** (with caches, multiple pipelines etc.) Often developed to support **multimedia applications**.

Vector processor and SIMD models

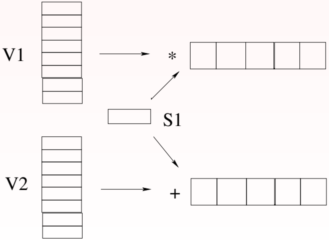
Some historical notes

- CDC series and and Cray computers: one of the most **successful chapters** in the development of parallel computers.
- A lot of early progress connected to Seymour Cray (1925 – 1996; father of supercomputing, chief constructor of latest model of CDC computers with some earliest parallel features, constructor of the first CRAYs: commercially successful **vector** computers (supercomputers) (Cray-1 (1976); Cray X-MP (1983); Cray C-90 (1991) etc.)
- In 70's **memory-memory** vector processors and **vector-register processors** existed side by side. The latter prevail nowadays.

Vector processor and SIMD models

Some vector processing principles and characteristics

- **Vector pipelines on chips** as in superscalar-based processing units × vector processing
- **Vector supercomputers** with typically different **(multiplied)** vector functional units / vector processing units for different operations.
- **Load/store** units may be also efficiently vectorized.
- Architecture includes also **scalar** units. Small efficiency of the scalar arithmetic → RISC workstations with efficient FPU+ALU.



Vector processor and SIMD models

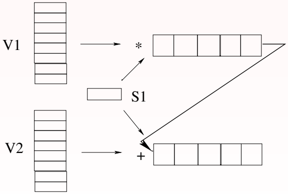
Some vector processing principles and characteristics

How can be characterized processing on a vector (supercomputer) architectures.

- Some early indicators
  - R_∞: **computer speed** (for example, in Mflops) on a vector of infinite length,
  - $n_{1/2}$: vector length needed to **reach half of the speed** R_∞,
  - n_v denotes the **vector length needed to get faster processing** than in the scalar mode.
- Some other terminology below

Chaining

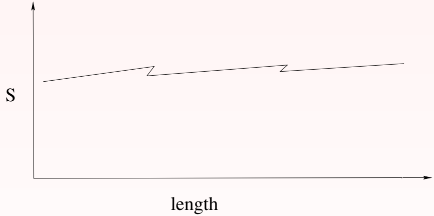
- **Chaining** represents a way of computation developed for early Crays (Cray-1 (1976), predecessor project STAR) and used since then.
- Based on **storing intermediate results of vector pipelines**, combining them possibly with scalar data and using them directly without communication with main memory: **supervector** performance.
- The process controlled by the main instruction pipeline. Closely related **overlapping** introduced for vector operations by Cray-1.



Vector processor and SIMD models

Stripmining

Long vectors should be **split to parts** of sizes less or equal to the **maximum vector length** allowed by vector registers and possibly other hardware components. Consequently, **dependence of computer speedup on vector length** is a **saw-like curve** as depicted called **stripmining**

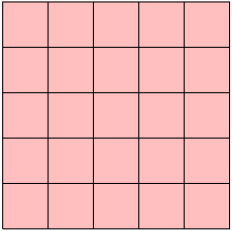


Splitting long vectors for Cyber-205 (late 70's; memory-memory vector processor) scheduled by an efficient microcode software. Since Cray X-MP this is done by hardware.

Vector processor and SIMD models

Stride

- Processing vectors with a non-unit distance among entries in memory.
- If this distance is regular, it is called **stride** and vector processor does not need to be always efficient in processing vectors with strides > 1.
- BLAS routines can deal with various strides (but there is a price for it).
- An example: a column in the following matrix stored by rows can be obtained by getting its entries with the stride 5



Vector processor and SIMD models

Vectorization examples

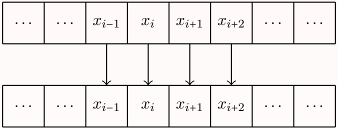
Vector norm

Algorithm (Computing (squared) norm $x^T x$ for $x \in R^n$.)

Input: *Vector* x.

Output: *Resulting squared norm.*

```
for $i = 1, \ldots, n$ do
    $x_i = x_i * x_i$
end for
```



- No dependence among the vector components.
- **Automatic vectorization**.

Vector processor and SIMD models

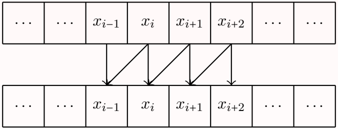
Vectorization examples

Algorithm (Product with forward shift $x_{1:n-1} = x_{1:n-1}^T x_{2:n}$.)

Input: *Vector* x.

Output: $x_{1:n-1} = x_{1:n-1}^T x_{2:n}$.

```
for $i = 1, \dots, n-1$ step $1$ do
    $x_i = x_i * x_{i+1}$
end for
```



- The loop vectorizes as well.

Vector processor and SIMD models

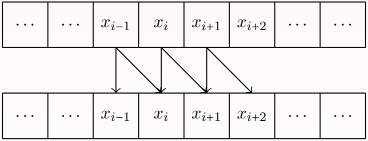
Vectorization examples

Algorithm (Product with backward shift.)

Input: *Vector x.*

Output: *See below*

```
for $i = 2, \ldots, n$ step $1$ do
    $x_i = x_i * x_{i-1}$
end for
```



- The loop does not vectorize: $x_i = \prod_{j=1}^{i} x_j$. (different than above).

Vectorization examples

Product of all vector components (continued)

Algorithm (**Reversing order of processing.**)

Input: *Vector x.*

Output: *See below*

```
for $i = n, \ldots, 2$ step $-1$ do
    $x_i = x_i * x_{i-1}$
end for
```

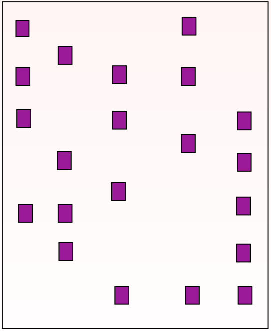
- But: the loop may produce a **different result**.
- Consequently, we should be careful.

Vector processor and SIMD models

Vectorization examples

Vector processing of sparse (indirectly addressed) vectors (continued)

- Sparse vectors can be parts (as rows or columns) of sparse matrices.
- **Enormous influence** on sparse algorithms and implementations.



Vector processor and SIMD models

Vectorization examples

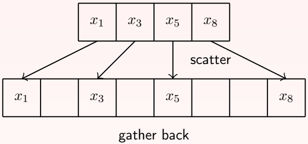
Vector processing of sparse (indirectly addressed) vectors (continued)

- Splitting the process into three parts
  - **scattering** dense vectors (indirectly addressed - this is how the sparse vectors are stored) into sparse ones
  - **computation** with dense vectors
  - **gathering** the result

Vector processor and SIMD models

Vectorization examples

Vector processing of **sparse (indirectly addressed) vectors**



- A breakthrough enabling this: **hardware/software** support of vectorizing sparse (indirectly addressed) data: vectorized **gather** and **scatter**
  - Cray X-MP/4, Cray X-MP/4, commodity stuff like AVX-512, ARM, InfiniBand; otherwise: prefetch should cover this
- Still slower than directly addressed vectors.

Vector processor and SIMD models

Vectorization examples

Algorithm (**Scatter** $x \in R^k$ into $y \in R^n$ using **selected indices** $mask(1:k), k < n$.)

Input: *Vector* $x \in R^k$.

Output: *See below*

```
for $i = 1, \ldots, k$ do
    $y(mask(i)) = x(i)$
end for
```

Algorithm (**Gather** a vector $y \in R^n$ into a vector $x \in R^k, k < n$.)

Input: *Vector* $y \in R^n$.

Output: *See below*

```
for $k$ selected indices $mask(1:k)$ from $i = 1, \ldots, n$ do
    $x(i) = y(mask(i))$
end for
```

Vector processor and SIMD models

More complex examples of vectorization

- Typically **unknown length** of the data pipeline, possible **chaining**.
- Also, formulas hidden inside a function f
- But, if the shift is known ... (nowadays NOT REALISTIC)
- E.g., in lagged Fibonacci sequence to get random sequences as

$$x_n = x_{n-a} + x_{n-b} \; mod \; m, \;\; 0 < a < b.$$

- **Vector lengths** (pipeline lengths) then at most $\min(a, b)$.

Algorithm

Constrained vectorization: *shift in the constrains*

```
1. for $i = 1, \ldots, n$ step $1$ do
2.         $x_i = f(x_{i-k})$
3. end $i$
```

Vector processor and SIMD models

More complex examples of vectorization

Algorithm

Similar loop (with an additional vector y): *easily vectorized*

```
1. for $i = 1, \ldots, n$ step $1$ do
2.         $\mathbf{y}_i = f(x_{i-k})$
3. end $i$
```

Vector processor and SIMD models

More complex examples of vectorization

Wheel method

- Another early (outdated) approach.
- ns **stages (segments)** of a pipeline: have to be known.
- Often not realistic (chaining, simply not known)

Algorithm

Wheel method *for the operation:* $sum = \sum_j a_j,\ j = 1, \dots, n$

```
1. for $i = 1, \dots, ns$ do
2.     for $k = 1, \dots, n/ns$ do
3.         $x_i = \sum_k a_{i+ns*k}$
4.     end $k$
5. end $i$
6. $sum = \sum_{i=1}^{ns} x_i$
```

Vector processor and SIMD models

More complex examples of vectorization

Loop unrolling

Consider the following AXPY operation that may correspond to a loop inside a computational code.

Algorithm

(S,D)AXPY operation.

```
**for** $i = 1, \dots, n$ **do**
    $y(i) = y(i) + \alpha * x(i)$
**end** $i$
```

- Loop unrolling algorithm
- Nowadays **often** automatic

Vector processor and SIMD models

Loop unrolling (continued)

Algorithm

4-*fold loop unrolling with an integer increment of indices* $incx$

```
for $i = 1, \dots, n$ step $5$ do
    $y(i) = \alpha\, x(i)$
    $y(i + incx) = \alpha\, x(i + incx)$
    $y(i + 2 * incx) = \alpha\, x(i + 2 * incx)$
    $y(i + 3 * incx) = \alpha\, x(i + 3 * incx)$
    $y(i + 4 * incx) = \alpha\, x(i + 4 * incx)$
end $i$
```

Vector processor and SIMD models

Loop unrolling (continued)

- Some other examples

Algorithm

A computational segment with two nested loops

```
for $i = 1, \ldots, n$ do
    for $j = 1, \ldots, n$ do
        $a(j,i) = \alpha\, b(j,i) + \beta\, c(j)$
    end $j$
end $i$
```

- Vectorization depends on the way the matrices are stored

Vector processor and SIMD models

Loop unrolling (continued)

Algorithm

A computational segment with 2-fold unrolling of the outer loop

```
for $i = 1, \ldots, n$ step $3$ do
    for $j = 1, \ldots, n$ do
        $a(j,i) = \alpha\, b(j,i) + \beta\, c(j)$
        $a(j,i+1) = \alpha\, b(j,i+1) + \beta\, c(j)$
        $a(j,i+2) = \alpha\, b(j,i+2) + \beta\, c(j)$
    end $j$
end $i$
```

Vector processor and SIMD models

Loop fusion

Algorithm

A computational segment with two loops that can be fused.

```
**for** $i = 1, \ldots, n$ **do**
    $y(i) = y(i) + \alpha\, x(i)$
**end** $i$
**for** $j = 1, \ldots, n$ **do**
    $u(j) = u(j) + \beta\, x(j)$
**end** $j$
```

- Typically during optimization by the compiler. But loops may include more complex objects.

Vector processor and SIMD models

Loop fusion (continued)

The two loops in this program segment can be fused together as follows.

Algorithm

A computational segment with two loops that were fused.

```
1. for $i = 1, \ldots, n$ do
2.     $y(i) = y(i) + \alpha\, x(i)$
3.     $u(i) = u(i) + \beta\, x(i)$
4. end $i$
```

Clearly, the loop fusion reduces the number of memory accesses due to reuse of the values $x(i)$, $i = 1, \ldots, n$.

Vector processor and SIMD models

Associative transformations

- Exploiting associativity in the dot product of two vectors.

Algorithm (Dot product s of two vectors.)

Input: *Vectors x and y.*
Output: *Dot product of the input vectors.*

```
$s = 0$
for $i = 1, \ldots, n$ do
    $s = s + x(i)\, y(i)$
end for
```

Should be rewritten as follows (if not done automatically).

Vector processor and SIMD models

More complex examples of vectorization

Algorithm

Transformed dot product s of two vectors.

```
$s_1 = 0$
$s_2 = 0$
**for** $i = 1, \ldots, n$ **step** $2$ **do**
    $s_1 = s_1 + x(i)\,y(i)$
    $s_2 = s_2 + x(i+1)\,y(i+1)$
**end** $i$
$s = s_1 + s_2$
```

- Similar schemes discussed later
- Logarithmic (complexity - number of steps) curse

Vector processor and SIMD models

Vector processor model and linear algebra codes

- The idea of **temporal locality** can be formally characterized by the ratio q defined as follows. (data reused in closely after should be kept inside cache)

$$q = \frac{\text{flops counts}}{\text{number of memory accesses}} \tag{18}$$

- The effort to increase the fraction q: driving force to continue in development of BLAS **outside BLAS1**. Let us first show q for some typical **representative operations**. Here we consider $\alpha \in R$, $x, y, z \in R^n$, $A, B, C, D \in R^{n \times n}$

operation	**op**eration count	amount of **comm**unication	$q = op/comms$
$z = \alpha x + y$	$2 * n$	$3 * n + 1$	$\approx 2/3$
$z = \alpha Ax + y$	$2 * n^2 + n$	$n^2 + 3 * n + 1$	≈ 2
$D = \alpha AB + C$	$2 * n^3 + n^2$	$4 * n^2 + 1$	$\approx n/2$

Vector processor and SIMD models

Vector processor model and linear algebra codes

- BLAS2 (1988): **better use of vector machines**. It includes operations as the matrix-vector product $z = \alpha Ax + y$, rank-1 updates and rank-2 updates of matrices, triangular solves and many other operations.
- BLAS3 (1990): **better use of computer architectures with caches** It covers, e.g., GEMM $(D = AB + C)$.
- As with BLAS1, stress put to machine-specific efficient implementations. All BLAS subroutines: standardized as procedures in high-level languages and as **calls** to machine-dependent libraries on different architectures.
- Possible BLAS **cons**: sometimes **time-consuming interface** for simple operations.

Vector processor and SIMD models

Standardization at a higher level: LAPACK

The set of subroutines called LAPACK covers many solving such problems related to dense and/or banded matrices as

- Solving systems of linear equations
- Solving eigenvalue problems
- Solving least-squares solutions of overdetermined systems
- The actual solvers are based, for example, on the associated factorizations like **LU, Cholesky, QR, SVD, Schur factorization** completed by many additional routines used, e.g., to **estimation of condition numbers**, **reorderings by pivoting**. The whole package is based on earlier LINPACK (1979) and EISPACK (1976) projects that provided also computational core of the early Matlab.
- A schematic example of using smaller blocks that fit the computer cache in LU and QR factorizations follows.

Vector processor and SIMD models

Standardization at a higher level: LAPACK

Blocks in LU decomposition

- LU decomposition → Block LU decomposition:

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix} = \begin{pmatrix} L_{11} & & \\ L_{21} & L_{22} & \\ L_{31} & L_{32} & L_{33} \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} & U_{13} \\ & U_{22} & U_{23} \\ & & U_{33} \end{pmatrix}$$

- LU solve step: **substitutions** with diagonal blocks, **multiplications** by off-diagonal blocks

Standardization at a higher level: LAPACK

LAPACK: blocks in QR decomposition

- $$A = Q\begin{pmatrix} R \\ 0 \end{pmatrix}$$
- Householder's method is based on the reflection matrices of the form

$$P = I - \alpha u u^T \text{ with } \alpha u^T u = 2 \tag{19}$$

In the k-th step we get

$$Q_k^T A = \begin{pmatrix} R_k & S_k \\ & A_k \end{pmatrix}, \; Q_k = Q_{k-1}\begin{pmatrix} I & \\ & I - \alpha_k \tilde{u}_k \tilde{u}_k^T \end{pmatrix} = Q_{k-1}(I - \alpha_k u_k u_k^T). \tag{20}$$

BLAS3 QR factorization is based on a **matrix representation** of the product of the transforms. Consider k of them.

$$\prod_{j=1}^{k} (I - \alpha_j u_j u_j^T) = I - YTY^T \tag{21}$$

Vector processor and SIMD models

Standardization at a higher level: LAPACK

Blocks in QR decomposition (continued)

$$Y = (u_1, \ldots, u_k) \in R^{n\times k},\ T \in R^{k\times k} \text{ upper triangular.} \tag{22}$$

Then for $u = u_{k+1}$

$$\begin{aligned}
(I - YTY^T)(I - \alpha uu^T) &= I - \alpha uu^T - YTY^T + \alpha YTY^T uu^T \\
&= I - \begin{pmatrix} Y & u \end{pmatrix} \begin{pmatrix} TY^T - \alpha TY^T uu^T \\ \alpha u^T \end{pmatrix} \\
&= I - \begin{pmatrix} Y & u \end{pmatrix} \begin{pmatrix} T & -\alpha TY^T u \\ & \alpha \end{pmatrix} \begin{pmatrix} Y^T \\ u^T \end{pmatrix} \\
&= I - (Y, u) \begin{pmatrix} T & h \\ & \alpha \end{pmatrix} (Y, u)^T
\end{aligned}$$

- Even more compact WY form (instead of YTY^T form) possible.

Vector processor and SIMD models

BLAS3 in practice: Matrix-matrix multiplications and cache

- Assume a **fast memory (cache)** of the size $M \geq n$
- Three example cases

Case 1: M can store a row of a square matrix.

$$M \approx n. \tag{23}$$

$$\Downarrow$$

Standard dot product of A_{i*} and B_{*j} in the innermost loop.

Vector processor and SIMD models

BLAS3 in practice: Matrix-matrix multiplications and cache

Case 1: M can store a row of a square matrix (continued).

Algorithm

Standard dense matrix-matrix multiplication

Input: *Matrices* $A, B, C \in R^{n \times n}$

Output: *Product* $C = C + AB$.

```
for $i = 1, \ldots, n$ do
        for $j = 1, \ldots, n$ do
            for $k = 1, \ldots, n$ do
                $C_{ij} = C_{ij} + A_{ik} B_{kj}$
        end $k$
    end $j$
end $i$
```

Vector processor and SIMD models

BLAS3 in practice: Matrix-matrix multiplications and cache

Case 1: M can store a row of a square matrix (continued).

- Assume that M stores a row of A
- Communication: n^2 for A (input just once), $2n^2$ for C (load and store), n^3 for B (B is read **for each** row of A)
- operations: $2n^3$ (we count both additions and multiplications)
- summary: $q = ops/refs = 2n^3/(n^3 + 3n^2) \approx 2$

Consequently, the algorithm is as **slow as BLAS2.**

Vector processor and SIMD models

Case 2: $M \approx n + 2n^2/N$ **for some** N.

- M is slightly larger than n but still **not large enough**
- B and C into N split into column blocks of size n/N.

$$C = [C^{(1)}, \dots, C^{(N)}], B = [B^{(1)}, \dots, B^{(N)}].$$

- Example for 3×3 block matrices.

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} & B_{13} \\ B_{21} & B_{22} & B_{23} \\ B_{31} & B_{32} & B_{33} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} & C_{13} \\ C_{21} & C_{22} & C_{23} \\ C_{31} & C_{32} & C_{33} \end{pmatrix}$$

$$\begin{pmatrix} C_{11} \\ C_{21} \\ C_{31} \end{pmatrix} = \begin{pmatrix} a_{11} \\ a_{21} \\ a_{31} \end{pmatrix} B_{11} + \begin{pmatrix} a_{12} \\ a_{22} \\ a_{32} \end{pmatrix} B_{21} + \begin{pmatrix} a_{13} \\ a_{23} \\ a_{33} \end{pmatrix} B_{31}.$$

- The whole A, one block of B (by rows), one block of C (by outer products)

Vector processor and SIMD models

Case 2: $M \approx n + 2n^2/N$ **for some** N **(continued).**

Algorithm

Dense matrix-matrix multiplication $C = C + AB$ *with* N *column blocks of size* n/N *in* C *and* B*.*

Input: *Matrices* $A, B, C \in R^{n\times n}$ *with blocks*

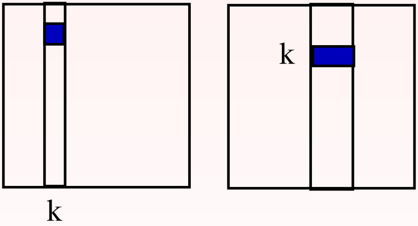
$$C = [C^{(1)}, \dots, C^{(N)}],\ B = [B^{(1)}, \dots, B^{(N)}]$$

Output: *Product* $C = C + AB$*.*

```
1. for $j = 1, \dots, N$ do
2.     for $k = 1, \dots, n$ do
3.         $C^{(j)} = C^{(j)} + A_{*k} B^{(j)}(k,*)$
4.     end $k$
5. end $j$
```

Vector processor and SIMD models

Case 2: $M \approx n + 2n^2/N$ **for some** N **(continued).**



Vector processor and SIMD models

Case 2: $M \approx n + 2n^2/N$ for some N (continued).

- assuming $M \approx n^2/N + n^2/N + n = 2n^2/N + n$ (block of C, block of B, column of A)
- That is $M \approx 2n^2/N$.
- communication: read+write C: $2n^2$, read B sequentially by blocks: n^2, read A N-times: Nn^2
- $q = 2n^3/(3+N)n^2 \approx MNn/(3+N)n^2 \approx M/n$

Vector processor and SIMD models

Case 3: $M \approx 3(n/N)^2$ **for some** N**.**

Let us consider row and column blocks $A^{(ij)}, B^{(ij)}, C^{(ij)}$ at a grid $n/N \times n/N$ and the matrix-matrix multiplication based on the following algorithm.

Algorithm

Dense matrix-matrix multiplication $C = C + AB$

Input: *Matrices* $A, B, C \in R^{n \times n}$ *with the two-dimensional grid of blocks*

Output: *Product* $C = C + AB$.

```
for $i = 1, \ldots, N$ do
        for $j = 1, \ldots, N$ do
            for $k = 1, \ldots, N$ do
                $C^{(ij)} = C^{(ij)} + A^{(ik)} B^{(kj)}$
        end $k$
    end $j$
end $i$
```

Vector processor and SIMD models

Case 3: $M \approx 3(n/N)^2$ **for some** N **(continued).**

- Assuming $M \approx 3(n/N)^2$
- This gives $n/N \approx M/3$.
- Communication: $2n^2$ for C, Nn^2 for A and B
- $q = 2n^3/(n^2(2+2N)) \approx n/(1+N) \approx \sqrt{M/3}$
- much better

Multiprocessor model

Multiprocessor model

Multiprocessor model

- The most general computational model.
- As in previous models, there are general concepts one should take into account when porting computations to this model.
- Here, more extensively, only **parallel matrix** computations considered.
- Parallel computations need **new algorithms**.
- Algorithms must be very often **new** and **not** only **straightforward parallelizations** of serial algorithms.

Multiprocessor model

REPETITION

- **Multiprocessor** computational having **under one operating (control) system a possibility of independent concurrent computations**.
- Many possible and very different architectures.
- Uniprocessor: interested mainly in **latency, bandwidth** for the **processor-memory** relation, locality, regularity.
- Vector model: in addition **good and bad (data) pipelining/vectorization**, connection to BLAS, LAPACK, cache and matrix-matrix operations.
- Multiprocessor: In addition: more stress to **regularity** of computations, **communication** (processor-processor (core-core, etc.), but also granularity, decomposition, load balancing, programming patterns.

Multiprocessor model

Hardware **considerations**

- **shared** / **partially shared** or **fully distributed**.
- **memory access** often non-uniform **(NUMA)**
- **uniform access to memory** sometimes supported by techniques like COMA (cache-only memory architectures)
- many **cache-specific items** related to multiprocessors to be solved
- **remote access latencies** (if data for a processor are updated in a cache of another processor and not yet in the main memory).
- **general regularity principles** plus **problem division regularity**
- codes on multiprocessors: difficult to prefetch

Multiprocessor model

Algorithmic considerations

- algorithmic/programming patterns
- programming tools
- algorithm/code features
  - granularity
  - decomposition
  - load balancing

Multiprocessor model

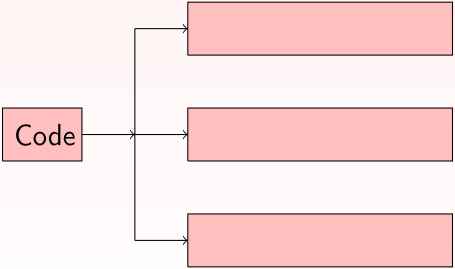
Multiprocessor programming patterns

- Parallel programs:
    - Collection of tasks executed by **processes** or **threads** on multiple computational units.
    - Must be compatible with the decomposition schemes
    - We will mention first programming patterns
    - The some tools to implement them will be commented on

Multiprocessor model

Multiprocessor programming patterns (continued)

- 1. SPMD/SIMD
  - **Fixed number of threads/ processes** to process different data as their acronyms state.



Multiprocessor model

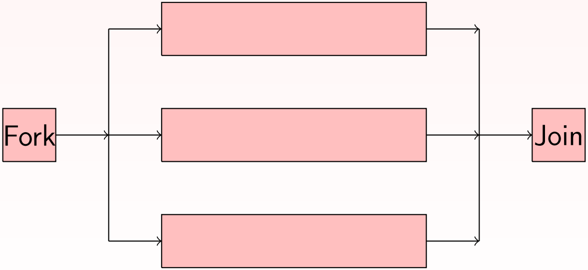
Multiprocessor programming patterns (continued)

- 1. SPMD/SIMD
  - **SIMD** Based (often) on **data pipelining**: performing operations synchronously.
    - ★ **GPU** processing may cover more SIMD streams
  - **SPMD** (single program, multiple data streams)
    - ★ typically connected with asynchronous work, on different CPUs
    - ★ the independent tasks in a single program/code,
    - ★ possibly some synchronization points.
    - ★ processes or threads have **equal rights**.

Multiprocessor model

Multiprocessor programming patterns

- Parallel programs:
  - Collection of tasks executed by **processes** or **threads** on multiple computational units.
  - Must be compatible with the decomposition schemes
- **2. Fork-join** constructs:



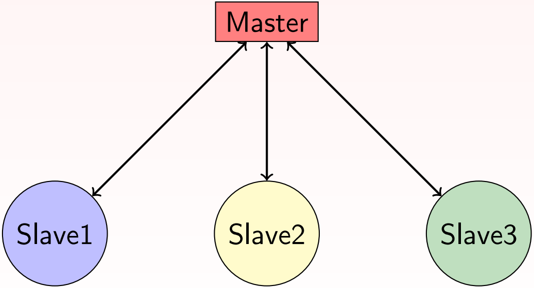
Multiprocessor model

2. Fork-join

- Process or thread creates a set of **child processes or threads** that work in parallel. This is so-called **fork**.
- The **parent** process then **waits** until all the child subtasks are done using the **join** statement.
- The parent process can either wait, perform different tasks or perform one of the subtasks as well.
- Another name: **spawn-exit**.
- In **programming languages:** this can be easily coded, for example, as **parbegin/parend** or **cobegin/coend**.

Multiprocessor programming patterns (continued)

- **3. Master-slave**
  - **One master** that controls the execution of other processes / threads.
  - Master can take tasks from a pool (set of block rows, matrices, etc.)



Multiprocessor model

Multiprocessor programming patterns (continued)

- **4. MPMD**
  General **MPMD style** uses multiple programs as well multiple data streams.
  - **Balancing** the code with available general communication patterns needed.
  - An example is a structured **client/server** model. where a specific non-parallelizable tasks (as, possibly parts of input and output can be) are processed at a designated master processing unit. In contrast to the master/slave model where the master controls, here the clients communicate with the server in a more general way.
  - General MPMD pattern processors may have different roles and may be interconnected by various ways.

Multiprocessor model

Multiprocessor programming tools

- To follow the multiprogramming patterns we have much less. Moreover, we depend on **available hardware** and concepts the keep evolving. Most common tools (from the point of view of an application, not wishing to be devoured by short-life concepts.
- MPI: Standard and very flexible library. Specific instructions from this library should be embedded in application codes. Very extensive set of instructions, but for basic use, not many of them are needed.
    - An easy treatment on **more distributed** computer architectures
    - an example: **MPSD** (related to MISD mentioned above) treatment.
- OpenMP: Application programming interface (API) on the side of compiler. Able to create internally a set of (safe) threads. No external libraries needed, dependence on compiler, parallel sections of code easily defined. But, a notion of shared-memory programming behind.
- Using threads that are a specific low-level tool on shared-memory computers, their control is often a part of the operating system

Multiprocessor model

Granularity:

- Relates to average **sizes of code chunks** that processed/communicated concurrently.
- also: ratio: amount of computation / amount of communication (time for computation / time for communication)
  - coarse (example: submatrices)
  - medium (example: rows, columns)
  - fine (example: individual values)
- Decisions on granularity always close to **choice of algorithms**
- Classification is problem dependent.

Multiprocessor model

Problem decomposition

- **Problem decomposition** is the way to divide problem processing among individual computational units (how to achieve intended granularity).
  - **task-based**
  - **data-based**
- **Load balancing** denotes then specifically the strategies and/or techniques to minimize T_{par} on multiprocessors by approximate equalizing **workload/worktime** tasks for individual computational units and possibly also minimizing the **synchronization overheads**.
  - **static** problem decomposition and load balancing
  - **dynamic** problem decomposition and load balancing
- The consequent assignment of the divided parts to computational units via **processes** or **threads** is called **mapping**.

Multiprocessor model

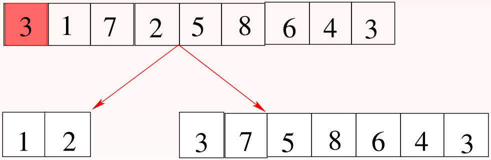
Problem decomposition: task-based

- 1. Recursive/hierarchical decomposition:
  - Dividing the problem into a set of independent subproblems
  - The same strategy applied to the subproblems **recursively**.
  - The full solution assembled from the partial solutions of the subproblems.
  - This type of decomposition is typically connected to algorithms that use the **divide and conquer** strategy.
  - An example: the sorting algorithm **quicksort**.

Multiprocessor model

Problem decomposition: task-based (continued)

- 1. Recursive/hierarchical decomposition:



3 1 7 2 5 8 6 4 3

1 2

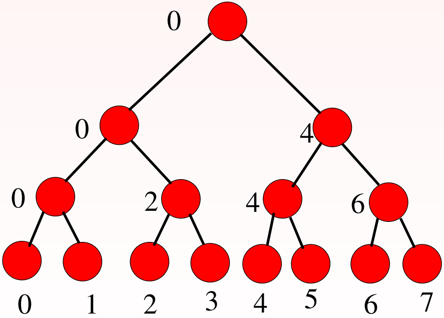
3 7 5 8 6 4 3

Multiprocessor model

Problem decomposition: task-based (continued)

- 1. Recursive/hierarchical decomposition:

Regular hierarchical decomposition uses a mapping of the subtasks to a binary tree with 8 processing units is depicted below.



Multiprocessor model

Problem decomposition: task-based (continued)

- 2. Exploratory decomposition
- Splits the search space using results of previous steps.
- Performed **hand-in-hand** with execution.
- An example: **15-puzzle problem**. The goal is to find a path from the initial configuration of a 4×4 grid into the final configuration by moves of a tile into an empty position.

Multiprocessor model

Problem decomposition: task-based (continued)

- 3. Speculative decomposition:
- specific variation of exploratory decomposition that not only uses results of the previous steps but also speculates on their results.
- Used when program flow depends on results of branch instructions.

Multiprocessor model

Problem decomposition: data-based

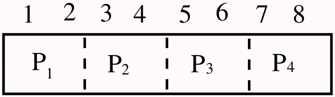
- **Available data sets** are shared, decomposed and balanced among the computational units.
- We distinguish separately **static data decomposition** and **dynamic data decomposition**.
- As for the load balancing, standard strategy in the former case is to use **static load balancing** while in the latter case the load should be balanced dynamically.
- Needed to distinguish **input** decomposition, **intermediate** decomposition and **output** decomposition.

Multiprocessor model

Problem decomposition: data-based (continued)

A. Static data decomposition (continued)

- 1D arrays, 1D block data distribution
- Each process owns block of $1 \le b\lceil n/p \rceil$ entries.
- Picture for $n = 8$, $p = 2$, $b = 2$ depicted

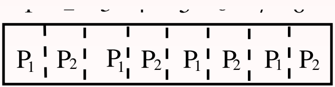


Multiprocessor model

Problem decomposition: data-based (continued)

A. Static data decomposition (continued)

1D arrays, 1D cyclic data distribution
For the same 1D array an entry v_i, $i = 1, \ldots, n$ of the array is assigned to the process $P_{(i-1)\ mod\, p+1}$. Here $p = 2$, $n = 8$.



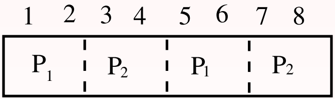
P_1	P_2	P_1	P_2	P_1	P_2	P_1	P_2

Multiprocessor model

Problem decomposition: data-based (continued)

A. Static data decomposition (continued)

1D arrays, 1D block cyclic data distribution
This is a combination of the block and cyclic distribution as shown in the figure.

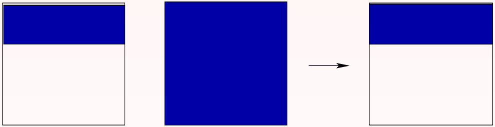


Multiprocessor model

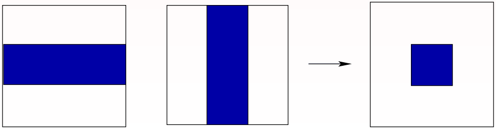
Problem decomposition: data-based (continued)

A. Static data decomposition (continued)

- 2D arrays, 1D block data distribution
- —- By rows, columns, block rows, block columns.
- 2D arrays, 2D block data distribution
- —- blocks of size $n/\sqrt{p} \times n/\sqrt{p}$, both by rows and columns



1D partitioning

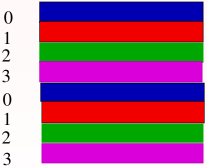


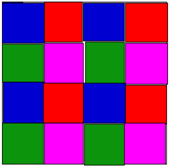
Multiprocessor model

Problem decomposition: data-based (continued)

A. Static data decomposition (continued)

- 2D arrays, 1D cyclic data distribution
- —- The same, but cyclically. Row cyclic distribution is at the figure below.
- 2D arrays, 2D block cyclic array (2D) distribution.





Multiprocessor model

Problem decomposition: data-based (continued)

A. Static data decomposition (continued)

- **Other static data decompositions**
  - **Randomized block decompositions**
  - Various types of **hybrid decompositions**.
  - Specifically, for sparse matrices: traditionally called graph / hypergraph partitioning will be **mentioned later**.

- The schemes can be modified by **weights** that take into account specific architectural considerations
- In any case, decomposition should be **balanced** with the algorithm.

Multiprocessor model

Problem decomposition: data-based

B. Dynamic data decomposition

- Usually closely connected to programming models.
- In centralized (master/slave) schemes, one special process (computational unit) manages a pool of available tasks. **Slave** processors/processes then choose and perform tasks taken from the pool. Can be modified/improved by various **scheduling strategies**:
  - **self-scheduling** (choosing tasks by independent demands),
  - **controlled-scheduling** (master involved in providing tasks) or
  - **chunk-scheduling** where the slaves take a block of tasks to process.
- Fully distributed dynamic scheduling within **non-centralized** processing schemes
  - Nontrivial synchronization

Multiprocessor model

Linear algebra standardization and multiprocessor model

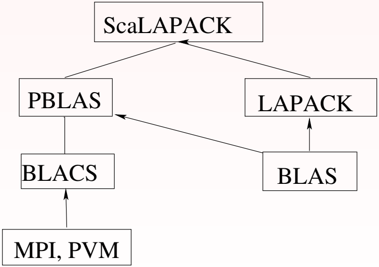
Multiprocessing model influenced development of basic linear algebra subroutines in two basic directions.

- Standardization of communication
- Development of LA libraries on the top of the communication paradigms.
- **BLACS**
  Covers low level of concurrent programming, creates standardized interface on the top of message passing layers like MPI (message passing interface) or PVM (parallel virtual machine).
- **PBLAS** Parallel BLAS called **PBLAS** represents an implementation of BLAS2 and BLAS3 for distributed memory architectural model.

Multiprocessor model

Linear algebra standardization and multiprocessor model

- **ScaLAPACK**
  The standardized library of high-performance linear algebra for message passing architectures. Its basic linear subroutines heavily rely on PBLAS. The following figure shows schematically the dependencies among linear algebra high-performance software that target distributed memory architectures.



Multiprocessor model

Linear algebra standardization and multiprocessor model

Next development came with the advent of more involved multiprocessors

- **Multi-core** processors: computing component with a small number of independent processing units ("cores").
- **Manycore** processors: specialized multi-core processors designed to get a high degree of parallel processing. They typically contain a large number of simpler, independent processor cores (e.g. 10s, 100s, or 1,000s). Various tricks to achieve low level of cache coherency.

Forcing the concepts like

- Massive **fork-join parallelism**
- Nesting the fork and join can be efficiently implemented **divide-and-conquer** strategies.
- Use of **tiling** based on reordering matrix data into **smaller regions of contiguous memory.**
- **Tile algorithms** allow fine granularity parallelism and asynchronous dynamic scheduling.

Multiprocessor model

Linear algebra standardization and multiprocessor model

- **PLASMA**
  A high level linear algebra library for parallel processing that takes into account multicore computer architectures and forms a counterpart of the (part of) high level libraries LAPACK and ScaLAPACK is called **PLASMA**. Apart from new algorithms, as "communication avoiding" QR factorization, the approach considers concepts of **tile layout** of the processed matrices and dataflow scheduling using the fork-join concept.
- **MAGMA**
  Going to manycore processors has significantly increased heterogenity of computer architectures. Hybrid linear algebra algorithms: **MAGMA**. Among its important algebraical features: **varying granularity** based on a strong task scheduler with a possibility to schedule statically or dynamically.

Outline

1. Foreword
2. Computers, computing, communication
3. Parallel computing
4. Parallel processing and us - parallel programming
5. Parallel computer architectures: hardware and classification
6. Combining pieces together: computational models
   - Uniprocessor model
   - Vector and SIMD model
   - Multiprocessor model
7. Parallelizing problems
8. Sparse data decomposition: graph partitioning
9. Parallel and parallelized algebraic preconditioning

Straightforward fine grain parallelism

1. Pointwise Jacobi iterations in 2D grids

- Poisson equation in two dimensions $A \in R^{n \times n}$ with Dirichlet boundary conditions and its standard (two-dimensional) five-point discretization on a uniform $\sqrt{n} \times \sqrt{n}$ grid.

$$\begin{aligned} -\Delta u = & \quad f \ in \ \Omega \\ u = & \ \ 0 \ at \ \delta\Omega \end{aligned} \tag{24}$$

- Initial distribution to a 2D $\sqrt{n} \times \sqrt{n}$ grid of processors

$$A = \begin{pmatrix} B & -I & & \\ -I & B & -I & \\ & \dots & \dots & \\ & & -I & B \end{pmatrix}, \quad B = \begin{pmatrix} 4 & -1 & & \\ -1 & 4 & -1 & \\ & \dots & \dots & \\ & & -1 & 4 \end{pmatrix} \tag{25}$$

Straightforward fine grain parallelism

1. Pointwise Jacobi iterations in 2D grids (continued)

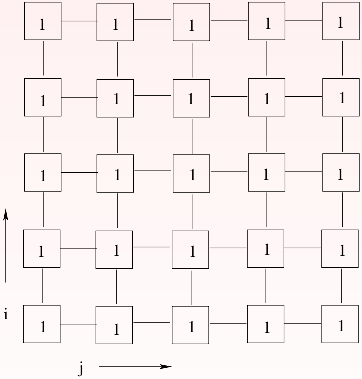
- The Jacobi iterations that use vectors b, x of compatible dimensions are given by

$$x^{+} = (I - D^{-1}A)x + D^{-1}b, D = \begin{pmatrix} 4 & & & \\ & 4 & & \\ & & \dots & \\ & & & 4 \end{pmatrix} \tag{26}$$

- $$x_{ij}^{+} = x_{ij} + (b_{ij} + x_{i-1,j} + x_{i,j-1} + x_{i+1,j} + x_{i,j+1} - 4 * x_{ij})/4$$

Straightforward fine grain parallelism

1. Pointwise Jacobi iterations in 2D grids



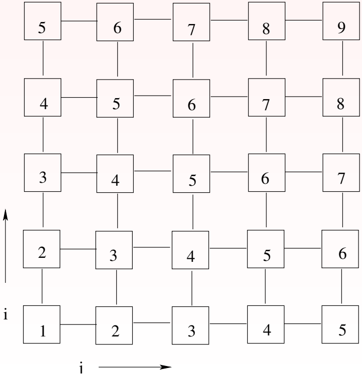
- Processors – gridpoints

Straightforward fine grain parallelism

2. Pointwise Gauss-Seidel iterations in 2D grids

- $$x^+ = (I - (D - L)^{-1}A)x + (D - L)^{-1}b$$

- $$x^+_{ij} = x_{ij} + (b_{ij} + x^+_{i-1,j} + x^+_{i,j-1} + x_{i+1,j} + x_{i,j+1} - 4 * x_{ij})/4$$



Parallelizing program branches

3. Parallelizing branches

- Branches appear very frequently in all application codes. They often prohibit efficient parallelization.
- Case 1: Both branches executed if evaluated cheaply

Algorithm

Both branches are executed *if both f and g can be evaluated cheaply*

```
for $i = 1, \dots, n$ step $1$ do
    if $e(a_i) > 0$ then
        $c_i = f(a_i)$
    else
        $c_i = g(a_i)$
    end if
    the code continues using $c_i$
end $i$
```

Parallelizing program branches

3. Parallelizing branches

- Case 2: At least one of the code branches expensive: "gather/scatter"

Algorithm

```
inda=1, indb=1
for $i = 1, \ldots, n$ step $1$ do
        if $e(a_i) > 0$ then
            $ja(inda) = i$
            $inda = inda + 1$
        else
            $jb(indb) = i$
            $indb = indb + 1$
        end if
end $i$
perform operations as they are indirectly addressed in $ja, jb$
```

Parallel operations with dense vectors and matrices

An attempt to mix and measure computation and communication

- **Basic** models of parallel computation with vectors and matrices discussed.
- Different variations of **data decomposition** considered.
- **Computation and communication operations** and asymptotic behavior of their combination discussed.
- **An assumption on communication needed.**
- Description combines both the explicit notation of timings as well as standard $\Theta(.)$ and $O(.)$ notation. notation.

Parallel operations with dense vectors and matrices (continued)

Measuring communication and computation

- **Parallel time** T_{par} is proportional to the number of parallel steps
- **Process time** or **work time** denote by T_{pr} we will define by

$$T_{pr} = \Theta(p\ T_{par}). \tag{27}$$

- We say that an algorithm is **cost-optimal** if the process time is proportional to the sequential time, that is, if

$$T_{pr} = \Theta(T_{seq}). \tag{28}$$

Parallel operations with dense vectors and matrices (continued)

Measuring communication and computation: an assumption

- **Tentative communication costs** for some basic communication operations needed. Of course, they can be very different for different computer **interconnects** and different **routing** and **switching**.
- For evaluations related to realistic coupling of computation and communication we need to use an **assumption** on the architectural details.
- Here we assume that the architecture embeds a **hypercube commmunication interconnect** with the hypercube having $p = 2^d$ nodes.

Parallel operations with dense vectors and matrices (continued)

Measuring communication and computation: one-to-all broadcast

- Hypercube interconnect results in the following parallel time

$$T_{par} = \min((T_{clatency} + mT_{word})\log_2 p, 2(T_{clatency}\log_2 p + mT_{word})), \tag{29}$$

  where $T_{clatency}$ denotes the communication latency.

- Simplified

$$T_{par} = (T_{clatency} + mT_{word})\log_2 p. \tag{30}$$

- The complexity assumes $\log_2 p$ simple point-point message transfers. This can be again influenced by switching and routing.
- Similar formulas for mesh or balanced binary tree interconnect.

Parallel operations with dense vectors and matrices (continued)

Measuring communication and computation: all-reduce

Similarly as in the previous case (one-to-all broadcast), the parallel time we use here is

$$T_{par} = T_{clatency} \log_2 p + mT_{word} \log_2 p. \tag{31}$$

Parallel operations with dense vectors and matrices (continued)

Measuring: all-to-all broadcast and all-to-all reduction

We assume the parallel time in the form

$$T_{par} = T_{clatency} \log_2 p + mT_{word}(p-1). \tag{32}$$

- The following term also in complexity for ring or mesh inteconnect.

$$mT_{word}(p-1) \tag{33}$$

- An idea how to see this: the term $O(m(p-1))$ can be considered as a lower bound for all multiprocessors that can communicate using only **one its link at a time** since each of the processors should receive $m(p-1)$ amount of data.

Parallel operations with dense vectors and matrices (continued)

Measuring: personalized reduction operations scatter and gather

$$T_{par} = T_{clatency} \log_2 p + mT_{word}(p-1). \tag{34}$$

Parallel operations with dense vectors and matrices (continued)

- The time T_{word} to transfer a word is an **indicator of the available bandwidth**.
- Used to count, e.g., number of transferred **numbers** that can be composed from more individual storage units called words.
- Note that reduction operations contain also some **arithmetic operations** (addition, maximum, etc.), overall a negligeable amount.
- Time T_{flop} needed for this is typically **dominated** by the (much faster) actual communication.
- Note that the formulas discussed below approach reality **only if** the computational tools are able efficiently balance **(typically slow) communication** timings with **(fast) flops**.

Parallel operations with dense vectors and matrices (continued)

AXPY operation

Consider the **AXPY** operation for $x, y \in R^n, \alpha \in R$ in the following notation.

$$y = \alpha x + y \tag{35}$$

- Assume the computation uses p processors
- 1D block decomposition
- Each processor owns a block of n/p numbers from both x and y.

$$\begin{aligned} \text{Sequential time} \quad &: \quad T_{seq} = 2n\ T_{flop} \\ \text{Parallel time} \quad &: \quad T_{par} = 2(n/p)\ T_{flop} \end{aligned}$$

- Consequently, the speedup is

$$S = T_{seq}/T_{par} = p$$

- Not very computationally intensive operation.

Parallel operations with dense vectors and matrices (continued)

Dot product

Consider a **dot product** $\alpha = x^T y$ for $x, y \in R^n$.

- As above, p processors, 1D decomposition

$$\begin{aligned}
\text{Seq time} \quad &: \quad T_{seq} = (2n-1)\ T_{flop} \approx 2n\ T_{flop} \\
\text{Par time} \quad &: \quad T_{par} \approx 2n/p\ T_{flop} + T_{reduce} \equiv 2n/p\ T_{flop} + (T_{clatency} + 1 \times T_{word}) \log_2 p
\end{aligned}$$

The speedup is

$$\begin{aligned}
S = T_{seq}/T_{par} \quad &\approx \quad \frac{2n\, T_{flop}}{2n/p\ T_{flop} + (T_{clatency} + T_{word}) \log_2 p} \\
&= \quad \frac{p\, T_{flop}}{T_{flop} + (T_{clatency} + T_{word}) p\ \log_2 p/(2n)} \\
&= \quad \frac{p}{1 + p\ \log_2 p/(2n)\ \times\ (T_{clatency} + T_{word})/T_{flop}} < p
\end{aligned}$$

Remind: typically $T_{clatency} >> T_{word}$ holds.

Parallel operations with dense vectors and matrices (continued)

Dense matrix-vector multiplication

The sequential multiplication with $T_{seq} = \Theta(n^2)$:

Algorithm

A simple scheme for dense matrix-vector multiplication $y = Ax$, $x, y \in R^n, A \in R^{n \times n}$.

```
for $i = 1, \ldots, n$ do
    Set $y_i = 0$
    for $j = 1, \ldots, n$ do
        $y_i = y_i + a_{ij} x_j$
    end $j$
end $i$
```

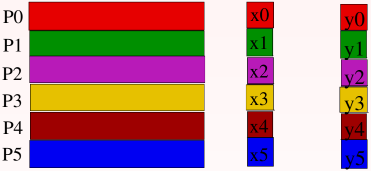
The parallel time and process time differ due to a chosen decomposition. We will discuss parallel computation using rowwise 1D and 2D decompositions and their block versions.

Parallel operations with dense vectors and matrices (continued)

4

Rowwise 1D partitioning: dense matvec

Distribution: One row is owned by one processor ($n = p$), each processor: one vector component. Similarly for the output vector. Schematically:



Algorithm

Parallel dense matrix-vector multiplication $y = Ax$, rowwise 1D partitioning

1. Broadcast vector x (all-to-all communication)

2. **do** *local row-column vector multiplication* **in parallel** *(keep the result distributed)*

Parallel operations with dense vectors and matrices (continued)

Rowwise 1D partitioning: dense matvec (continued)

$$\begin{aligned}
\text{Broadcast all} - \text{to} - \text{all time} &: T_{clatency}\log_2 n + T_{word}(n-1)\\
&: (all - to - all\ communication,\ message\ size: \quad m = 1)\\
\text{Multiplication time} &: (2n-1)\ T_{flop}\\
&: (local\ dot\ product\ between\ a\ row\ of\ A\ and\ x)\\
\text{Parallel time} &: T_{par} = (2n-1)T_{flop} + T_{clatency}\log_2 n + T_{word}(n-1)\\
\text{Process time} &: T_{pr} = \Theta(n^2)
\end{aligned}$$

Consequently, the matrix-vector multiplication is **cost optimal.**

Parallel operations with dense vectors and matrices (continued)

Block rowwise 1D partitioning: dense matvec $y = Ax$

Distribution: Less processors than rows. Each processor owns a block of n/p rows and a block component of x (vector with n/p components). Each processor then provides one vector block of y with n/p components.

Algorithm

Parallel dense matrix-vector multiplication (block rowwise 1D partitioning)

1. Broadcast vector x

(all-to-all communication among p processors; messages of size n/p broadcasted)

2. **do** *local block-row-column vector multiplication* **in parallel** *(keep the result distributed)*

Parallel operations with dense vectors and matrices (continued)

Block rowwise 1D partitioning: dense matvec $y = Ax$ (continued)

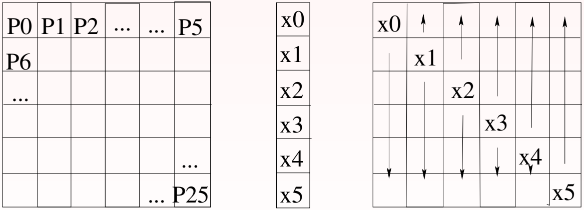
$$\begin{array}{rcl}
\text{Sequential time} & : & T_{seq} = \Theta(n^2) \\
\text{Comm time} & : & T_{clatency} \log_2 p + (n/p)\, T_{word}\, (p-1) \\
\text{Multiplication} & : & T_{flop}\ (2n-1)\, n/p \\
 & : & \text{(local dot products between a row block and block part of x)} \\
\text{Parallel time} & : & T_{par} = T_{flop}\ n(2n-1)/p + T_{clatency} \log_2 p + T_{word}\ (n/p)(p-1) \\
\text{Process time} & : & T_{pr} = T_{flop}\ n(2n-1) + T_{clatency}\ p \log_2 p + T_{word}\ n(p-1)
\end{array}$$

Consequently, this parallel matrix-vector multiplication is **cost optimal for** $p = O(n)$**.**

Parallel operations with dense vectors and matrices (continued)

2D partitioning: dense matvec

Distribution: Assume $p = n^2$ processors in a 2D mesh $n \times n$. Assume that the vector x is in the last processor column, or aligned along the diagonal. Schematically we have



Parallel operations with dense vectors and matrices (continued)

2D partitioning: dense matvec (continued)

Algorithm

Parallel dense matrix-vector multiplication (block 2D partitioning)

1. Initial **alignment** *of processors (standard initial phase)*
(one-to-one communication – it can be done within the initial data distribution)

2. **Vector distribution** *along processor columns - n parallel one-to-all broadcasts*

3. **Local scalar-multiplication**

4. Assembling the result at one processor of each row - n parallel **one-to-all reductions**

The alignment means that the vector components are **communicated to some processors that could redistribute** them along the columns. The processors that own the diagonal blocks often play this role.

Parallel operations with dense vectors and matrices (continued)

2D partitioning: dense matvec (continued)

$$
\begin{array}{rcl}
\text{Sequential time} & : & T_{seq} = \Theta(n^2) \\
\text{Alignment time} & : & \Theta(\log_2 n) \\
 & : & (\mathrm{one} - \mathrm{to} - \mathrm{one\ communication\ along\ a\ column}) \\
 & : & \text{sending data to “columns seeds}'') \\
\text{Distribution in cols} & : & (T_{clatency} + T_{word})\ \log_2 n\ (one - to - all\ communication) \\
\text{Multiplication} & : & T_{flop} \\
\text{Assembly along rows} & : & (T_{clatency} + T_{word})\ \log_2 n \\
\text{Parallel time} & : & T_{par} = \Theta(\log_2 n) \\
\text{Process time} & : & T_{pr} = \Theta(p \log_2 n) \equiv \Theta(n^2 \log_2 n)
\end{array}
$$

The algorithm is apparently **not cost optimal** which means that the processors are not used efficiently. On the other hand, the parallel run is fast.

Parallel operations with dense vectors and matrices (continued)

Block 2D partitioning: dense matvec

Distribution: Assume $p < n^2$ processors arranged in a 2D mesh $\sqrt{p} \times \sqrt{p}$. The blocks owned by the individual processors are square with dimensions $n/\sqrt{p}$.

Algorithm

Parallel dense matrix-vector multiplication (block 2D partitioning)

1. Initial alignment.

2. Vector distribution along processor columns - $\sqrt{p}$ parallel one-to-all broadcasts

3. Local block multiplications

4. Assembling the result at one of processors in each row - $\sqrt{p}$ parallel one-to-all reductions

Parallel operations with dense vectors and matrices (continued)

Block 2D partitioning: dense matvec (continued)

$$
\begin{array}{rcl}
\text{Sequential time} & : & T_{seq} = \Theta(n^2) \\
\text{Alignment time} & : & (T_{clatency} + T_{word}\ n/\sqrt{p})\ \log_2 \sqrt{p} \\
 & : & (\mathrm{one-to-one\ with\ n/\sqrt{p};\ can\ be\ actually\ smaller}) \\
\text{Distribution in columns} & : & (T_{clatency} + T_{word}\ n/\sqrt{p})\ \log_2 \sqrt{p} \\
\text{Multiplication} & : & n/\sqrt{p}\ (2n/\sqrt{p} - 1) T_{flop} = \Theta(n^2/p) \\
\text{Assembly along rows} & : & (T_{clatency} + T_{word}\ n/\sqrt{p})\ \log_2 \sqrt{p} \\
 & : & (\mathrm{reduction\ in\ a\ row}) \\
\text{Parallel time} & : & T_{par} \approx 2\, T_{flop}\, n^2/p + T_{clatency}\ \log_2 p + T_{word}\ (n/\sqrt{p}) \log_2 p \\
\text{Process time} & : & T_{pr} = \Theta(n^2) + \Theta(p \log_2 p) + \Theta(n\sqrt{p} \log_2 p)
\end{array}
$$

The maximum number of processors that can be used cost optimally can be derived as follows:

Parallel operations with dense vectors and matrices (continued)

Block 2D partitioning: dense matvec (continued)

Consider the expression for the process time. We must have

$$\sqrt{p}\log_2 p = O(n) \quad \text{which implies}$$

$$\begin{aligned} p\log_2^2 p &= O(n^2) \\ \log_2(p\log_2^2 p) &= O(\log_2(n^2)) \\ \log_2 p + 2\log_2\log_2 p \approx \log_2 p &= O(\log_2 n) \end{aligned}$$

Substituting $\log_2 p = O(\log_2 n)$ into $p\log_2^2 p = O(n^2)$ we have the **cost optimality** if

$$p = O(n^2/\log^2 n)$$

In this case

$$T_{pr} = \Theta(n^2) + T_{clatency}\, O(n^2/\log_2 n) + T_{word}\, O(n^2).$$

This is the asymptotic upper bound on the number of processors to be cost

Parallel operations with dense vectors and matrices (continued)

Block 2D partitioning: simple dense matrix-matrix multiplication

Distribution: Assume $p < n^2$ processors in a 2D mesh $\sqrt{p} \times \sqrt{p}$. Blocks owned by individual processors are square with dimensions $n/\sqrt{p}$. Consider the dense matrix-matrix multiplication from Algorithm 6.13:

Algorithm

Dense matrix-matrix multiplication

```
for $i = 1, \dots, \sqrt{p}$ do
        for $j = 1, \dots, \sqrt{p}$ do
            $C_{ij} = 0$
            for $k = 1, \dots, \sqrt{p}$ do
                $C_{ij} = C_{ij} + A_{ik} B_{kj}$
        end $k$
    end $j$
end $i$
```

Parallel operations with dense vectors and matrices (continued)

Block 2D partitioning: simple dense matmat (continued)

P0	P1	P2	...	...	P5
P6					
...					
					...
				...	P25

Parallel operations with dense vectors and matrices (continued)

Block 2D partitioning: simple dense matmat (continued)

$$\begin{array}{rcl}
\text{Sequential time} & : & T_{seq} = \Theta(n^3) \\
\text{Two broadcast steps} & : & 2(T_{clatency} \log_2 \sqrt{p} + T_{word}(n^2/p)(\sqrt{p} - 1)) \\
 & : & \mathrm{all - to - all} \\
 & : & (\sqrt{\mathrm{p}} \text{ concurrent broadcasts among groups of } \sqrt{\mathrm{p}} \text{ processes}) \\
\text{Multiplication} & : & \text{Blocks of dimension } n/\sqrt{p},\ \sqrt{p} - times: \\
 & : & \text{Hence}: \sqrt{p} \times (n/\sqrt{p})^2 (2\,(n/\sqrt{p}) - 1)\ T_{flop} = \Theta(n^3/p)\ T_{flop} \\
 & : & \text{The factor } \sqrt{\mathrm{p}} \text{ is here for the number of matmats} \\
\text{Parallel time} & : & T_{par} \approx 2\,T_{flop}\, n^3/p + T_{clatency} \log_2 p + 2\,T_{word}\ n^2/\sqrt{p} \\
\text{Process time} & : & T_{pr} = \Theta(n^3) + T_{clatency} p \log_2 p + 2\,T_{word}\, n^2 \sqrt{p}
\end{array}$$

Cost optimality is achieved for $p = O(n^2)$.

Parallel operations with dense vectors and matrices (continued)

Cannon algorithm: local memory efficient dense matmat

- Replaces the traditional scheme of coarse interleaving of communication and computation from Algorithm 7.8 by a finer scheme.
- Uses 2D distribution with p processors, $p = k^2$ for some $k = \sqrt{p} > 1$.
- Remind that standard form of updates is given by

$$C_{ij} = \sum_{k=1}^{\sqrt{p}} A_{ik} B_{kj} \qquad (36)$$

Parallel operations with dense vectors and matrices (continued)

Cannon algorithm: local memory efficient dense matmat

- The summation and communication in the Cannon algorithm are interleaved using the following rule that combines the summation and **cyclic block shifts**

$$C_{ij} = \sum_{k=1}^{\sqrt{p}} A_{i,i+j+k-1 \, mod \, \sqrt{p}} B_{i+j+k-1 \, mod \, \sqrt{p},j}. \tag{37}$$

- That is, for example:

$$C_{23} = A_{25}B_{53} + A_{26}B63 + \ldots \equiv A_{21}B_{13} + A_{22}B23 \ldots$$

- How the computation and communication can be mixed? So that at the position of C_{ij} we have the correct terms.

Parallel operations with dense vectors and matrices (continued)

Cannon algorithm: local memory efficient dense matmat (continued)

- Initial alignment:
  - Blocks in i-th row in matrix A are cyclically shifted left by $i-1$ positions.
  - Blocks in j-th column in matrix B are cyclically shifted up by $j-1$ positions.
- Then the computation is in-place multiplying the blocks actually available at the positions of the processors. Communication based on cyclic shifts of data. Blocks in A are after each multiply-add cyclically shifted by one position left in its row and blocks in B by one position up in its column as demonstrated below.
- Example: 4×4, $p = 16 = 4 \times 4$ processors, block matrices having square blocks.
- The 16 processors correspond to 2D partitioning of the input matrices and the resulting matrix product C is partitioned in the same way.

Parallel operations with dense vectors and matrices (continued)

Cannon algorithm: local memory efficient dense matmat (continued)

- Original matrices

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{pmatrix} \quad \begin{pmatrix} B_{11} & B_{12} & B_{13} & B_{14} \\ B_{21} & B_{22} & B_{23} & B_{24} \\ B_{31} & B_{32} & B_{33} & B_{34} \\ B_{41} & B_{42} & B_{43} & B_{44} \end{pmatrix}$$

Parallel operations with dense vectors and matrices (continued)

Cannon algorithm: local memory efficient dense matmat (continued)

- Step 1: initial alignment

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{22} & A_{23} & A_{24} & A_{21} \\ A_{33} & A_{34} & A_{31} & A_{32} \\ A_{44} & A_{41} & A_{42} & A_{43} \end{pmatrix} \quad \begin{pmatrix} B_{11} & B_{22} & B_{33} & B_{44} \\ B_{21} & B_{32} & B_{43} & B_{14} \\ B_{31} & B_{42} & B_{13} & B_{24} \\ B_{41} & B_{12} & B_{23} & B_{34} \end{pmatrix}$$

Parallel operations with dense vectors and matrices (continued)

Cannon algorithm: local memory efficient dense matmat (continued)

- Step 2: after first multiplication and communication

$$\begin{pmatrix} A_{12} & A_{13} & A_{14} & A_{11} \\ A_{23} & A_{24} & A_{21} & A_{22} \\ A_{34} & A_{31} & A_{32} & A_{33} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{pmatrix} \quad \begin{pmatrix} B_{21} & B_{32} & B_{43} & B_{14} \\ B_{31} & B_{42} & B_{13} & B_{24} \\ B_{41} & B_{12} & B_{23} & B_{34} \\ B_{11} & B_{22} & B_{33} & B_{44} \end{pmatrix}$$

Parallel operations with dense vectors and matrices (continued)

Cannon algorithm: local memory efficient dense matmat (continued)

- Step 3: after the second numerical operation (multiplication and adding to the partially formed block of C) and communication

$$\begin{pmatrix} A_{13} & A_{14} & A_{11} & A_{12} \\ A_{24} & A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{42} & A_{43} & A_{44} & A_{41} \end{pmatrix} \quad \begin{pmatrix} B_{31} & B_{42} & B_{13} & B_{24} \\ B_{41} & B_{12} & B_{23} & B_{34} \\ B_{11} & B_{22} & B_{33} & B_{44} \\ B_{21} & B_{32} & B_{43} & B_{14} \end{pmatrix}$$

Parallel operations with dense vectors and matrices (continued)

Cannon algorithm: local memory efficient dense matmat (continued)

To see the computation more closely, consider, for example, computation of the entry C_{23}. This entry is in the standard algorithm given by

$$C_{23} = A_{21}B_{13} + A_{22}B_{23} + A_{23}B_{33} + A_{24}B_{43}. \tag{38}$$

Observing the position $(2,3)$ in the depicted step we can see that after the first step we have at this position the product of the residing blocks, that is

$$A_{24}B_{43} \tag{39}$$

After the second step we add to the partial product at this position the product

$$A_{21}B_{13} \tag{40}$$

and so on.

Parallel operations with dense vectors and matrices (continued)

Cannon algorithm: local memory efficient dense matmat (continued)

$$
\begin{array}{rcl}
\text{Sequential time} & : & T_{seq} = \Theta(n^3) \\
\text{Comm (shift of a pos)} & : & 2(T_{clatency} + n^2/p\,T_{word}) \\
 & : & (two\ matrix\ blocks\ with\ n^2/p\ numbers) \\
\text{Comm (all shifts)} & : & 2(T_{clatency} + n^2/p\,T_{word})\sqrt{p} \\
\text{Comm (initial align)} & : & the\ same\ order\ at\ most\ -\ neglected \\
 & : & (only\,a\,few\,longer\,shifts) \\
\text{Parallel time} & : & T_{par} = T_{flop}\ \Theta(n\,\dfrac{n}{\sqrt{p}}\,\dfrac{n}{\sqrt{p}}) + 2T_{clatency}\sqrt{p} + 2T_{word}\ n^2/\sqrt{p} \\
 & : & T_{par} = T_{flop}\ \Theta(n^3/p) + 2T_{clatency}\sqrt{p} + 2T_{word}\ n^2/\sqrt{p} \\
\text{Process time} & : & T_{pr} = \Theta(n^3) + \Theta(p\sqrt{p}) + \Theta(n^2\sqrt{p})
\end{array}
$$

Parallel operations with dense vectors and matrices (continued)

Cannon algorithm: local memory efficient dense matmat (continued)

- Parallel time and cost-optimality conditions are asymptotically the same as above
- Cannon algorithm can be generalized for multiplying **rectangular matrices**. There are similar approaches along this line.
- Cannon algorithm is a **memory-efficient** version of the matrix-matrix multiplication in the sense that it uses constant and predictable memory size for each processor.

Parallel operations with dense vectors and matrices (continued)

Scalable universal matrix multiply - SUMMA

- Generally less efficient than the Cannon algorithm.
- Much easier to generalize for non-uniform splittings and unstructured processor grids.
- Called **SUMMA (Scalable Universal Matrix Multiply)** but the same algorithmic principles have been proposed a couple of times independently. This is the scheme used inside the ScaLapack library

Parallel operations with dense vectors and matrices (continued)

SUMMA algorithm: local memory efficient dense matmat (continued)

- Assume that we have to form the matrix product $C = AB$ such that the all the involved matrices are distributed over a two-dimensional grid of processors

$$p_r \times p_c.$$

- The block entry C_{ij} within the grid of processors can be formally written in the standard way as

$$C_{ij} = \begin{pmatrix} A_{i1} & \dots & A_{i,p_c} \end{pmatrix} \begin{pmatrix} B_{1j} \\ \vdots \\ B_{p_r,j} \end{pmatrix} = \tilde{A}_i \tilde{B}_j \qquad (41)$$

where the blocks multiplied together have compatible dimensions.

Parallel operations with dense vectors and matrices (continued)

Scalable universal matrix multiply - SUMMA (continued)

- The whole $\tilde{A}_i$ is assigned to the i-the row of processors and $\tilde{B}_j$ is assigned to the j-th column of processors in the processor grid.
- Assume now that the introduced block row $\tilde{A}_i$ and block column $\tilde{B}_j$ are split into k blocks as follows

$$\tilde{A}_i = \left(\tilde{A}_i^1, \ldots, \tilde{A}_i^k\right) \begin{pmatrix} \tilde{B}_j^1 \\ \vdots \\ \tilde{B}_j^k \end{pmatrix} \tag{42}$$

- Then we can equivalently write as a sum of outer products.

$$C_{ij} = \sum_{l=1}^{k} \tilde{A}_i^l \tilde{B}_j^l \tag{43}$$

- i, j denote row and column indices of the processor grid, k can be the column dimension of A (equal to the row dimension of B) or another blocking using a smaller k can be used.

Parallel operations with dense vectors and matrices (continued)

Scalable universal matrix multiply - SUMMA (continued)

- The communication within the algorithm is based on **broadcasts** within the processor rows and processor columns instead of the circular shifts used in the Cannon algorithm. And these broadcasts can be efficiently implemented depending on the computational architecture.
- The number of processors p_r and p_c can be generally different. Moreover, the mapping of blocks to processing units using $p_r \times p_c$ grid of processors can be rather general.
- In the following example we use, for simplicity, uniform block size N (**the same for rows and columns**).

Parallel operations with dense vectors and matrices (continued)

Scalable universal matrix multiply - SUMMA (continued)

Algorithm

SUMMA parallel matrix-matrix multiplication

```
Set all $C_{ij} = 0$
for $l = 1, \dots, k$ do
    for $i = 1, \dots, p_r$ do in parallel
        One-to-all broadcast of $\tilde{A}_i^l$ to row block owners $p_{i1}, \dots, p_{ip_c}$ as $A_{temp}$
    end $i$
    for $j = 1, \dots, p_c$ do in parallel
        One-to-all broadcast of $\tilde{B}_j^l$ to column block owners $p_{1j}, \dots, p_{p_r j}$ as $B_{temp}$
    end $j$
    Once data received, perform parallel add-multiply $C_{ij} = C_{ij} + A_{temp} B_{temp}$
end $l$
```

Parallel operations with dense vectors and matrices (continued)

Scalable universal matrix multiply - SUMMA (continued)

- SUMMA: sends more data than the Cannon algorithm; more flexible.
- If we assume that $p_r \equiv p_c \equiv \sqrt{p}$ as well as that we multiply square matrices of dimensions n then the parallel time is

$$T_{par} \approx 2n^3/p\, T_{flop} + 2\, T_{clatency}\, \sqrt{p}\, \log_2 p + 2\, T_{word}\, n^2/\sqrt{p} \log_2 p \quad (44)$$

  since each of the two (row and column) communication steps sends $\sqrt{p}$-times blocks of the size $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ (along rows and columns).
- This gives overall the message size

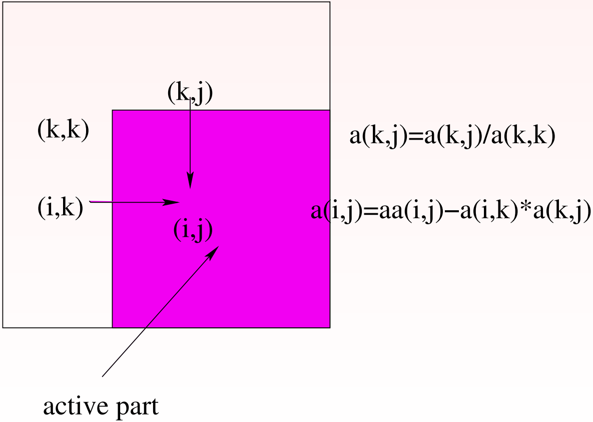
$$\sqrt{p}\, (\frac{n}{\sqrt{p}})^2 = \frac{n^2}{\sqrt{p}}.$$

  The latency corresponds to $2\sqrt{p}$ communication steps. More ways to make the implementation more specific efficient.

Parallel operations with dense vectors and matrices (continued)

Gaussian elimination: kij, 1D decomposition

Consider $p = n$ and 1D decomposition of the matrix $A \in R^{n \times n}$ by rows. The kij scheme of the factorization/elimination is depicted below.



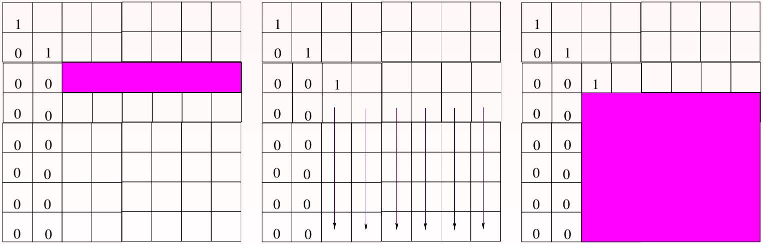
Parallel operations with dense vectors and matrices (continued)

Gaussian elimination: kij, 1D decomposition

Consider the approach where we interleave

- Operations on rows (row updates) and
- communication of the processed row (part of U) to all the other processors

Schematically depicted as



Parallel operations with dense vectors and matrices (continued)

Gaussian elimination: kij, 1D decomposition

Consider the process costs.

$$\begin{aligned}
\text{Seq time} &: \quad T_{seq} = T_{flop}((2/3)n^3 + O(n^2)) \equiv \Theta(n^3) \\
\text{Comm rows} &: \quad \sum_{k=1}^{n-1} (T_{clatency} + T_{word}(n-k)) \log_2 n \\
&: \quad (\text{one}-\text{to}-\text{all}),\ \text{row has } n-k \text{ entries} \\
&: \quad \log_2 n \text{ is an upper bound for the number of comm steps} \\
\text{Elimination} &: \quad \approx 3 \sum_{k=1}^{n} (n-k)\ T_{flop} = 3n(n-1)/2\ T_{flop} \\
&: \quad (\text{scaling, and multiply}-\text{add}) = 3 \text{ operations for each entry} \\
\text{Parallel time} &: \quad T_{par} \approx 3n(n-1)/2\ T_{flop} + T_{clatency}\ n \log_2 n + T_{word}(n(n-1)/2) \log_2 n \\
\text{Process time} &: \quad T_{pr} \approx \Theta(n^3) + \Theta(n^2\ \log_2 n) + \Theta(n^3\ \log_2 n)
\end{aligned}$$

Not cost-optimal: the process time is $\Theta(n^3 \log n)$.

Parallel operations with dense vectors and matrices (continued)

Pipelined Gaussian elimination: kij, 1D decomposition

The pipelined Gaussian elimination computes and communicates **asynchronously**. Consider again $p = n$ and 1D decomposition based on assigning a row to a processor. Each processor repeatedly performs the following set of three operations on matrix rows.

Algorithm

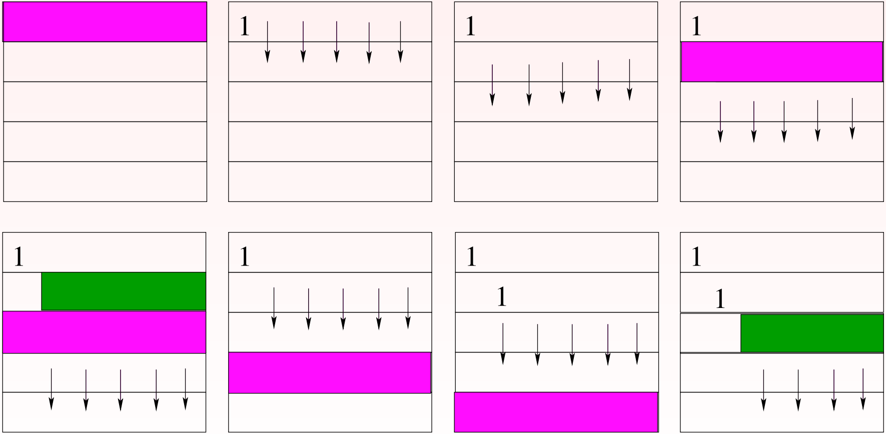
Pipelined Gaussian elimination

```
    1. if a processor has data used by other processors, it sends the data them
    2. if a processor can has all data for a computation, it computes
    3. otherwise the processor waits
```

The following figures demonstrate the process.

Parallel operations with dense vectors and matrices (continued)

Pipelined Gaussian elimination: kij, 1D decomposition



. . .

Parallel operations with dense vectors and matrices (continued)

Pipelined Gaussian elimination: kij, 1D decomposition (continued)

Sequential time	:	$T_{seq} = T_{flop}((2/3)n^3 + O(n^2)) \equiv \Theta(n^3)$
Total number of steps	:	$\Theta(n)$
	:	each processor either computes or communicates data
	:	with previous Θ(n) rows
	:	Each of these operations has O(n) cost
	:	– – communication of O(n) entries;
	:	– – division O(n) entries by a scalar;
	:	– – elimination step on O(n) entries
Parallel time	:	$T_{par} = \Theta(n^2)$
Process time	:	$T_{par} = \Theta(n^3)$

Parallel operations with dense vectors and matrices (continued)

Pipelined Gaussian elimination: kij, 1D decomposition (continued)

- The multipliers of the asymptotic complexity are **not the same** as in the sequential case. Some processors will stay in any case idle. In practice, a partial solution is to use 1D **cyclic** decomposition.
- Also 2D distribution possible. This is more scalable. For example, for **block 2-D partitioning** we get the process time

$$T_{pr} = \Theta(n^3/p) \tag{45}$$

  for p processors but we will not discuss this here.
- A partial pivoting can be embedded into the **standard** parallel elimination based on 1D partitioning explained above at the expense of $O(n)$ search in each row. In case of the pipelined approach, **pivoting is strongly restricted**. Weaker variants of pivoting may lead to **strong degradation** of the numerical quality of the algorithm.

Parallel operations with dense vectors and matrices (continued)

Solving triangular systems

First the sequential algorithm

Algorithm

Sequential back-substitution for $Ux = y$, $U = (u_{ij})$ is unit upper triangular.

```
do k=n,1,-1
    $x_i = y_i$
    do $i = k-1, 1, -1$
        $y_i = y_i - x_k u_{ik}$
    end do
end do
```

Sequential complexity of the backward substitution is

$$T_{seq} = (n^2/2 + O(n))T_{flop}.$$

Parallel operations with dense vectors and matrices (continued)

Solving triangular systems (continued)

- Two possibilities of parallel implementation

1. Rowwise block 1-D decomposition

Rowwise block 1-D decomposition with y decomposed accordingly. Blocks have n/p rows. Back-substitution with **pipelining** results in the constant communication time since the algorithm always

- either communicates one number (component of the solution)
- performs n/p flops.

All computational units work asynchronously in each of the $n-1$ steps. Each of these steps is dominated then by $O(n/p)$ cost. Then

$$\begin{aligned} \text{Parallel time} &: \quad T_{par} = \Theta(n^2/p) \\ \text{Process time} &: \quad T_{pr} = O(n^2) \end{aligned}$$

Apparently, the algorithm is **cost-optimal**.

Parallel operations with dense vectors and matrices (continued)

Solving triangular systems (continued)

2. Block 2D decomposition

This decomposition is be better but it still does not lead to the cost optimality. If the block 2-D partitioning using the $\sqrt{p} \times \sqrt{p}$ grid of computational units, $\sqrt{p}$ steps and the pipelined communication we get the parallel time

$$\begin{aligned}
\text{Parallel time} &: \quad T_{par} = \Theta(n^2/\sqrt{p}) \equiv \Theta((n/\sqrt{p}) \times (n/\sqrt{p}) \times \sqrt{p}) \\
&: \quad \sqrt{p} \text{ steps } \textit{substitution costs } (n/\sqrt{p}) \times (n/\sqrt{p}) \\
\text{Process time} &: \quad T_{pr} = O(n^2\sqrt{p})
\end{aligned}$$

Parallelizing linear recurrences: subvector scaling

4. Parallelizing linear recurrences

$$x_i = b + (i-1)a,\ i = 1, \ldots, n, \tag{46}$$

where a and b are scalars.

- Sequential loop can be used

$$x_1 = b, x_2 = b + a, \ldots, x_i = x_{i-1} + a,\ i = 2, \ldots, n, \tag{47}$$

- But this loop does not **straightforwardly vectorize** or **parallelize**.
- Can be parallelized like this - curse of logarithmic depth

$$\begin{aligned} x_1 &= b \\ x_2 &= b + a \\ x_{3:4} &= x_{1:2} + 2a \\ x_{5:8} &= x_{1:4} + 4a \\ x_{9:16} &= x_{1:8} + 8a \end{aligned}$$

$$\ldots$$

Parallelizing linear recurrences: scalar sums

4. Parallelizing linear recurrences

- Simple case

$$x = \sum_{i=1}^{n} d_i. \tag{48}$$

- Procedure depicted for $n = 8$.

$$s_1 = x_1 + x_2 \quad s_2 = x_3 + x_4 \quad s_3 = x_5 + x_6 \quad s_4 = x_7 + x_8 \tag{49}$$

$$t_1 = s_1 + s_2 \quad t_2 = s_3 + s_4 \tag{50}$$

$$x = t_1 + t_2 \tag{51}$$

Parallelizing first-order linear recurrences

4. Parallelizing linear recurrences

- More general case of linear recurrence

$$x_i = d_i + a_i x_{i-1},\ i = 1, \ldots, n;\ x_0 = 0.$$

- Distinguish two possibilities of the result
  - Needed only the last x_i
  - Needed all intermediate x_i

Parallelizing first-order linear recurrences

4. Parallelizing linear recurrences

$$x_i = d_i + a_i x_{i-1},\ i = 1, \dots, n;\ x_0 = 0.$$

- Two subsequent expressions for x_{i-1} and x_i

$$x_{i-1} = d_{i-1} + a_{i-1} x_{i-2},\ x_i = d_i + a_i x_{i-1}$$

combined by eliminating x_{i-1}, getting dependency of x_i on

$$x_{i-2} \equiv x_{i-2^1}$$

- We get

$$x_i = d_i + a_i(d_{i-1} + a_{i-1} x_{i-2}) = a_i d_{i-1} + d_i + a_i a_{i-1} x_{i-2} = d_i^{(1)} + a_i^{(1)} x_{i-2}.$$

- subsequent eliminations to find the dependency of x_i on $x_{i-4} \equiv x_{i-2^2}$ follow.
- New equations relate variables with the distance 2^2.
- A **fan-in** algorithm follows.

Parallelizing first-order linear recurrences (continued)

4. Parallelizing linear recurrences

Remind: the algorithm provides **on output** only x_n.

Algorithm

fan-in *algorithm for linear recurrences,* $n = 2^{\log_2 n}$.

Input: Initial coefficients use the notation $a_i^{(0)} \equiv a_i$, $d_i^{(0)} \equiv d_i$, $i = 1, \ldots, n$.

```
1. for $k = 1, \ldots, \log_2 n$ do
2.     for $i = 2^k, \ldots, n$ step $2^k$ do
3.         $a_i^{(k)} = a_i^{(k-1)} a_{i-2^{k-1}}^{(k-1)}$
4.         $d_i^{(k)} = a_i^{(k-1)} d_{i-2^{k-1}}^{(k-1)} + d_i^{(k-1)}$
5.     end $i$
6. end $k$
7. $x_n = d_n^{(\log_2 n)}$
```

- Before the dependency of x_n on $x_{n-1} \equiv x_{n-2^0}$ is explicitly known.
- After: $\log_2 n$ steps, dependency of x_n on x_0 is

$$x_n = d_n^{(\log_2 n)}. \tag{52}$$

Parallelizing first-order linear recurrences (continued)

4. Parallelizing linear recurrences

- If **all values** $x_1, \ldots, x_n$ needed, **more work** is necessary. We should then compute also coefficients for some other equations.

- Before putting the corresponding procedure formally let us first show a simple scheme **graphically**: in step k, $k = 1, \ldots, \log_2 n$ of this scheme the first $2^k - 1$ components are computed.

Parallelizing first-order linear recurrences (continued)

4. Parallelizing linear recurrences

- Getting all x_i cascadically: Start with x_0. In the step 1 we can compute x_1 since we know $a_1^{(0)}$, $d_1^{(0)}$ as well as x_0 (the rest x_i's are updated). In step 2 we can compute x_2 and x_3 based on the known coefficients and the known values of x_0 and x_1 (the rest x_i's are updated). In the step 3 we know $2^2 - 1$ components of x and we can compute further 2^2 values and so on.

$$
\begin{array}{l|l|l}
x_1 = a_1^{(0)} x_{1-2^0} + d_1^{(0)} & & \\
x_2 = a_2^{(0)} x_{2-2^0} + d_2^{(0)} & x_2 = a_2^{(1)} x_{2-2^1} + d_2^{(1)} & \\
x_3 = a_3^{(0)} x_{3-2^0} + d_3^{(0)} & x_3 = a_3^{(1)} x_{3-2^1} + d_3^{(1)} & \\
x_4 = a_4^{(0)} x_{4-2^0} + d_4^{(0)} & x_4 = a_4^{(1)} x_{4-2^1} + d_4^{(1)} & x_4 = a_4^{(2)} x_{4-2^2} + d_4^{(2)} \\
\ldots & \ldots & \ldots \\
x_n = a_n^{(0)} x_{n-2^0} + d_n^{(0)} & x_n = a_n^{(1)} x_{n-2^1} + d_n^{(1)} & x_n = a_n^{(2)} x_{n-2^2} + d_n^{(2)}
\end{array}
$$

Parallelizing first-order linear recurrences (continued)

The algorithm of the so called **cascadic** approach is given below.

Algorithm

Cascadic algorithm *for first-order linear recurrences.*

Input: Initial coefficients use the notation $a_i^{(0)} \equiv a_i$, $d_i^{(0)} \equiv d_i$, $i = 1, \ldots, n$.

```
1. for $k = 1, \ldots, \log_2 n$ do
2.     for $i = 2^k, \ldots, n$ step $1$ do
3.         $a_i^{(k)} = a_i^{(k-1)} a_{i-2^{k-1}}^{(k-1)}$
4.         $d_i^{(k)} = a_i^{(k-1)} d_{i-2^{k-1}}^{(k-1)} + d_i^{(k-1)}$
5.     end $i$
6. end $k$
```

- The computation of all x_i, $i = 1, \ldots, n$ can be done even more efficiently as we will show below for a slightly different problem, for solving systems of linear equations with a tridiagonal matrix. Namely, like **forward** and **back** solve step.

Parallel prefix operation

4. Parallelizing linear recurrences

- Standardization: parallel prefix operation
- General associative operation $\heartsuit$. The prefix operation of the length n:

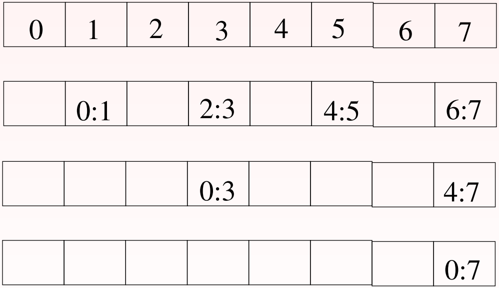
$$
\begin{array}{rcl}
y_0 & = & x_0 \\
y_1 & = & x_0 \heartsuit x_1 \\
 & \dots & \\
y_n & = & x_0 \heartsuit\, x_1 \ldots \heartsuit x_n
\end{array}
$$

- All $y_0, y_1, \ldots, y_n$: sequentially in $O(n)$ operations.
- But also in parallel in $O(\log_2 n)$ parallel steps
- Assume $n = 2^{\log_2 n}$.
- Two parallel steps for all y_i, one parallel step to get y_n only.
- The schemes are called fan-in and fan-out.

Parallel prefix operation

4. Parallelizing linear recurrences

The following figure demonstrates the three steps of the first pass of the parallel prefix operation for $n = 2^3$.

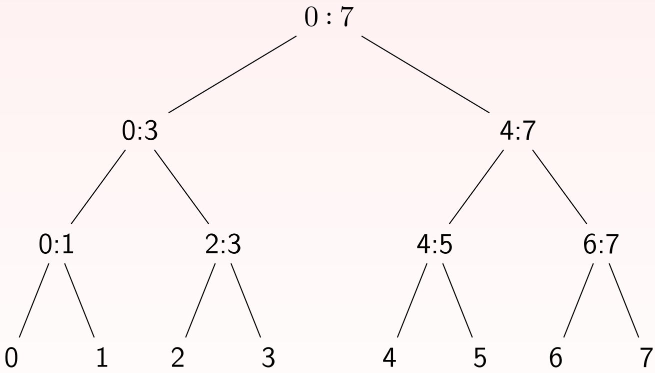


0	1	2	3	4	5	6	7
	0:1		2:3		4:5		6:7
			0:3				4:7
							0:7

Parallel prefix operation

4. Parallelizing linear recurrences

- In binary tree



$$\color{red}{value(node) = value(left\_son) + value(right\_son)}$$

- **bottom-up summing**

Parallel prefix operation

- Fan-in : formal description

Algorithm

Fan-in of the parallel prefix operation

```
for $k = 0, \dots, \log_2 n - 1$ do
    for $j = 1, \dots, 2^{\log_2 n - k - 1}$ step $2^k$ do in parallel
        $y_{j\times 2^{k+1}-1} = y_{j\times 2^{k+1}-1} \heartsuit y_{j\times 2^{k+1}-2^k-1}$
    end $j$
end $k$
```

Parallel prefix operation

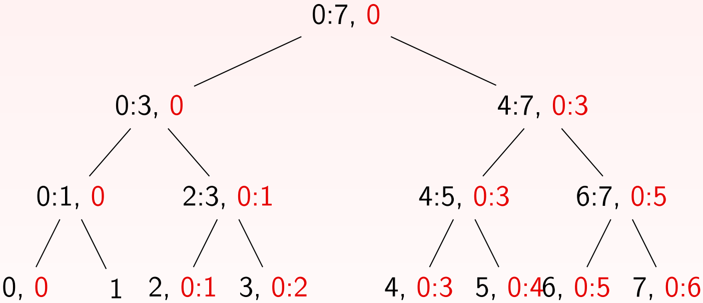
Second pass is a top-down computation of all the remaining sums.

0	1	2	3	4	5	6	7
	0:1		2:3		4:5		6:7
			0:3				4:7
							0:7
					0:5		
		0:2		0:4		0:6	

Parallel prefix operation

4. Parallelizing linear recurrences

- In binary tree



$$prefix(root) = 0$$
$$prefix(left\_son) = prefix(node)$$
$$prefix(right\_son) = prefix(node) + value(left\_son)$$

- **top-down summing**

Parallel prefix operation

The scheme for $n = 2^4$.

$$0:1 \quad 2:3 \quad 4:5 \quad 6:7 \quad 8:9 \quad 10:11 \quad 12:13 \quad 14:15$$

$$0:3 \quad 4:7 \quad 8:11 \quad 12:15$$

$$0:7 \quad 8:15$$

$$0:15$$

$$0:11$$

$$0:5 \quad 0:9 \quad 0:13$$

$$0:2 \quad 0:4 \quad 0:6 \quad 0:8 \quad 0:10 \quad 0:12 \quad 0:14$$

- One **fan-in** and one **fan-out** pass.
- This results in $O(\log_2 n)$ parallel steps.
- The first pass of the prefix operation is just a **fan-in**
- The second pass is **less straightforward** and corresponds to the second step of the cyclic reduction

Parallel prefix operation

Once the prefix operation is **standardized**, it can be used to evaluate known problems in parallel similarly to the previously mentioned linear recursions.

Algorithm

Parallelizing linear recursion $z_{i+1} = a_i z_i + b_i$ exploiting the parallel prefix (PP) computation

```
Compute $p_i = a_0 \ldots a_i$ using PP with ♡ ≡ .
for $i = 1, \ldots, n$ in parallel do
        $\beta_i = b_i / p_i$
end $i$
Compute $s_i = \beta_0 + \ldots + \beta_i$ using PP with ♡ ≡ +
for $i = 1, \ldots, n$ in parallel do
        $z_i = s_i p_{i-1}$
end $i$
```

Second-order linear recurrences

4. Parallelizing linear recurrences

second-order linear recurrence

$$x_i = a_i + b_{i-2}x_{i-1} + c_{i-2}x_{i-2},\ i = 3,\ldots,n;\ x_1 = a_1,\, x_2 = a_2.$$

This recurrence can be rewritten in the first-order form

$$\begin{pmatrix} x_i \\ x_{i+1} \end{pmatrix} = \begin{pmatrix} 0 \\ a_{i+1} \end{pmatrix} + \begin{pmatrix} 0 & 1 \\ c_{i-1} & b_{i-1} \end{pmatrix} \begin{pmatrix} x_{i-1} \\ x_i \end{pmatrix},\ i = 2,\ldots,n-1 \tag{53}$$

initialized by

$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} a_1 \\ a_2 \end{pmatrix}. \tag{54}$$

Cyclic reduction

4. Parallelizing linear recurrences

- Another scheme based on the same principle: factorization and using explicit permutation of even and odd rows and columns (exploiting symmetry)
- $$A = \begin{pmatrix} b_1 & c_1 & & & & & \\ a_2 & b_2 & c_2 & & & & \\ & a_3 & b_3 & c_3 & & & \\ & & a_4 & b_4 & c_4 & & \\ & & & a_5 & b_5 & c_5 & \\ & & & & a_6 & b_6 & c_6 \\ & & & & & a_7 & b_7 \end{pmatrix}$$
- The odd-even permutation P is given by

$$P(1, 2, ..., n)^T = (1, 3, ..., |2, 4, ...)^T \qquad (55)$$

Cyclic reduction

4. Parallelizing linear recurrences

The permuted matrix is

$$P^T A P = \begin{pmatrix} b_1 & & & & c_1 & & \\ & b_3 & & & a_3 & c_3 & \\ & & b_5 & & & a_5 & c_5 \\ & & & b_7 & & & a_7 \\ a_2 & c_2 & & & b_2 & & \\ & a_4 & c_4 & & & b_4 & \\ & & a_6 & c_6 & & & b_6 \end{pmatrix}$$

Cyclic reduction

4. Parallelizing linear recurrences

- After one step of the **fan-in** of block **partial factorization** based on the odd-indexed unknowns we get

$$\begin{pmatrix} 1 & & & & & & \\ & 1 & & & & & \\ & & 1 & & & & \\ & & & 1 & & & \\ l_1 & m_1 & & & 1 & & \\ & l_2 & m_2 & & & 1 & \\ & & l_3 & m_3 & & & 1 \end{pmatrix} \begin{pmatrix} b_1 & & & & c_1 & & \\ & b_3 & & & a_3 & c_3 & \\ & & b_5 & & & a_5 & c_5 \\ & & & b_7 & & & a_7 \\ & & & & \bar{b}_1 & \bar{c}_1 & \\ & & & & \bar{a}_2 & \bar{b}_2 & \bar{c}_2 \\ & & & & & \bar{a}_3 & \bar{b}_3 \end{pmatrix} \tag{56}$$

- Forward reduction (**fan-in**) / backward substitution (**fan-out**).
- This rewriting (and its adoption for more general A) reveals vectorizability and parallelizability.
- Using blocks leads to useful approaches.
- May result in worse cache treatment.

Recursive doubling technique for polynomials

5. Recursive doubling for polynomials

- Polynomials in real of complex variable x.
- Standard evaluation of polynomials is based on the efficient **Horner's rule**:

$$
\begin{aligned}
p(x) &= a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots + a_n x^n \\
p(x) &= a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + a_n x)))
\end{aligned}
$$

Algorithm

Horner's polynomial evaluation $p(x) \equiv p^{(n)}(x)$ *in* $x \in R$.

```
1. $p^{(0)}(x) = a_n$
2. for $k = 1, \dots, n$ do
3.     $p^{(k)}(x) = a_{n-k} + x p^{(k-1)}(x)$
4. end $i$
```

Recursive doubling technique for polynomials (continued)

5. Recursive doubling for polynomials

- The Horner's scheme is **recursive** and non-vectorizable/non-parallelizable although some its transformations result in less multiplications (Knuth, 1962, 1988).
- A way to parallelize it by Estrin (1960):
- Based on finding subexpressions of the type $(\alpha + \beta x)$ and x^{2^k}.
- A few members of a sequence of partially evaluated polynomials:

$$
\begin{aligned}
p^{(3)}(x) &= (a_0 + a_1 x) + (a_2 + a_3 x)x^2 \\
p^{(4)}(x) &= (a_0 + a_1 x) + (a_2 + a_3 x)x^2 + a_4 x^4 \\
p^{(5)}(x) &= (a_0 + a_1 x) + (a_2 + a_3 x)x^2 + (a_4 + a_5 x)x^4 \\
p^{(6)}(x) &= (a_0 + a_1 x) + (a_2 + a_3 x)x^2 + ((a_4 + a_5 x) + a_6 x^2)x^4 \\
p^{(7)}(x) &= (a_0 + a_1 x) + (a_2 + a_3 x)x^2 + ((a_4 + a_5 x) + (a_6 + a_7 x)x^2)x^4
\end{aligned}
$$

Recursive doubling technique for polynomials

The computation is repeatedly divided into separate tasks that can be run in parallel. An example: evaluation of a polynomial of degree 7.

Example

Example use of the Estrin's method to evaluate a polynomial of degree 7

```
do (in parallel)
    $x^{(1)} = x^2$
    $a_3^{(1)} = a_7 x + a_6$
    $a_2^{(1)} = a_5 x + a_4$
    $a_1^{(1)} = a_3 x + a_2$
    $a_0^{(1)} = a_1 x + a_0$
end do
do (in parallel)
    $x^{(2)} = (x^{(1)})^2$
    $a_1^{(2)} = a_3^{(1)} x^{(1)} + a_2^{(1)}$
    $a_0^{(2)} = a_1^{(1)} x^{(1)} + a_0^{(1)}$
end do
Set $p(x) = a_1^{(2)} x^{(2)} + a_0^{(2)}$
```

Polynomial evaluation, discrete and fast Fourier transform (DFT, FFT)

6. DFT/FFT

- Polynomial evaluation:

$$A_n(x) = \sum_{j=0}^{n-1} a_j x^j, \tag{57}$$

  where $a_0, \ldots a_{n-1}$ are generally complex coefficients.
- Special case where x represents powers of the **complex root of unity:** (sometimes described differently)

$$\omega_n = e^{2\pi i/n} \equiv \cos(2\pi/n) + i\sin(2\pi/n) \tag{58}$$

- Assume that $n = 2^m$ for some integer $m \geq 0$.
- **Discrete Fourier transform (DFT)** is a linear transform that evaluates the polynomial

$$y_k = A_n(\omega_n^k) = \sum_{j=0}^{n-1} a_j \omega_n^{kj},\ k = 0, \ldots, n-1.$$

DFT, FFT (continued)

6. DFT/FFT

- Practical purpose of DFT: convert a finite sequence of **equally-spaced** samples of a function into a sequence of samples of a complex-valued function of **frequency**.
- Standard computation: $O(n^2)$ operations: $O(n^2)$ different powers needed, DFT can be expressed just as matrix-vector multiplication.

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega_n & \omega_n^2 & \dots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \dots & \omega_n^{2(n-1)} \\ \vdots & \vdots & \vdots & \dots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \dots & \omega_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{pmatrix} \equiv W_n a.$$

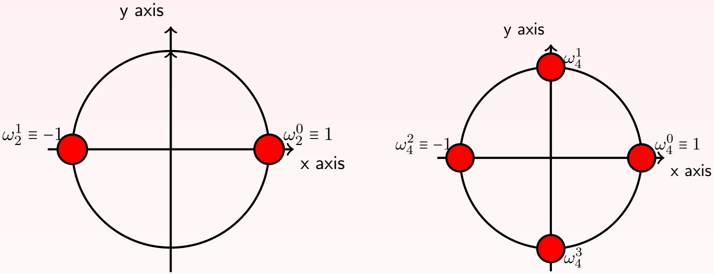
DFT, FFT (continued)

6. DFT/FFT

- But DFT polynomials can be evaluated faster exploiting specific properties of its arguments.
- This is called **fast Fourier transform (FFT)**.
- It achieves $O(n \log n)$ complexity and it is based on the recurrent strategy that we explain here.
- The trick is that some powers of ω_n are the same.

DFT, FFT (continued)

6. DFT/FFT



- Roots of unity for $n = 2$ and $n = 4$ in the complex plane.

$$\omega_4^2 = \omega_2^1$$

DFT, FFT (continued)

6. DFT/FFT

- Example: consider $n = 2$, then $n = 4$.

$$y_k = \sum_{j=0}^{1} a_j \omega_2^{kj} = (-1)^{k.0} a_0 + (-1)^{k.1} a_1 = a_0 + (-1)^k a_1,\ k = 0, 1$$

$$y_0 = a_0 + a_1,\ y_1 = a_0 - a_1$$

- $$y_k = \sum_{j=0}^{3} a_j \omega_4^{kj} = a_0 + (-i)^k a_1 + (-i)^{2k} a_2 + (-i)^{3k} a_3$$

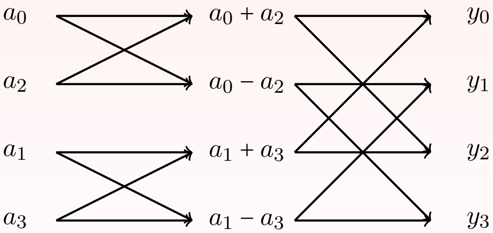
- $$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix}.$$

$y_0 = (a_0 + a_2) + (a_1 + a_3), y_1 = (a_0 - a_2) - i(a_1 - a_3), y_2 = (a_0 + a_2) - (a_1 + a_3), y_3 = (a_0 - a_2) + i(a_1 - a_3)$

DFT, FFT (continued)

6. DFT/FFT

- Communication graphically for $n = 4$:



DFT, FFT (continued)

6. DFT/FFT

- Basic idea behind FFT: split coefficients of A into the even-indexed and the odd-indexed ones.
- And do it recrsively
- Example:

$$\begin{aligned} A^{[0]}(x) &= a_0 + a_2 x + a_4 x^2 + \ldots + a_{n-2} x^{n/2-1} \\ A^{[1]}(x) &= a_1 + a_3 x + a_5 x^2 + \ldots + a_{n-1} x^{n/2-1} \end{aligned}$$

- $$A(x) = A^{[0]}(x^2) + xA^{[1]}(x^2)$$

  is expressed via (shorter) polynomials in x^2.
- Enough to evaluate the polynomials $A^{[0]}(x^2), A^{[1]}(x^2)$ and combine them together.

DFT, FFT (continued)

6. DFT/FFT

- FFT algorithm assembles the value of the polynomial $A(\omega_n^k)$ using the dependency casted as

Recurrence in the FFT transform for $n = 8$

$(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7)$

(a_0, a_2, a_4, a_6) (a_1, a_3, a_5, a_7)

(a_0, a_4) (a_2, a_6) (a_1, a_5) (a_3, a_7)

(a_0) (a_4) (a_2) (a_6) (a_1) (a_5) (a_3) $(a_7$

- FFT can be implemented recursively or non-recursively using an explicit stack.

DFT, FFT (continued)

6. DFT/FFT

- Properties of the complex roots of the unity formally
- $\omega_n^n = \omega_n^0 = 1,\ \omega_n^j \omega_n^k = \omega_n^{j+k},\ \omega_n^{n/2} = -1;\ \omega_n^{k+n/2} = -\omega_n^k$
- $\omega_{dn}^{dk} = \omega_n^k$ sometimes called the **cancellation lemma**:

$$\omega_{dn}^{dk} = (e^{2\pi i/dn})^{dk} = (e^{2\pi i/n})^k = \omega_n^k, \quad \omega_4^2 = \omega_2^1$$

- The following property ($n > 0$, even) is called the **halving** property and it is **crucial** for our purposes

$$\begin{aligned} & (\omega_n^0)^2, (\omega_n^1)^2, \ldots, (\omega_n^{n/2-1})^2, (\omega_n^{n/2})^2 \ldots,\ (\omega_n^{n-1})^2 \\ = \ & \omega_{n/2}^0, \omega_{n/2}^1, \ldots, \omega_{n/2}^{n/2-1}, \omega_{n/2}^{n/2}, \ldots, \omega_{n/2}^{n-1} \text{ cancellation} \\ = \ & \omega_{n/2}^0, \omega_{n/2}^1, \ldots, \omega_{n/2}^{n/2-1}, \omega_{n/2}^0, \ldots, \omega_{n/2}^{n/2-1} \text{ multiplying by } \omega_{n/2}^{n/2} \end{aligned}$$

DFT, FFT (continued)

6. DFT/FFT

- We can see this formally

$$\begin{aligned}(\omega_n^k)^2 &= \omega_n^{2k} = \omega_{n/2}^k \\ (\omega_n^{k+n/2})^2 &= \omega_n^{2k+n} = \omega_n^{2k}\omega_n^n = \omega_n^{2k}\,.\,1 = \omega_n^{2k} = \omega_{n/2}^k.\end{aligned}$$

- The polynomial evaluation problem **reduces** to evaluating two polynomials at the $n/2$ points of the $(n/2)$-th complex roots of unity.

DFT, FFT (continued)

Recursive implementation of FFT We will skip this

Algorithm

Recursive Fast Fourier Transform (RFFT)

```
    procedure RFFT(a)    [ $a = (a_0, a_1, \ldots a_{n-1})$ ]
    1. $n = length\_of\_a$; if $n == 1$ return $a$; set $\omega_n = e^{2\pi i/n}, \omega = 1$
    4. $a^{[0]} = (a_0, a_2, \ldots, a_{n-2})$  [ even coeffs ]  $a^{[1]} = (a_1, a_3, \ldots, a_{n-1})$  [ odd coeffs ]
    6. $y^{[0]} = RFFT(a^{[0]})$    [gets $y_k^{[0]} = A^{[0]}(\omega_{n/2}^k) \equiv A^{[0]}(\omega_n^{2k}),\ k = 0, \ldots, n/2 - 1$]
    7. $y^{[1]} = RFFT(a^{[1]})$    [gets $y_k^{[1]} = A^{[1]}(\omega_{n/2}^k) \equiv A^{[1]}(\omega_n^{2k}),\ k = 0, \ldots, n/2 - 1$]
       explanation of the following loop: compose $y$ based on the previous level
    8  for $k = 0, \ldots, n/2 - 1$ step $1$ do
    9.     $y_k = y_k^{[0]} + \omega y_k^{[1]}$, explanation: $y_k = A(\omega_n^k) = A^{[0]}(\omega_n^{2k}) + \omega_n^k A^{[1]}(\omega_n^{2k})$
           At this moment, $\omega = \omega_n^k$ and the squares $\omega_n^{2k}$ are computed as $\omega_{n/2}^k$
    10.    $y_{k+n/2} = y_k^{[0]} - \omega y_k^{[1]}$,  explanation: $-\omega = \omega_n^{k+n/2}$
           explanation: $y_{k+n/2} = A(\omega_n^{k+n/2}) = A^{[0]}(\omega_n^{2k+n}) + \omega_n^{k+n/2} A^{[1]}(\omega_n^{2k+n})$
  $= A^{[0]}(\omega_n^{2k}) + \omega_n^{k+n/2} A^{[1]}(\omega_n^{2k}) = y_k^{[0]} + \omega_n^{k+n/2} y_k^{[1]} = y_k^{[0]} + \omega_n^{k+n/2} y_k^{[1]} = y_k^{[0]} - \omega_n^k y_k^{[1]}$.
    11.    $\omega = \omega\omega_n$
    12. end
    13. return $y$    [of the length of the input]
```

DFT, FFT (continued)

6. DFT/FFT

Operation count for the FFT is given by

$$T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n).$$

A non-recursive algorithm (with explicit stack) is possible as well.

- The basic scheme can be generalized for general n and can be efficiently implemented.
- For example, the order of the coefficients in the leaves of the computational scheme (see above) can be determined by a **bit-reversal** permutation.

DFT, FFT (continued)

6. DFT/FFT

Inverse FFT Rewriting the DFT in the matrix form, it is easy to see that the inverse linear transform can be **computed fast** as well. Namely, if we write DFT in the form

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega_n & \omega_n^2 & \dots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \dots & \omega_n^{2(n-1)} \\ \vdots & \vdots & \vdots & \dots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \dots & \omega_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{pmatrix} \equiv W_n a. \qquad (59)$$

Then

$$y = W_n a \leftrightarrow a = W_n^{-1} y$$

and we can see that

$$(W_n^{-1})_{jk} = \omega_n^{-kj}/n.$$

Extreme parallelism for solving linear algebraic systems

7. Extreme parallelism

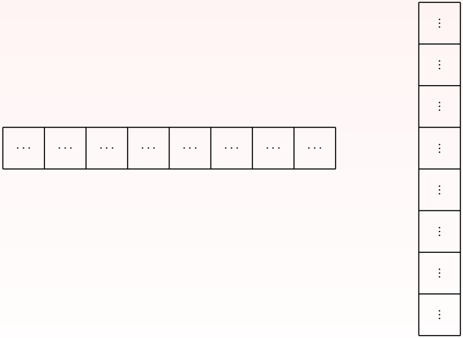
- Proposed by Csanky to get a solution x of the system of linear equations $Ax = b$, regular $A \in R^{n \times n}$ and $x \in R^n, b \in R^n$.
  - The scheme exploits parallelism as much as possible
  - **Does not take into account** actual architectural resources, interconnect and ways to hide latencies.
  - Assume for simplicity $n = 2^l$ for some integer $l \geq 1$.

Parallel operations with dense vectors and matrices (continued)

7. Extreme parallelism

Step 1: Compute powers of A: $A^2, A^3, \dots A^{n-1}$

- Compute A^2: all entries computed in parallel



Parallel operations with dense vectors and matrices (continued)

7. Extreme parallelism

- Compute then A^4, A^8; then the remaining powers
- it can be done with the parallel prefix type procedure called **repeated squaring** – a variation of the techniques outlined above. This results in two steps of **logarithmic complexity only**.
- Each matrix-matrix multiplication has $\Theta(\log_2 n)$ complexity (all products rows and columns are computed in parallel) and the logarithmic term caused by the reduction steps.
- Altogether: Step 1 has $\Theta(\log^2 n)$ complexity.

Parallel operations with dense vectors and matrices (continued)

7. Extreme parallelism

Step 2: Compute traces $s_k = tr(A^k)$ of the powers.

- This is a straightforward computation with $\Theta(\log_2 n)$ complexity (just the reductions).

Parallel operations with dense vectors and matrices (continued)

7. Extreme parallelism

Step 3: Solve Newton identities for coeffs of the characteristic polynomial (Fadeev-LeVerrier algorithm).

- Consider the characteristic polynomial in λ in the form

$$\det(\lambda I_n - A) = p(\lambda) = \sum_{k=0}^{n} c_k \lambda^k \tag{60}$$

- We know that $c_n = 1$ and $c_0 = (-1)^n \det A$. The remaining coefficients c_i can be computed by solving the following triangular system.

$$\begin{pmatrix} 1 & & & & & \\ s_1 & 2 & & & & \\ \vdots & \ddots & \ddots & & & \\ s_{m-1} & \ddots & s_1 & m & & \\ \vdots & \ddots & \ddots & \ddots & \ddots & \\ s_{n-1} & \dots & s_{m-1} & \dots & s_1 & n \end{pmatrix} \begin{pmatrix} c_{n-1} \\ c_{n-2} \\ \vdots \\ c_{n-m} \\ \vdots \\ c_0 \end{pmatrix} = - \begin{pmatrix} s_1 \\ s_2 \\ \vdots \\ s_m \\ \vdots \\ s_n \end{pmatrix}$$

Parallel operations with dense vectors and matrices (continued)

7. Extreme parallelism

- **Scaled** lower triangular matrix such that its diagonal is unit can be directly written as follows. The scaling is denoted by changing s_i to $\hat{s}_i$.

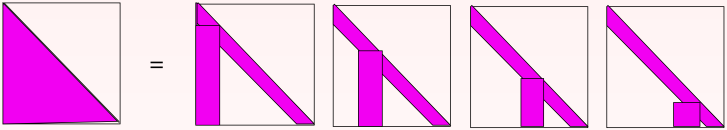
$$\begin{pmatrix} 1 & & & & \\ \hat{s}_1 & 1 & & & \\ \hat{s}_2 & \hat{s}'_1 & 1 & & \\ \vdots & \ddots & \ddots & \ddots & \\ \hat{s}_{n-1} & \dots & \dots & \hat{s}_1 & 1 \end{pmatrix} =$$

$$\begin{pmatrix} 1 & & & & \\ \hat{s}_1 & 1 & & & \\ \hat{s}_2 & 0 & \ddots & & \\ \vdots & \vdots & \ddots & 1 & \\ \hat{s}_{n-1} & 0 & \dots & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & & & & \\ 0 & 1 & & & \\ 0 & \hat{s}'_1 & \ddots & & \\ \vdots & \vdots & \ddots & 1 & \\ 0 & \hat{s}'_{n-2} & 0 & 0 & 1 \end{pmatrix} \cdots \begin{pmatrix} 1 & & & & \\ 0 & 1 & & & \\ 0 & 0 & \ddots & & \\ \vdots & \vdots & \ddots & 1 & \\ 0 & 0 & \dots & \hat{s}''_1 & 1 \end{pmatrix}$$

Parallel operations with dense vectors and matrices (continued)

7. Extreme parallelism

Once more, schematically.



- Inverses of the elementary factors are obtained by changing the sign of their off-diagonal entries.
- Explicit construction of the inverse: multiply the individual inverses in the reversed order.
- This multiplication can be based on parallel prefix algorithm and we obtain $\Theta(\log_2^2 n)$ complexity as in the Step 1.

Parallel operations with dense vectors and matrices (continued)

7. Extreme parallelism

Step 4: Compute the inverse using the Cayley-Hamilton theorem and evaluate the solution by matrix/vector

$$A^{-1} = \frac{A^{n-1} + c_{n-1}A^{n-2} + \ldots + c_1 I}{-c_0}$$

- The complexity of this step as well as of multiplying of the right-hand side to get the solution is $\Theta(\log_2 n)$.

Unfortunately, the solver outlined above in the four steps is very unstable and not useful in practice.

Parallel operations with dense vectors and matrices (continued)

8. Alternative parallel matrix-matrix multiplications

- Possible parallelism versus accuracy
- Consider the following two possibilities of matrix-matrix multiplication: standard versus a proposal by Strassen

$$C = AB,\ \ A, B, C \in R^{2^k \times 2^k}$$

$$A = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix},\ B = \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix},\ C = \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

$$A_{i,j}, B_{i,j}, C_{i,j} \in R^{2^{k-1} \times 2^{k-1}}$$

Parallel operations with dense vectors and matrices (continued)

8. Alternative parallel matrix-matrix multiplications

- Standard computation: 8 multiplications

$$
\begin{aligned}
C_{1,1} &= A_{1,1}B_{1,1} + A_{1,2}B_{2,1} \\
C_{1,2} &= A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\
C_{2,1} &= A_{2,1}B_{1,1} + A_{2,2}B_{2,1} \\
C_{2,2} &= A_{2,1}B_{1,2} + A_{2,2}B_{2,2}
\end{aligned}
$$

Parallel operations with dense vectors and matrices (continued)

8. Alternative parallel matrix-matrix multiplications

- Computation according to Strassen:

$$\begin{aligned}
M_1 &= (A_{1,1}+A_{2,2})(B_{1,1}+B_{2,2})\\
M_2 &= (A_{2,1}+A_{2,2})B_{1,1},\ M_3 = A_{1,1}(B_{1,2}-B_{2,2})\\
M_4 &= A_{2,2}(B_{2,1}-B_{1,1}),\ M_5 = (A_{1,1}+A_{1,2})B_{2,2}\\
M_6 &= (A_{2,1}-A_{1,1})(B_{1,1}+B_{1,2}),\ M_7 = (A_{1,2}-A_{2,2})(B_{2,1}+B_{2,2})\\
C_{1,1} &= M_1+M_4-M_5+M_7,\ C_{1,2} = M_3+M_5\\
C_{2,1} &= M_2+M_4,\ C_{2,2} = M_1-M_2+M_3+M_6
\end{aligned}$$

- Only 7 multiplications: this is what is important for the complexity

Parallel operations with dense vectors and matrices (continued)

8. Alternative parallel matrix-matrix multiplications

- More ways to pad the matrices by zeros to apply the Strassen's multiplication.
- Complexity: $n^{\log_2 7} \approx n^{2,807}$ instead of $n^{\log_2 8} = n^3$.
- Accuracy bounds of the classical multiplication and the Strassen's one can be written as
  - $$|fl_{conventional}(AB) - AB| \leq n\epsilon|A||B|$$
  - $$||fl_{Strassen}(AB) - AB||_M \leq f(n)\epsilon||A||_M||B||_M, \ f(n) \ \approx O(n^{3.6}),$$

  respectively, where
  $$||X||_M = \max_{i,j}|x_{ij}|.$$
- Clearly, the Strassen's multiplication **can be** significantly more inaccurate. Appropriate scaling can improve the result.

Highly parallel operations with Monte Carlo methods

9. High parallelism as the Monte Carlo method

- Monte Carlo (MC) methods: based on **repeated independent and random sampling**. Easy to parallelize.
- First example: integration over a bounded interval

$$F = \int_a^b f(x)dx. \tag{61}$$

- The result can be "well" approximated by

$$F = (b-a)\mathbb{E}f \approx (b-a)\frac{1}{n}\sum_{i=1}^{n} f(x_i)$$

x_i, $i = 1, \dots, n$: uniformly distributed random numbers from the interval

- $\mathbb{E}f$: the expectation value of the function f from the interval.
- Here one-dimensional example, can be generalized.

Highly parallel operations with Monte Carlo (continued)

9. High parallelism as the Monte Carlo method

- Second example: **MC computation of π**
- Consider a unit square with an inscribed circle.
- The ratio R of the area of the circle S (of the radius 1) with respect to the area of the square is given by

$$R \approx \frac{area_{circle}}{area_{square}} = \frac{\pi 1^2}{2^2} = \frac{\pi}{4}$$

- The following algorithm can be thus used to compute an approximate value of π in parallel.

Highly parallel operations with Monte Carlo (continued)

9. High parallelism as the Monte Carlo method

Algorithm

MC computation of π

Input: *Samples count $nsample$. Random points in the unit square.*

Output: *Approximate value of π.*

```
for $i = 1 : nsample$ do in parallel
    Choose a random point in the square
end do
    Count the ratio $R$: random points inside the circle over $nsample$.
return $\pi \approx 4 \times R$
```

Outline

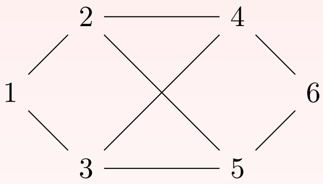
1. Foreword
2. Computers, computing, communication
3. Parallel computing
4. Parallel processing and us - parallel programming
5. Parallel computer architectures: hardware and classification
6. Combining pieces together: computational models
   - Uniprocessor model
   - Vector and SIMD model
   - Multiprocessor model
7. Parallelizing problems
8. Sparse data decomposition: graph partitioning
9. Parallel and parallelized algebraic preconditioning

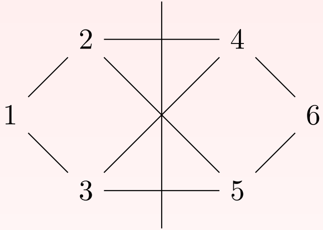
Graph partitioning

Partitioning of sparse data: general comments

- Standard array-based decomposition schemes mentioned above are often efficient in case of **dense matrices, vectors** or **regular (structured patterns)** (like sparse matrix with 1D partitioning - sparse row blocks with similar nonzero counts owned by processors).
- General sparse case" **more sophisticated techniques** to decompose data are needed. Such techniques are called **graph/hypergraph partitioning**.
- More complex ways to partition data correspond to the need to use **more complicated algorithms** like factorization.

Graph partitioning formulation and its goals





separating vertices by edges: **edge separator**

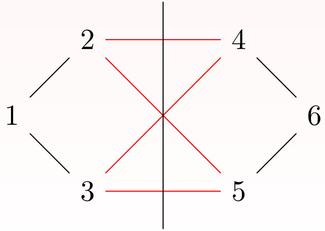
$$\begin{array}{cc} & \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} & \left(\begin{array}{cccccc} * & * & * & & & \\ * & * & & * & * & \\ * & & * & * & * & \\ & * & * & * & & * \\ & * & * & & * & * \\ & & & * & * & \end{array}\right) \end{array} \qquad \begin{array}{cc} & \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} & \left(\begin{array}{cccccc} {\color{red}*} & {\color{red}*} & {\color{red}*} & & & \\ {\color{red}*} & {\color{red}*} & & * & * & \\ {\color{red}*} & & {\color{red}*} & * & * & \\ & * & * & {\color{blue}*} & & {\color{blue}*} \\ & * & * & & {\color{blue}*} & {\color{blue}*} \\ & & & {\color{blue}*} & {\color{blue}*} & \end{array}\right) \end{array}$$

Graph partitioning: separators

- Interfaces separating the detached parts are called separators. A **vertex or edge set** is called **separator** if its removal from the graph **increases number of the graph components**.
- **Vertex separators** V_S and **edge separators** E_S distinguished.
- There are simple **transformation procedures** between the classes of edge and vertex. Nevertheless, straightforward transformations are generally **not advisable**. There exist more involved transformations such that the resulting separators are **approximately optimal** in some sense.

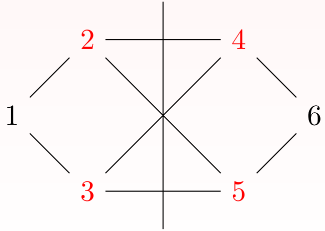
Graph partitioning: separators

- Interfaces separating the detached parts are called separators. A **vertex or edge set** is called **separator** if its removal from the graph **increases number of the graph components**.
- **Vertex separators** V_S and **edge separators** E_S distinguished.
- There are simple **transformation procedures** between the classes of edge and vertex. Nevertheless, straightforward transformations are generally **not advisable**. There exist more involved transformations such that the resulting separators are **approximately optimal** in some sense.
-



Graph partitioning: separators

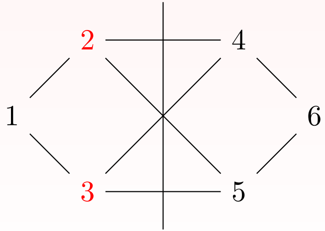
- Interfaces separating the detached parts are called separators. A **vertex or edge set** is called **separator** if its removal from the graph **increases number of the graph components**.
- **Vertex separators** V_S and **edge separators** E_S distinguished.
- There are simple **transformation procedures** between the classes of edge and vertex. Nevertheless, straightforward transformations are generally **not advisable**. There exist more involved transformations such that the resulting separators are **approximately optimal** in some sense.



-

Graph partitioning: separators

- Interfaces separating the detached parts are called separators. A **vertex or edge set** is called **separator** if its removal from the graph **increases number of the graph components**.
- **Vertex separators** V_S and **edge separators** E_S distinguished.
- There are simple **transformation procedures** between the classes of edge and vertex. Straightforward transformations are generally **not advisable**. We also have: $|V_S| \leq |E_S|$, $|E_S| \leq |V_S| \times \max \text{ degree}$.



-

Graph partitioning formulation and its goals

From edge to vertex separator

$$\begin{array}{c|cccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 \\
1 & * & * & * & & & \\
2 & * & * & & * & * & \\
3 & * & & * & * & * & \\
4 & & * & * & * & & * \\
5 & & * & * & & * & * \\
6 & & & & * & * &
\end{array}$$

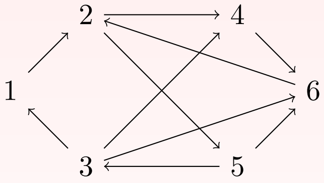
$$\downarrow$$

$$\begin{array}{c|cccccc|}
 & 1 & 4 & 5 & 6 & 2 & 3 \\
1 & * & & & & * & * \\
4 & & * & & * & * & * \\
5 & & & * & * & * & * \\
6 & & * & * & & & \\
2 & * & & * & * & * & \\
3 & * & * & * & & & *
\end{array}$$

The last two rows and columns represent separator in the reordered matrix

Graph partitioning formulation and its goals

- Graph model can be also **directed**. Directed edges may capture **source - destination** relation.



$$\begin{array}{c|cccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
1 & * & * & & & & \\
2 & & * & & * & * & \\
3 & * & & * & * & & * \\
4 & & & & * & & * \\
5 & & & * & & * & * \\
6 & & * & & & & *
\end{array}$$

Graph partitioning formulation and its goals

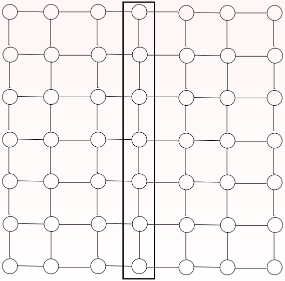
- The **goal** of the graph partitioning is to separate a given matrix or its structure (graph, hypergraph) into
  - parts of **similar sizes** (e.g., roughly equal vertex/edge counts in separated parts)
  - having **small** sizes of graph interfaces that separate these parts (measured, e.g., by vertex/edge counts)
- Here we discuss only partitioning of **undirected graphs** that represent structural models of **symmetric matrices.**
- Further restriction to **bisections** that cut a domain into two parts.
- Bisection can be applied **recursively**.
- Generalized partitioning can use using **weighted** graphs and **hypergraphs**

Graph partitioning: some theoretical results

- The number of all possible partitions for $|V| = n$ is given by

$$\binom{n}{n/2} \approx 2^n \sqrt{2/\pi n} \tag{62}$$

- Consider a graph of a **planar mesh** as in the following figure with $n = k \times k$ for $k = 7$. There is a vertex separator of optimal size having $k = \sqrt{n}$ vertices that is also shown in the figure.



Graph partitioning: some theoretical results

- In 2D (planar) case we have the following theorem.

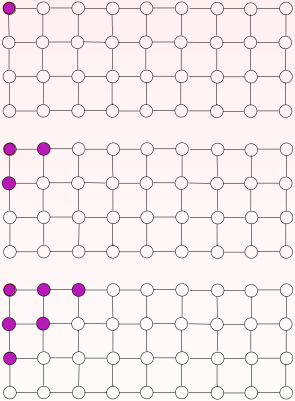
Theorem

Let $G = (V, E)$ *be a* **planar** *graph. Then there is a vertex separator* $S = (V_S, E_S)$ *which divides* V *into two disjoint sets* V_1 *and* V_2 *such that* $\max(|V_1|, |V_2|) \leq 2/3|V|$ *and* $|V_S| = O(\sqrt{|V|})$.

- In case of more general graphs that include under the concept of overlap graphs like 3D finite element grids there are separators with $O(n^{(d-1)/d})$ edges for d-dimensional grids.

Graph partitioning: approaches

1. **Level structure algorithm: based on the breadth-first search**



- The level structure algorithm is a very simple way to find a **vertex separator** but it often does not provide good results. It can be considered as a **preprocessing step** for other algorithms.

Graph partitioning: approaches

1. Level structure algorithm (continued)

- Formally the partitioning of V if constructed.

$$L_0 = \{r\}, \quad L_1 = Adj(L_0) \setminus L_0, \ \ldots, \quad L_k = Adj(L_{k-1}) \setminus L_{k-1} \tag{63}$$

  such that

$$V = \bigcup_{i=0}^{k} L_i,$$

  and all the sets $L_0, \ldots, L_k$ are nonempty. Size of the level set is $k+1$.

- Formally V is cut by its **median set** from $L_0, \ldots, L_k$.

$$V^+ = \bigcup_{i=0}^{j-1} L_i, \ V^- = \bigcup_{i=j+1}^{k} L_i, \ S = V_j. \tag{64}$$

Graph partitioning: approaches

1. Level structure algorithm (continued)

Algorithm

Level structure algorithm (one component)

1. Iterative procedure to find a suitable starting point r

2. Perform the **breadth-first search** *from r*

3. **Sort** *vertices by their levels (distances from r)*

4. Choose as separator the set L_j with distance j from r such that the subgraphs induced by V^+ (vertices of smaller distance from r than j) and V^- (vertices of larger distance from r than j) are **approximately balanced**.

Graph partitioning

1. Level structure algorithm (continued)

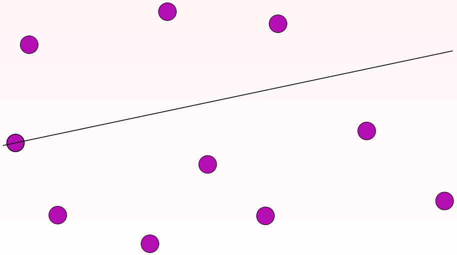
- The Step 1 of the algorithm iteratively finds r that approximately minimizes k. In graph terminology this procedure finds a **pseudoperipheral vertex** of the graph G.
    - **Peripheral** vertex: a vertex of V that has the largest distance from some other node of V
    - Distances measured by lengths of **shortest paths.**

- Having found r, first three steps of the breadth-first search in Algorithm 8.1 were demonstrated above.

Graph partitioning

2. Inertial algorithm: vertices as points in space

- Assuming that the **vertex coordinates are available**.
- This enables to consider graph vertices as **points in** 2D or three-dimensional (3D) **space**
- Its steps for graph partitioning in 2D are schematically given below.



2. Inertial algorithm (continued)

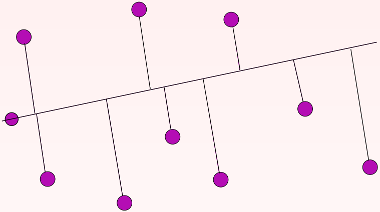
- Step 1: **choose a line** (analogy in 3D would be a plane) and assume the line equation in the form

$$a(x - x_0) + b(y - y_0) = 0,\ a^2 + b^2 = 1.$$

  - (a, b) is the unit vector **perpendicular** to the line. It can be shown like this:
    - ★ consider two points of the line (x_1, y_1) and (x_2, y_2)
    - ★ subtract their equations: $a(x_1 - x_2) + b(y_1 - y_2) = 0 = (a, b) \times (x_1 - x_2, y_1 - y_2)$
  - and $(-b, a)$ is the unit vector **parallel** to the line.
  - Slope of this line is $-a/b$ since we have $(y - y_0) = -a/b(x - x_0)$.
  - The line goes through the **chosen point** (x_0, y_0).

2. Inertial algorithm (continued)

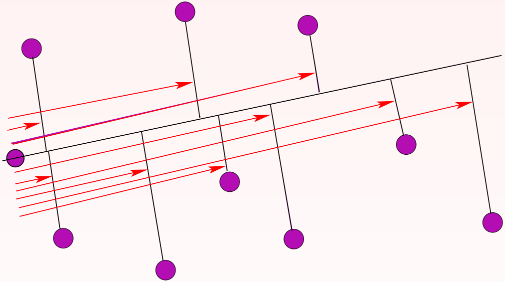
- Step 2: find **projections** of the points to this line (plane in 3D)



  - Distances c_i of the nodes (x_i, y_i) from their projections to the line are given by
$$c_i = |a(x_i - x_0) + b(y_i - y_0)|,$$
  - Or using directly the Pythagorean theorem:
$$c_i^2 = (x - x_i)^2 + (y - y_i)^2 - d_i^2,$$
where
$$d_i = |-b(x_i - x_0) + a(y_i - y_0)|.$$

2. Inertial algorithm (continued)

- Step 3: compute distances d_i of the projections along the line (with respect to the point (x_0, y_0) (line in 3D))



Distances d_i of the projections from (x_0, y_0) are (see previous slide):

$$d_i = \left| -b(x_i - x_0) + a(y_i - y_0) \right|.$$

2. Inertial algorithm (continued)

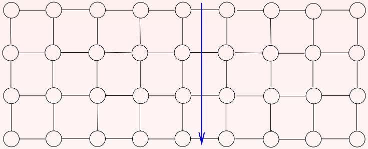
- Step 4: Compute median of these distances and separate the nodes into the two groups by their distances

Line choice

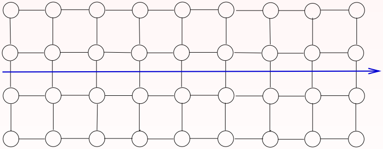
- The line in the 2D inertial algorithm is chosen such that it **minimizes a sum of squares of the projections** c_i^2.
- Considering the points as **mass units**, the line considered as an **axis of rotation** should minimize the **moment of inertia** among all possible lines. Hence the name of the approach.
- See the following figures

Graph partitioning

2. Inertial algorithm (continued)



Large sum: **bad choice** (9 edges in the edge separator).
(The separator is **perpendicular to the blue line**).



Smaller sum: **good choice** (4 edges in the edge separator)

Graph partitioning

2. Inertial algorithm (continued)

- This type of graph partitioning is **simple and flexible**.
- It may give better results than if the separators would be just lines/planes parallel to coordinate directions.
- A disadvantage of this approach is that it considers **separations by a hyperplane ony**. Better partitioning could be obtained without this assumption.

Graph partitioning

2. Inertial algorithm (continued)

Expressing the sum we get (c_i could be directly used as well for the computation instead of the quantities d_i)

$$
\begin{aligned}
\sum_{i=1}^{n} c_i^2 &= \sum_{i=1}^{n}((x_i - x_0)^2 + (y_i - y_0)^2 - d_i^2) \\
&= \sum_{i=1}^{n}[(x_i - x_0)^2 + (y_i - y_0)^2 - (-b(x_i - x_0) + a(y_i - y_0))^2] \\
&= a^2 \sum_{i=1}^{n}(x_i - x_0)^2 + b^2 \sum_{i=1}^{n}(y_i - y_0)^2 + 2ab \sum_{i=1}^{n}(x_i - x_0)(y_i - y_0) \\
&= \begin{pmatrix} a & b \end{pmatrix} M \begin{pmatrix} a \\ b \end{pmatrix},
\end{aligned}
$$

2. Inertial algorithm (continued)

M is defined as follows

$$M = \begin{pmatrix} \sum_{i=1}^{n}(x_i - x_0)^2 & \sum_{i=1}^{n}(x_i - x_0)(y_i - y_0) \\ \sum_{i=1}^{n}(x_i - x_0)(y_i - y_0) & \sum_{i=1}^{n}(y_i - y_0)^2 \end{pmatrix} \tag{65}$$

Finding x_0, y_0, a and b (with $a^2 + b^2 = 1$) such that this quadratic form is minimized is the **total least squares problem**. One can show that we get the minimum if

$$x_0 = \frac{1}{n}\sum_{i=1}^{n} x_i,\ y_0 = \frac{1}{n}\sum_{i=1}^{n} y_i \text{ (center of mass)}. \tag{66}$$

and the vector $\begin{pmatrix} a \\ b \end{pmatrix}$ is normalized eigenvector corresponding to the minimum eigenvalue of M.

- The approach provides an **edge separator.**

3. Spectral partitioning

Define the Laplacian matrix first.

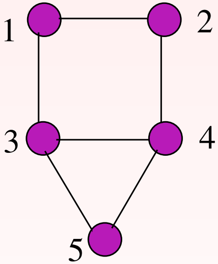
Definition

The Laplacian matrix of an undirected unweighted graph $G = (V, E)$ is L with the entries defined as follows

$$L_{ij} = \begin{cases} deg(i) & (i,j) \in E \; i = j \\ -1 & (i,j) \in E, i \neq j \\ 0 & otherwise \end{cases}$$

3. Spectral partitioning (continued)

Consider the following graph

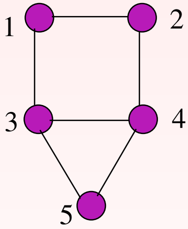


The Laplacian (vertex by vertex symmetric matrix) is

$$L = \begin{pmatrix} 2 & -1 & -1 & & \\ -1 & 2 & & -1 & \\ -1 & & 3 & -1 & -1 \\ & -1 & -1 & 3 & -1 \\ & & -1 & -1 & 2 \end{pmatrix}$$

Graph partitioning

3. Spectral partitioning



Also, $L = A^T A$: A is **oriented incidence** (edge by vertex) matrix of G.

$$A^T = \begin{array}{c|cccccc} & e_1 & e_2 & e_3 & e_4 & e_5 & e_6 \\ \hline 1 & -1 & & -1 & & & \\ 2 & 1 & -1 & & & & \\ 3 & & & 1 & -1 & -1 & \\ 4 & & 1 & & 1 & & -1 \\ 5 & & & & & 1 & 1 \end{array}$$

Graph partitioning

3. Spectral partitioning (continued)

Formal definition of the oriented incidence matrix

- The matrix is edge by vertex

$$A_{ij} = \begin{cases} 1 & \text{i is the first vertex of the edge } j \\ -1 & \text{i is the second vertex of the edge } j \\ 0 & otherwise \end{cases}$$

- The definition does not depend on the edge orientation (setting of 1's and -1's)

Graph partitioning

3. Spectral partitioning (continued)

Theorem

All eigenvalues of $L = (l_{ij}) \in R^{n \times n}$ are nonnegative. Moreover, L is singular. Consequently, L is positive semidefinite.

Proof.

Gershgorin circle theorem: every eigenvalue λ_j lies within a disc centered in some l_{ii}. The eigenvalues are real and radii r_i of the discs are **equal to the distance of their center from zero at most** because

$$r_i = \sum_{j,\ l_{ij} \neq 0, j \neq i} |l_{ij}|. \tag{67}$$

Then all eigenvalues are at least 0. Also $\begin{pmatrix} 1 & \dots & 1 \end{pmatrix}^T L = 0$. This implies that L is positive semidefinite.

□

3. Spectral partitioning (continued)

Theorem

Multiplicity of the zero eigenvalue of the Laplacian $L \in R^{n \times n}$ *of the graph* $G = (V, E)$ *is equal the number of the connected components of* G.

Proof.

L is symmetric and thus diagonalizable. Then multiplicity of zero as a root of the characteristic polynomial is equal to the dimension of its nullspace. Suppose that x is a normalized eigenvector corresponding to the zero eigenvalue. That is, $Lx = 0$. $L = A^T A$ implies that $Ax = 0$ and this implies that for adjacent vertices i and j of V must be $x_i = x_j$ whenever they are adjacent since A is the **edge by vertex** matrix. That is $x_i = x_j$ if there is a path between i and j that is, whenever i and j are in the same component of G. Consider the components of G and their characteristic vectors. Clearly, they are independent and they form the basis of the nullspace of L. □

Graph partitioning

3. Spectral partitioning (continued)

Observation

The eigenvector corresponding to the zero eigenvalue of the Laplacian $L \in R^{n \times n}$ of the connected graph is $x = (1, \dots, 1)^T / \sqrt{n}$.

-

- Nonnegativity of the eigenvalues of L is also visible from the quadratic form with the Laplacian L of $G = (V, E)$. This form can be written due to its relation to the graph incidence matrix as

$$x^T L x = x^T A^T A x = \sum_{\{i,j\} \in E} (x_i - x_j)^2. \tag{68}$$

Graph partitioning

3. Spectral partitioning (continued)

Example

Consider a graph (V, E) with even $|V|$ chosen as $V = \{1, \ldots, n\}$. Let V be partitioned into V^+ and V^- of the same size. Consider the vector $x \in R^n$ with $x_i = 1$ for $i \in V^+$ and $x_i = -1$ otherwise. Then the number of edges that connect V^+ and V^- is equal to $1/4 x^T L(G) x$.

Proof.

We can write

$$\begin{aligned} x^T L(G) x &= \sum_{(i,j) \in E} (x_i - x_j)^2 = \sum_{(i,j) \in E,\ i \in V^+,\ j \in V^-} (x_i - x_j)^2 = \\ &= 4 * \text{number of edges between } V^+ \text{ and } V^- \end{aligned} \tag{69}$$

since the quadratic terms contribute by 4 if the nodes of an edge are from different sets and 0 otherwise. □

Graph partitioning

3. Spectral partitioning (continued)

If G has one component only, the **second smallest eigenvalue** of L is nonzero. Denote it by μ.

- The Courant-Fischer theorem states that

$$\mu = \min\{x^T Lx \mid x \in \mathbb{R}^n \wedge x^T x = 1 \wedge x^T (1, \ldots, 1)^T = 0\}. \qquad (70)$$

- As we saw, this expresses the $\text{size of the edge partitioner}$ in the example.
- Can be proved for general undirected graphs and discrete setting $x_i = 1, -1,\ sum(x_i) = d, |d| < n$.
- A question is how this eigenvector $x(\mu)$ for the eigenvalue μ, often called the Fiedler vector, can be approximated.

Graph partitioning

3. Spectral partitioning (continued)

Consequently, the graph bisection problem for a graph with **even number of nodes** can be casted as

$$\begin{array}{rcl} Minimize & & f(x) = \frac{1}{4} x^T L x \\ subject\ to & & x \in \{\pm 1\}^n \\ x^T(1,\ldots,1)^T & = & 0. \end{array}$$

This represents a discrete optimization problem that can be solved approximatively using the following relaxed form.

Find $x \in R^n$ minimizing

$$x^T L x \tag{71}$$

such that $x^T(1,\ldots,1)^T \approx 0$ and $x^T x = 1$.

Graph partitioning

3. Spectral partitioning (continued)

In practice, Lanczos algorithm can be used and the computational scheme is as follows (connected graph)

Algorithm

Spectral graph bisection.

1. Find μ and $x(\mu)$ of L
2. Sort the vertices by x_i and split them by their **median** value

This **spectral partitioning** approach may be expensive, but it can provide high-quality partitions. As the previous approach, it finds an **edge separator.**

Graph partitioning

4. General multilevel partitioning

- The basic principle of the general multilevel partitioning is to apply the following three steps to where the middle one is applied **recursively**
  - Coarsen the graph,
  - Partition the graph,
  - Interpolate separator and refine the graph.

Graph partitioning

4. General multilevel partitioning (continued)

- Coarsening phase **collapses nodes** that define the matching edges into **coarse** nodes. Each edge can have weight that correspond to a number of original edges it represents.
- Vertices can have **weights** expressing how many vertices have been collapsed into them.
- There are more ways how such weights can be exploited.

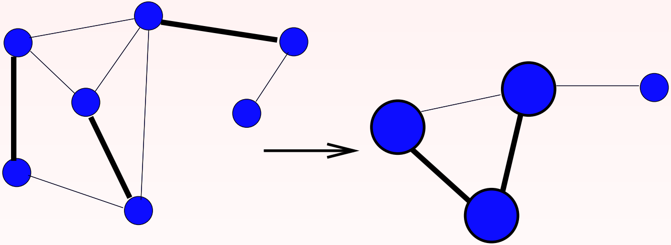
Graph partitioning

4. General multilevel partitioning

- There are more ways to do the coarsening. A popular **matching-based** coarsening is based on finding matchings of weighted graphs.
    - **Matching** of a graph $G = (V, E)$ is a subgraph $(\tilde{V}, \tilde{E})$ with $\tilde{E} \subseteq, \tilde{V} = V(\tilde{E})$
    - **Maximal matching** is a matching $(\tilde{V}, \tilde{E})$ such that there is no matching $(\hat{V}, \hat{E})$ of G such that $\tilde{E} \subset \hat{E}, \hat{V} = V(\hat{E})$
    - Construction by greedy algorithms (can be increased to maximum matchings using augmenting paths)
    - On the outer level, there are multiple variations of the basic approach called, for example, **multilevel nested dissection** or **multilevel spectral partitioning**. Another coarsening possibility can be based on recursive search of independent sets in the graph sequence.

Graph partitioning

4. General multilevel partitioning (continued)



Graph partitioning

5. Iterative KL greedy refinement by Kernighan and Lin (1970)

- One of the first approaches
  - Based on **local searches** and starting with an **initial** (possibly trivial) partitioning.
  - Targets **weighted** graphs.

Graph partitioning

5. Iterative KL greedy refinement

- Start with a graph $G = (V, E)$ having edge weights $w : E \to \mathbb{R}^+$ and with some initial partitioning $V = V^+ \cup V^-$. Consider its **cost functional** given by

$$COST = \sum_{a \in V^+,\ b \in V^-} w(a, b).$$

The goal is to improve this partitioning.

- The initial and any other partition **can be improved** if we find $X \subset V^+$ and $Y \subset V^-$ such that the partition formed as

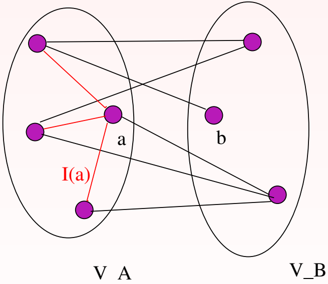
$$V = (V^+ \cup Y \setminus X) \cup (V^- \cup X \setminus Y) \tag{72}$$

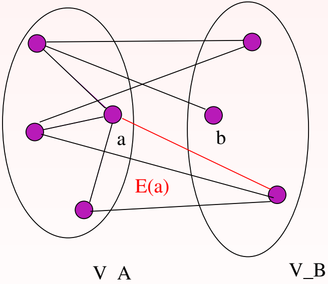
reduces the total cost of edges between V^+ and V^-.

Graph partitioning

5. Iterative KL greedy refinement

Denote by $E(x)$, $I(x)$ the **external** and the **internal** cost equal to the sum of weights of $x \in V$ in the given parts of the partition.





Graph partitioning

5. Iterative KL greedy refinement

- How a gain in the COST can be found and exploited?
- Consider a simplified case with

$$a \in V^{+} \quad \text{and} \quad b \in V^{-}, \tag{73}$$

  based on exchanging this pair of vertices a and b.
- Moving **simultaneously** a to V^{-} and b to V^{+} results in the updated sets $\tilde{V}^{+} = V^{+} \setminus \{a\} \cup \{b\}$ and $\tilde{V}^{-} = V^{-} \setminus \{b\} \cup \{a\}$.
- Denote for our simple case $X = \{a\}$ and $Y = \{b\}$.

Graph partitioning

5. Iterative KL greedy refinement (continued)

- Exchanging a and b: the COST functional changes to $\widehat{COST}$

$$\widehat{COST} = COST + I(a) - E(a) + w(a,b) + I(b) - E(b) + w(a,b) \equiv COST - gain(a,b) \tag{74}$$

where $gain(a,b)$ for $a \in V^+$ and $b \in V^-$ is defined by

$$E(a) - I(a) + E(b) - I(b) - 2w(a,b).$$

- Note that the weight of a possible edge must be subtracted from the $gain$.
- The algorithm can be then formally written as follows where GAIN denotes the sum of gains between pairs of vertices.

Graph partitioning

5. Iterative KL greedy refinement (continued)

Algorithm

Partitioning improvement by Kernighan and Lin

```
1. Compute COST of the initial partition
2. do until $GAIN \leq 0$
    3. forall nodes $x \in V$ compute $E(x), I(x)$
    4. unmark all nodes
    5. do while there are unmarked nodes
        6. find a suitable pair $a, b$ of vertices maximizing $gain(a, b)$
        7. mark $a, b$ (to be excluded from further exchanges in this loop)
    8. end do while
    9. find $GAIN$ maximizing the partial sum of gains computed in the loop
    10. if $GAIN > 0$ then update the partition, $COST = COST - GAIN$
11. end do
```

Graph partitioning

5. Iterative KL greedy refinement (continued)

- Often used to **improve partitions** from other algorithms. It usually converges in a few major steps. Each of them has a complexity $O(n^3)$.
- Fiduccia and Mattheyses (1982) have shown that this complexity can be improved to $O(|E|)$.

6. Nested dissection (framework)

Algorithm

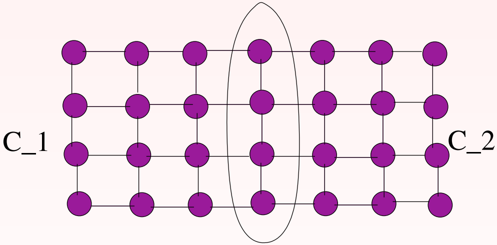
Nested dissection: framework for partitioning algorithms

1. *Find a bisection (dissection) (possible approaches explained)*
2. *Reorder matrix numbering nodes in the* **separator last**
3. *Perform the previous two steps* **recursively**

Graph partitioning

6. Nested dissection (continued)

- Separator in a simple mesh

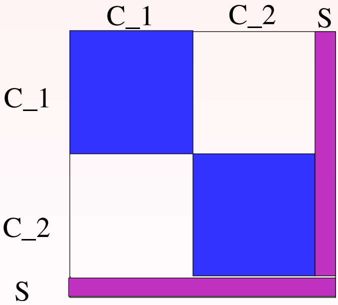


Vertex separator S

- Nested dissection (ND) matrix after the first level of the recursion

6. Nested dissection (continued)

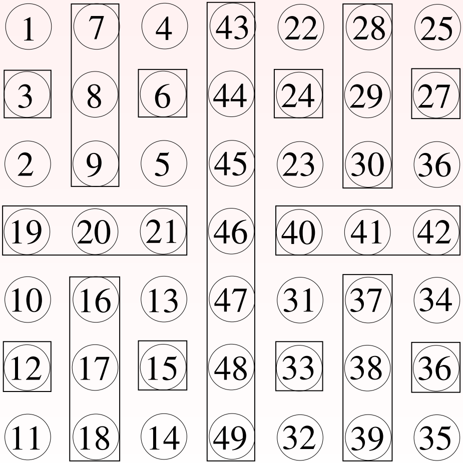
- ND matrix structure



Graph partitioning

6. Nested dissection (continued)

- ND algorithm after more levels of recursion



1	7	4	43	22	28	25
3	8	6	44	24	29	27
2	9	5	45	23	30	36
19	20	21	46	40	41	42
10	16	13	47	31	37	34
12	17	15	48	33	38	36
11	18	14	49	32	39	35

Graph partitioning

6. Nested dissection

- **Modern nested dissections** are based on various graph partitioners that enable to partition very general graphs. In this way, the nested dissection can be understood as an outer algorithmic **framework** that may include other algorithms that guarantee that the separated
    - components have very similar sizes and the
    - separator is small.

- Nested dissection implies **Modern local reorderings** that minimize fill-in in factorization algorithms are often on a few steps of an incomplete nested dissection.

7. Problems with the symmetric model

- What about partitioning of **nonsymmetric graph structures**?
  - Edge cuts are **not proportional** to the total communication volume.
  - **Latencies** of messages typically more important than the volume.
  - In many cases, **minmax problem** should be considered instead (minimizing maximum communication cost).
  - Specific nonsymmetric partitions might and can be considered **(bipartite graph model, hypergraph model)**.
  - General rectangular problem can be considered.
  - Partitioning **in parallel** (there are papers and codes), not only partitioning **for** parallel computations.

Sparse linear algebraic solvers

- Consider solving systems of linear equations

$$Ax = b$$

  where A is **large** and A is **sparse**.
- Two basic classes of methods for solving systems of linear algebraic equations roughly classified as **direct methods** and **iterative methods**.
- Each of these classes has its specific **advantages** and **disadvantages**.
- In the other words, both classes can be considered as complementary approaches as follows:
  - Iterative methods can make the solution obtained from a direct solver **more accurate** by performing a few additional iterations to improve the accuracy.
  - Approximate **direct factorizations** often used as auxiliary procedures (preconditioners) to make iterative methods more efficient.

Sparse linear algebraic solvers

Parallel direct and iterative methods

- More different tasks inside both approaches
  - Parallelizing **matvecs**
  - Parallelizing **matmats**
  - Parallelizing **factorizations**
  - **Avoiding** factorizations by some tricks
  - Tasks different for **direct** and **iterative** methods.
  - Non-uniform data decomposition: **graph partitioning** is a useful tool.

Parallel direct methods: summary

- Two steps that have been sometimes (traditionally) merged together.
- The first step of a direct method is **factorization** of the system matrix A.
- The second step of a direct method is the **solve** step (forward and back solves).
- Historically, the oldest approaches to factorize sparse systems were based on specific paradigms.
- Using these paradigms unlocked by **special reorderings**

$$Ax = b \tag{75}$$

is transformed into

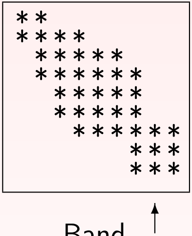
$$P^T AP(P^T x) = P^T b. \tag{76}$$

Parallel direct methods: 1. classical matrix shapes

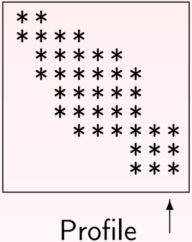
- Sometimes reorderings for parallelism and for efficiency **coincide.**
- Reorderings: **banded and envelope (profile) paradigms**. Basic idea behind them is to find a **reordering** of the system matrix A such that its nonzero entries are moved as close to the matrix diagonal as possible to get a reordered matrix P^TAP.
- An optimum reordering that minimizes some measure like the fill-in in the factorization cannot be found since the corresponding combinatorial decision problem is generally **NP-complete**).
- Therefore, reorderings are typically based on heuristics.
- The **band** and **profile** types of reorderings often lead to **denser factors** then general **fill-in minimizing** reorderings. But stacking nonzeros towards the diagonal may result in **better cache reuse**. Examples of possible nonzero shapes as a result of the mentioned reorderings are depicted below.

Sparse linear algebraic solvers

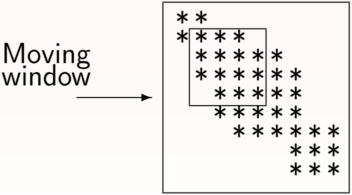
Parallel direct methods: 1. classical matrix shapes



Band



Profile



Frontal method - dynamic band

Parallel direct methods: 1. classical matrix shapes

- Ways to parallelize can be based on the following **principles or their combination**.
    - **decomposing the matrix by diagonals**,
    - using a related **block diagonal structure**,
    - exploiting a **block tridiagonal shape**.

Sparse linear algebraic solvers

1. Parallel direct methods: general shapes

- Generic scheme based (most often) on the LU or Cholesky factorization.
- Assume the system matrix sparse **with a general sparsity structure**
- Obstacle - short row (column) vectors.
- Solution: Hardware support for **vector processing indirectly addressed vectors**.
- Developed in the early days of computer development and it is often called **hardware gather-scatter**.
- Consequently, generally sparse data structures can be **modestly vectorized** with a reasonable asymptotic speed. (Though not often close to R_∞.)

Sparse linear algebraic solvers

Parallel direct methods: 2. general matrix shapes

- Efficiency can be enhanced by matrix reorderings.
- All of this may still not be enough.
- Parallel computation means considering: communication and (balanced) decomposition
  - Task-based decomposition: **fan-in and fan-out** approaches.
  - Data-based decomposition: **cache-efficient block processing**, **tree parallelism**
  - Both we will demonstrate using LDLT factorization

Sparse linear algebraic solvers

Parallel direct methods: 2a. task-based parallelization

- Need to combine **computational dependency** with **communication dependency**
- **demand driven** approach also called **fan-in** (left-looking) approach
  - the nodes wait for data until a computational task is ready
- **data driven** approach also called the **fan-out** (right-looking) approach
  - each computational unit sends data as soon as they are obtained

Parallel direct methods: 2a. task-based parallelization

Algorithm (Simplified sparse LDLT factorization (left-looking))

Input: *Sparse symmetric factorizable matrix A; sparsity pattern of L.*

Output: *Factors $L = \{l_{ij}\}$ and $D = diag\{d_1, \ldots, d_n\}$ of A.*

```
for $j = 1 : n$ do
    for $k \in \{k < j \,|\, l_{jk} \neq 0\}$ do
        for $i \in \{i \geq j \,|\, l_{ik} \neq 0\}$ do
            $a_{ij} \leftarrow a_{ij} - l_{ik} l_{jk}$
        end for
    end for
    $d_j = l_{jj}$
    for $i \in \{i > j \,|\, a_{ij} \neq 0\}$ do
        $l_{ij} \leftarrow a_{ij} d_j^{-1}$
    end for
end for
```

Parallel direct methods: 2a. task-based parallelization

- Define column oriented subtasks
  - $cdiv(j)$ for $1 \leq j \leq n$ denotes dividing column j of the factorization by the diagonal entry d_j
  - $cmod(j,k)$ for $1 \leq k < j \leq n$: modification of column j by column k
  - Communication dependency is expressed by the following precedence relations shown here for the column indices $1 \leq k < j < i \leq n$.

$$cmod(j,k) \longrightarrow cdiv(j) \longrightarrow cmod(i,j).$$

  - These precedence relations: graph form of **communication**
  - Complete update of a column j by columns k such that $k \in row_L\{j\}$ split into a sum of aggregate updates. Each of them correspond to a different processor involved in the update of the column j. An aggregate update from a processor $p \in procs$ involved in updating column j we will denote by $u[j,p]$ and we have

$$\sum_{k \in row_l\{j\}} l_{jk} L_{j:n,k} = \sum_{p \in procs} u[j,p].$$

Sparse linear algebraic solvers

Algorithm (Demand driven (fan-in) Cholesky factorization)

```
for each processor $p$ do                                ▷ Loop of processors processed in parallel
    $mycols(p) = \{j \,|\, map(j) = p\}$
    for $j = 1 : n$ do
        if $j \in mycols(p)$ or $col(j) \cap mycols(p) \neq \emptyset$ then
            $u = 0$                                      ▷ Initialize $L_{j:n,j}$ in a local auxiliary vector
            for $k \in col(j) \cap mycols(p)$ do
                $u = u + l_{jk} L_{j:n,k}$
                if $map(j) \neq p$ then Send $u$ to $map(j)$
                else
                    Incorporate $u$ in $L_{j:n,j}$
                    while Aggregate updates $u[j,q]$ for $q \neq p$ need to be received do
                        Receive aggregate and incorporate into $L_{j:n,j}$
                    end while
                    Set $d_j = l_{jj}$ and scale $L_{j:n,j} = L_{j:n,j} d_j^{-1}$
                end if
            end for
        end if
    end for
end for
```

Sparse linear algebraic solvers

Parallel direct methods: 2a. task-based parallelization

Algorithm (Simplified sparse LDLT factorization (right-looking))

Input: *Sparse symmetric factorizable matrix A; sparsity pattern of L.*

Output: *Factors $L = \{l_{ij}\}$ and $D = diag\{d_1, \ldots, d_n\}$ of A.*

```
for $k = 1 : n$ do
    $d_k = a_{kk}$
    for $i \in \{i > k \,|\, a_{ik} \neq 0\}$ do
        $l_{ik} \leftarrow a_{ik} d_k^{-1}$
    end for
    for $j \in \{j > k \,|\, l_{jk} \neq 0\}$ do
        for $i \in \{i \geq j \,|\, l_{ik} \neq 0\}$ do
            $a_{ij} \leftarrow a_{ij} - l_{ik} l_{jk}$
        end for
    end for
end for
```

Algorithm (Data driven (fan-out) Cholesky factorization)

```
for each processor $p$ do                                ▷ Loop of processors processed in parallel
    Define $mycols(p) = \{j \,|\, map(j) = p\}$, $procs(j) = \{map(k) \,|\, k \in col_L\{j\}\}$
    for $j \in mycols(p)$ do
        if $j$ is a leaf node in $\mathcal{T}(A)$ then
            Set $d_j = l_{jj}$ and scale $L_{j:n,j} = L_{j:n,j} d_j^{-1}$
            Send $L_{j:n,j}$ to all members of $procs(j)$
            Set $mycols(p) = mycols(j) \setminus \{j\}$
        end if
        while $mycols(p) \neq \emptyset$ do
            Wait until any column of $L$, say $L_{k:n,k}$ is received
            for $j \in row_L\{k\} \cap mycols(p)$ do
                Set $L_{j:n,j} = L_{j:n,j} - l_{jk} L_{j:n,k}$
                if $L_{j:n,j}$ is fully updated then
                    Set $d_j = l_{jj}$, scale $L_{j:n,j} = L_{j:n,j} d_j^{-1}$, send $L_{j:n,j}$ to $procs(j)$
                    Set $mycols(p) = mycols(j) \setminus \{j\}$
                end if
            end for
        end while
    end for
end for
```

Sparse linear algebraic solvers

Parallel direct methods: 2a. task-based parallelization

- Since both of the fan-in and fan-out approaches are **different and complementary** such that depending on a problem, one of them may be more useful than the other, it is possible to combine them into a blended fan-both implementation.
- There exists another type of factorization where considering data-based partitioning is more natural
- To run the factorization in parallel, some of them need to parallelize also substitution steps.

Sparse linear algebraic solvers

Parallel direct methods: 2b. data-based parallelization

- Blocks in the original matrix
- Blocks created throught: **supernodes**. The idea of a supernode is to **group together columns with the same nonzero structure**, so they can be treated as a dense matrix for storage and computation.
- Supernodes influenced by specific permutations
- Contemporary Cholesky/LU factorizations strongly based on the concept of **supernodes** or **panels** that represent blocks that are **dense in the factor**.
- Such dense block can be efficiently found before the factorization actually starts, or, computed on-the-fly in case of pivoting.

Parallel direct methods: 2b. data-based parallelization

- A specific block-based factorization right-looking algorithm: the **multifrontal** method. Combines two effects:
  - the tree parallelism with
  - parallel processing of the dense blocks.

Sparse linear algebraic solvers: multifrontal method

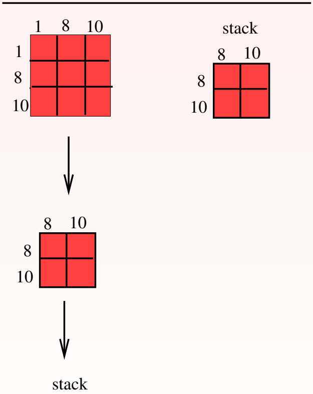
Parallel direct methods: 2b. data-based parallelization

- Right-looking (submatrix) method
- Does not form the Schur complement directly. Instead, the **updates** are moved to a **stack** as dense matrices and **used when needed**.
- The processing order is based on the **elimination tree**
- We will see that in order to have the needed **updates** at the stack top, postordering is needed.
- **Specific postorderings** used to minimize the needed amount of memory.
- Now example, properties repeated once more later.

Sparse linear algebraic solvers: multifrontal method

Parallel direct methods: 2b. data-based parallelization

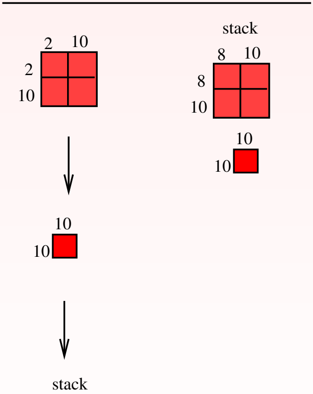
$$\begin{pmatrix}
* & & & & & & & * & & * \\
& * & & & & & & & & * \\
& & * & & & & * & & & * \\
& & & * & & & * & & & \\
& & & & * & * & & & * & \\
& & & & * & * & & & f & * \\
& & * & * & & & * & & * & f \\
* & & & & & & & * & & * \\
& & & & * & f & * & & * & f \\
* & * & * & & & * & f & * & f & *
\end{pmatrix}$$



Sparse linear algebraic solvers: multifrontal method

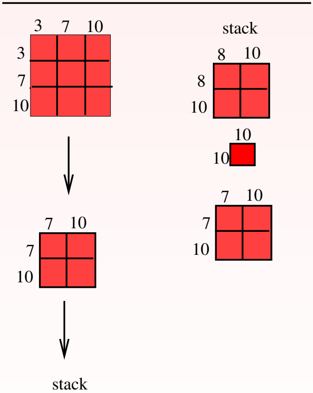
Parallel direct methods: 2b. data-based parallelization

$$\begin{pmatrix}
* & & & & & & & * & & * \\
 & * & & & & & & & & * \\
 & & * & & & & * & & & * \\
 & & & * & & & * & & & \\
 & & & & * & * & & & * & \\
 & & & & * & * & & & f & * \\
 & & * & * & & & * & & * & f \\
* & & & & & & & * & & * \\
 & & & & * & f & * & & * & f \\
* & * & * & & & * & f & * & f & *
\end{pmatrix}$$



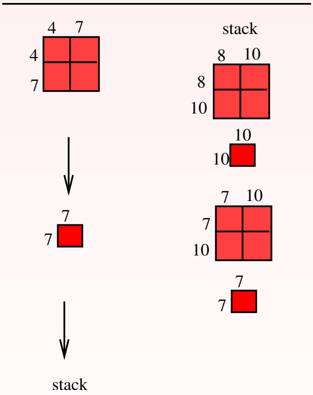
Parallel direct methods: 2b. data-based parallelization

$$\begin{pmatrix}
* & & & & & & & * & & * \\
& * & & & & & & & & * \\
& & * & & & & * & & & * \\
& & & * & & & * & & & \\
& & & & * & * & & & * & \\
& & & & * & * & & & f & * \\
& & * & * & & & * & & * & f \\
* & & & & & & & * & & * \\
& & & & * & f & * & & * & f \\
* & * & * & & & * & f & * & f & *
\end{pmatrix}$$



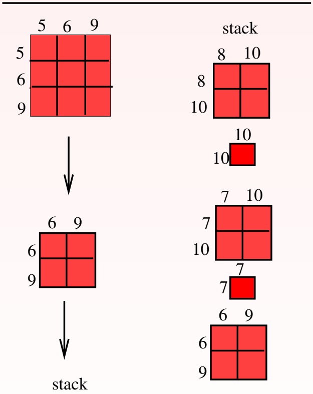
Direct methods: Multifrontal method

$$\begin{pmatrix}
* & & & & & & & * & & * \\
& * & & & & & & & & * \\
& & * & & & & * & & & * \\
& & & * & & & * & & & \\
& & & & * & * & & & * & \\
& & & & * & * & & & f & * \\
& & * & * & & & * & & * & f \\
* & & & & & & & * & & * \\
& & & & * & f & * & & * & f \\
* & * & * & & & * & f & * & f & *
\end{pmatrix}$$



Parallel direct methods: 2b. data-based parallelization

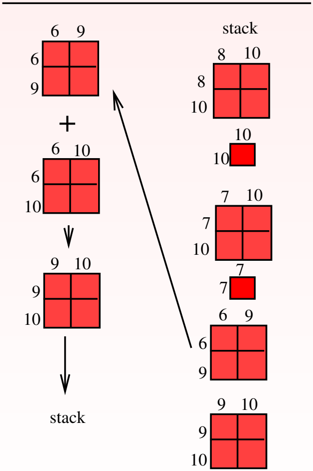
$$\begin{pmatrix}
* & & & & & & & * & & * \\
& * & & & & & & & & * \\
& & * & & & & * & & & * \\
& & & * & & & * & & & \\
& & & & * & * & & & * & \\
& & & & * & * & & & {\color{red}f} & * \\
& & * & * & & & * & & * & {\color{red}f} \\
* & & & & & & & * & & * \\
& & & & * & {\color{red}f} & * & & * & {\color{red}f} \\
* & * & * & & & * & {\color{red}f} & * & {\color{red}f} & *
\end{pmatrix}$$



Sparse linear algebraic solvers: multifrontal method

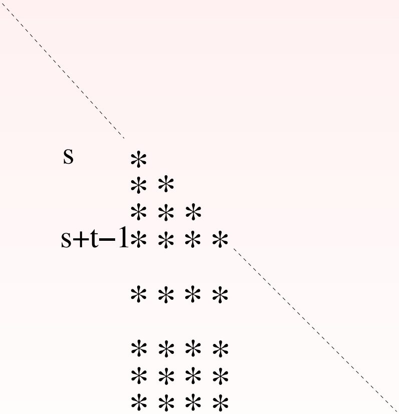
Parallel direct methods: 2b. data-based parallelization

$$\begin{pmatrix}
* & & & & & & & * & & * \\
 & * & & & & & & & & * \\
 & & * & & & & * & & & * \\
 & & & * & & & * & & & \\
 & & & & * & * & & & * & \\
 & & & & * & * & & & f & * \\
 & & * & * & & & * & & * & f \\
* & & & & & & & * & & * \\
 & & & & * & f & * & & * & f \\
* & * & * & & & * & f & * & f & *
\end{pmatrix}$$



Sparse linear algebraic solvers: supernodes

Parallel direct methods: 2b. data-based parallelization



Sparse linear algebraic solvers: supernodes

Parallel direct methods: 2b. data-based parallelization

- Nonsymmetric case: more ways to define supernodes

$$
\begin{array}{c|ccccccccc|}
 & & s & & s' & & & & & \\
\hline
 & \ddots & & & & & & & & \\
s & & * & & & & * & * & * & \\
 & & * & * & & & * & * & * & \\
t & & * & * & * & & * & * & * & \\
 & & * & * & * & \ddots & & & & \\
 & & & & & & \ddots & & & \\
 & & * & * & * & & & \ddots & & \\
 & & * & * & * & & & & \ddots & \\
 & & * & * & * & & & & & \ddots \\
\hline
\end{array}
$$

Sparse linear algebraic solvers: supernodes

Parallel direct methods: 2b. data-based parallelization

- Consider a **matrix-matrix multiplication** as a part of an update operation within the LU factorization
- **Symmetric case:** BLAS3 efficiency
- **Nonsymmetric case:**
  - Second operand can be possibly arranged as a set of dense columns
  - Data access such that they are efficiently addressed.
  - This is called **supernode-panel**: formally a set of matrix-vector operation that profits from multiplying a set of dense columns.

Sparse linear algebraic solvers: supernodes

Parallel direct methods: 2b. data-based parallelization

- Schematically: The three columns of U can be considered as a panel that is a not fully dense block.

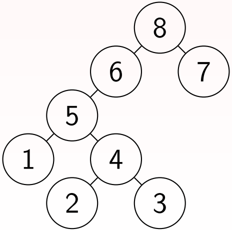
$$\begin{array}{cc}
 & \begin{array}{ccccccccc} & s & & s' & & & & & \end{array} \\
\begin{array}{c} \\ s \\ \\ t \\ \\ \\ \\ \\ \\ \end{array} & \begin{pmatrix}
\ddots & & & & & & & & \\
 & * & & & & & * & & \\
 & * & * & & & * & * & & \\
 & * & * & * & & * & * & * & \\
 & * & * & * & \ddots & & & & \\
 & & & & & \ddots & & & \\
 & * & * & * & & & \ddots & & \\
 & * & * & * & & & & \ddots & \\
 & * & * & * & & & & & \ddots
\end{pmatrix}
\end{array}$$

Sparse linear algebraic solvers: tree parallelism

Parallel direct methods: 2b. data-based parallelization

Factorizations are driven by an elimination tree

$$\begin{array}{c|cccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\hline
1 & * & & & & * & * & & \\
2 & & * & & * & * & & & * \\
3 & & & * & * & & & & * \\
4 & & * & * & * & f & & & f \\
5 & * & * & & f & * & f & & f \\
6 & * & & & & f & * & & f \\
7 & & & & & & & * & * \\
8 & & * & * & f & f & f & * & *
\end{array}$$



Parallel direct methods: 2b. data-based parallelization

Theoretical basis for the SPD case

Theorem

Let the Cholesky factorization of SPD A be LL^T. Let $\mathcal{T}(s)$ and $\mathcal{T}(t)$ be two disjoint subtrees of the elimination tree $\mathcal{T}(A)$. Then for all $i \in \mathcal{T}(s)$ and $j \in \mathcal{T}(t)$ we have $l_{ij} = 0$.

- Theorem implies that submatrices corresponding to **disjoint subtrees** of $\mathcal{T}(A)$ can be processed **in parallel**.

Parallel direct methods: 2b. data-based parallelization

Tree parallelism

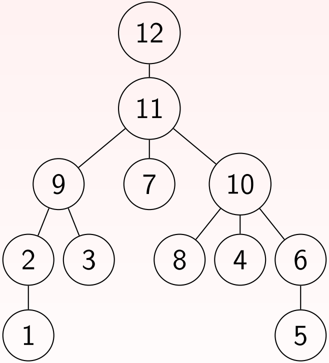


Figure: *An example tree for task scheduling.*

Sparse linear algebraic solvers: substitution steps

Parallel direct methods: 2b. data-based parallelization

Parallel substitution steps and DAG parallelism

- Extending tree parallelism to **LU factorizations** of nonsymmetric A
  - Elimination tree of $A + A^T$.
  - Using directed acyclic graphs of L and U: detecting subtasks to be independently processed.
- Combining node and tree parallelism gives two levels of parallelism.

Sparse linear algebraic solvers: substitution steps

Parallel direct methods: 2b. data-based parallelization

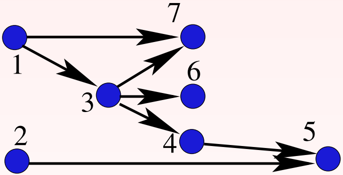
- Substitution steps: not easy to parallelize.
- One possibility: **level scheduling**.
- The idea is to construct a **directed graph model of the transposed factor**.

$$\begin{pmatrix} 1 & & & & & & \\ & 2 & & & & & \\ * & & 3 & & & & \\ & & * & 4 & & & \\ & * & & * & 5 & & \\ & & * & & & 6 & \\ * & & * & & & & 7 \end{pmatrix} \tag{77}$$

Sparse linear algebraic solvers: substitution steps

Parallel direct methods: 2b. data-based parallelization

The directed graph of L^T is



Level scheduling then determines **vertex sets** called **levels** such that the subgraphs induced by the levels **do not contain edges**, that is their induced submatrices are diagonal. The forward substitution of the solve step then considers the **symmetrically reordered** system such that the levels

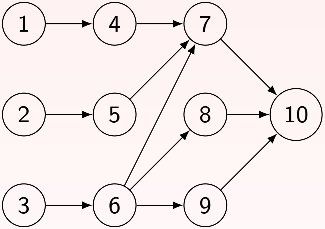
- are contiguously numbered
- and the matrix stays lower triangular.

Sparse linear algebraic solvers: substitution steps

Parallel direct methods: 2b. data-based parallelization

Figure gives an example of the level scheduling approach that finds the structure of sources $\mathcal{K} = \{\{1,2,3\},\{4,5,6\},\{7,8,9\},\{10\}\}$.

$$L = \begin{array}{cc} & \begin{array}{cccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 6 \\ 6 \\ 6 \\ 6 \end{array} & \left(\begin{array}{cccccccccc} * & & & & & & & & & \\ & * & & & & & & & & \\ & & * & & & & & & & \\ * & & & * & & & & & & \\ & * & & & * & & & & & \\ & & * & & & * & & & & \\ & & & * & * & * & * & & & \\ & & & & & * & & * & & \\ & & & & & * & & & * & \\ & & & & & & * & * & * & * \end{array} \right) \end{array}$$



Sparse linear algebraic solvers: substitution steps

Parallel direct methods: 2b. data-based parallelization

Parallel substitution steps and DAG parallelism

Theorem

Assume that a given digraph $G = (V, E)$ is acyclic. Then there exists its vertex v such that $adj^{-}(v) = \emptyset$. Such vertex will be called the **source** *of the graph G.*

- The key to parallelize the substitution steps can be then based on repeated search for sources in a sequence of graphs that start with $\mathcal{G}(L)$ and output a **structure of sources** of $\mathcal{G}(L)$.. Sets of the structure of sources describe components of y that can be computed in parallel.

Sparse linear algebraic solvers: substitution steps

Parallel direct methods: 2b. data-based parallelization

Algorithm (**Find sets of components of x that can be computed in parallel in solving $Lx = b$**)

Input: *Lower triangular matrix matrix A of dimension n. Fully dense right-hand side vector b.*
Output: *Structure of sources $\mathcal{K} = \{K_1, \ldots, |\mathcal{K}|\}$ of indices of components such that solution components in the sets K_i, $i = 1, \ldots, |\mathcal{K}|$ can be computed in parallel.*

```
Set $G = \mathcal{G}(L)$, $i = 0$
while $V(G)$ is not empty do
    $i = i + 1$
    Define $K_i$ as the set of all sources of $G$
    Set $G = \mathcal{G}(V \setminus K_i)$
end while
```

Sparse linear algebraic solvers: other

Parallel direct methods: 2b. data-based parallelization

Twisted factorization

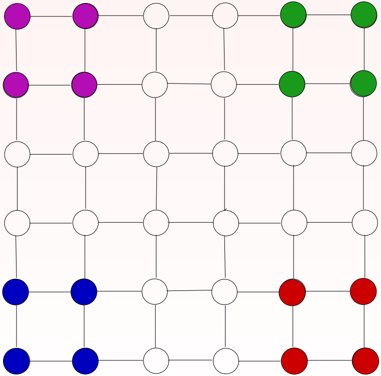
$$\begin{pmatrix}
* & & & & & & & & \\
* & * & & & & & & & \\
 & * & * & & & & & & \\
 & & * & * & & & & & \\
 & & & * & * & * & & & \\
 & & & & & * & * & & \\
 & & & & & & * & * & \\
 & & & & & & & * & * \\
 & & & & & & & & *
\end{pmatrix}
\quad
\begin{pmatrix}
* & * & & & & & & & \\
 & * & * & & & & & & \\
 & & * & * & & & & & \\
 & & & * & * & & & & \\
 & & & & * & & & & \\
 & & & & * & * & & & \\
 & & & & & * & * & & \\
 & & & & & & * & * & \\
 & & & & & & & * & *
\end{pmatrix}$$

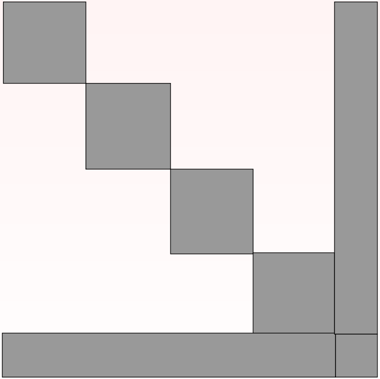
Sparse linear algebraic solvers: other

Parallel direct methods: 2b. data-based parallelization

More domains

- Twisted factorization: 2 domains
- Generalizations to more domains possible





Sparse linear algebraic solvers: other

Parallel direct methods: 2b. data-based parallelization

Parallelizing of factorizations by a posteriori modifications or reorderings

$$\begin{pmatrix} 11 & -3 & 4 & 1 & & & \\ & 1 & 3 & & 5 & & \\ 3 & & 8 & 7 & & & \\ & 6 & & 7 & 5 & 4 & \\ & 17 & 2 & & & & 5 \\ 1 & 2 & & & 3 & & \end{pmatrix} \longrightarrow \begin{pmatrix} 11 & -3 & 4 & 1 \\ 1 & 3 & 5 & \\ 3 & 8 & 7 & \\ 6 & 7 & 5 & 4 \\ 17 & 2 & 5 & \\ 1 & 2 & 3 & \end{pmatrix}$$

Matrix in the resulting format on the right-hand side can be possibly better vectorized. In practice, there are more very similar schemes of this kind. The output format can have different names, like **jagged diagonal format** or **striped format**, can be completed by an additional row permutation etc.

Sparse linear algebraic solvers

Parallel iterative methods: why?

- Consider a 2D matrix problem connected to a $k \times k$ mesh, $n = k^2$. In order to move an information across the mesh at least $O(k)$ steps are needed since in standard iterative methods this information moves only by a **constant number of gridpoints** per step.
- The conjugate gradient (CG) method has one matrix-vector multiplication per iteration.
- This accounts for $O(k^2)$ **time per iteration**.
- Then CG needs at least $O(k^3) = O(n^{3/2})$ time if the information have to be spread over all the gridpoints.
- 3D problem using $k \times k \times k$ grid.
- Again, $O(k)$ steps are needed to spread the information.
- Altogether: at least $O(k^4) = O(n^{4/3})$ time is needed.
- Consequently, **with respect to the dimension** the 3D case has **lower complexity** than 2D.

Parallel iterative methods: why? (continued)

- **Direct and iterative solvers** Direct solvers work for simple grid (Poisson) problem (24): time complexity
  - $O(n^{3/2})$ in 2D,
  - $O(n^2)$ in 3D.

  Memory for a general direct solver (based on the nested dissection model)
  - $O(n \log n)$ in 2D,
  - $O(n^{4/3})$ in 3D.
- Iterative methods: memory is always proportional to n for **grid problems**.
- Computational **model problem complexity** to solve the Poisson problem is $O(n^{3/2})$ in 2D and $O(n^{4/3})$ in 3D.
- But, realistic problems may be **far more difficult** to solve than the model problems.
- Consequently, an **acceleration of iterative methods by preconditioning** is a must.

Iterative methods: summary of operations (I)

- **Sparse matrix-vector multiplication**
- **data decomposition** for sparse matrices are different from those for dense matrices.
- They can be based on the nonzero counts but **separators from partitioning** can be taken into account as well.
- Mapping from **global to local** indices may be based on **hashing schemes** if a **global array cannot be stored at individual computational elements**. Hash tables generalize the concept of a direct addressing from a large array into a direct addressing of a small array that can be stored locally completed by a hash function. Hash functions can be based, for example, on **remainders** after divisions or on modified remainders.

Iterative methods: summary of operations (II)

- **Sparse matrix - dense matrix multiplications**
- In case if we have more right-hand sides, operations among dense submatrices and dense subvectors may use BLAS3.
- **Sparse matrix-matrix multiplications**.
- Data storage schemes that **can exploit the sparsity** should be used. See the text on sparse matrices.
- **Orthogonalization** in some algorithms (GMRES).
- Here we may have a problem of numerical issues versus parallelizability. An example: choice of a variant of the Gram-Schmidt orthogonalization.

$$CGS \quad \times \quad MGS. \tag{78}$$

- **Vector operations**
- Vector operations in iterative methods often based on **dense vectors** and can be often straightforwardly vectorized or parallelized.

Sparse linear algebraic solvers

Iterative methods: summary of operations (III - global reductions)

Standard HS implementation of the conjugate gradient (CG) method:

Algorithm

HS conjugate gradient method

Input: *Symmetric positive definite matrix* $A \in R^{n \times n}$, *right-hand side vector* $b \in R^n$ *of the system* $Ax = b$, *initial approximation* $x_0 \in R^n$ *to* $x \in R^n$.

Output: *Solution approximation* x_n *after the algorithm has been stopped.*

```
0. Initialization: $r_0 = b - Ax_0,\ p_0 = r_0$
1. for $i = 1 : nmax$ do
2.$\alpha_{i-1} = \frac{(r_{i-1}, r_{i-1})}{(p_{i-1}, Ap_{i-1})}$
3.$x_i = x_{i-1} + \alpha_{i-1} p_{i-1}$
4.$r_i = r_{i-1} - \alpha_{i-1} Ap_{i-1}$
5.evaluate the stopping criterion
6.$\beta_i = \frac{(r_i, r_i)}{(r_{i-1}, r_{i-1})}$
7.$p_i = r_i + \beta_i p_{i-1}$
8. end do
```

Iterative methods: summary of operations (III - global reductions)

- HS CG method by Hestenes and Stiefel has **two synchronization points** separated in the loop by vector operations. Parallel computation of the scalars that corresponds to the synchronization points can be based on **reduction** of the **fan-in** type. As above, depth of the fan-in scheme has **logarithmic** complexity.
- Some other variants of the conjugate gradient method may have only **one synchronization** point since the two scalar products **can be** computed at the same time and their computation is not separated by additional vector operations. This can be important from the parallelism point of view.
- This may lead to worse **behavior in finite precision** arithmetic.

Iterative methods: summary of operations (III - global reductions)

Algorithm

ST (three-term) conjugate gradient method *(Stiefel, 1955; Rutishauser, 1959*

Input: *Symmetric positive definite matrix $A \in R^{n \times n}$, right-hand side vector $b \in R^n$ of the system $Ax = b$, initial approximation $x_0 \in R^n$ to $x \in R^n$.*

Output: *Solution approximation x_n after the algorithm has been stopped.*

```
0. Initialization: $r_0 = b - Ax_0$, $p_0 = r_0$, $x_{-1} = x_0$, $r_{-1} = r_0$, $e_{-1} = 0$
1. for $i = 1 : nmax$ do
2.$q_{i-1} = \frac{(r_{i-1}, Ar_{i-1})}{(r_{i-1}, r_{i-1})} - e_{i-2}$
3.$x_i = x_{i-1} + \frac{1}{q_{i-1}}[r_{i-1} + e_{i-2}(x_{i-1} - x_{i-2})] \equiv x_{i-1} + \frac{1}{q_{i-1}}[r_{i-1} + e_{i-2}\Delta x_{i-1}]$
4.$r_i = r_{i-1} + \frac{1}{q_{i-1}}[-Ar_{i-1} + e_{i-2}(r_{i-1} - r_{i-2})] = r_{i-1} + \frac{1}{q_{i-1}}[-Ar_{i-1} + e_{i-2}\Delta r_{i-1}]$
5.evaluate the stopping criterion
6.$e_{i-1} = q_{i-1}\frac{(r_i, r_i)}{(r_{i-1}, r_{i-1})}$
7. end do
```

Sparse linear algebraic solvers

Iterative methods: changing layout of Krylov space methods

- Standard layout of iterative Krylov space methods can be changed in more ways.
  - **Moving synchronization points** mentioned above.
  - **Pipelining** vector operations (can be done in a straightforward way)
  - **Overlapping communication and computation**.
  - Overlapping by splitting the Cholesky factorization preconditioner on the next slide.

Iterative methods: changing layout of Krylov space methods

Algorithm

Preconditioned HS conjugate gradient method

Input: *Symmetric positive definite matrix* $A \in R^{n\times n}$*, right-hand side vector* $b \in R^n$ *of the system* $Ax = b$*, preconditioner* LL^T*, initial approximation* $x_0 \in R^n$ *to* $x \in R^n$*.*

Output: *Solution approximation* x_n *after the algorithm has been stopped.*

```
0. Initialization: $r_0 = b - Ax_0,\ p_{-1} = 0, \beta_{-1} = 0, \alpha_{-1} = 0, s = L^{-1}r_0, \rho_0 = (s,s)$
1. for $i = 0 : nmax$ do
2.     $w_i = L^{-T}s$
3.     $p_i = w_i + \beta_{i-1}p_{i-1},\ q_i = Ap_i$
7.$\gamma = (p_i, q_i),\ x_i = x_{i-1} + \alpha_{i-1}p_{i-1}$
2.$\alpha_i = \frac{\rho_i}{\gamma},\ r_{i+1} = r_i - \alpha_i q_1$
7.$s = L^{-1}r_{i+1}$
7.$\rho_{i+1} = (s,s)$
3.$x_i = x_{i-1} + \alpha_{i-1}p_{i-1},\ r_i = r_{i-1} - \alpha_{i-1}Ap_{i-1}$
5.evaluate stopping criterion; if satisfied $x_{i+1} = x_i + \alpha_i p_i$, stop
6.$\beta_i = \frac{\rho_{i+1}}{\rho_i}$
8. end do
```

Accelerating iterative methods: preconditioning

- A way to accelerate iterative solvers is **preconditioning**.
- Here we mention **algebraic preconditioners** and discuss their parallel aspects. There are two important tasks connected to this.
  - Parallelizing **preconditioner** construction
  - Parallelizing the solves with **preconditioners**.

Outline

1. Foreword
2. Computers, computing, communication
3. Parallel computing
4. Parallel processing and us - parallel programming
5. Parallel computer architectures: hardware and classification
6. Combining pieces together: computational models
   - Uniprocessor model
   - Vector and SIMD model
   - Multiprocessor model
7. Parallelizing problems
8. Sparse data decomposition: graph partitioning
9. Parallel and parallelized algebraic preconditioning

Approximate factorizations, splitting and preconditioning

Definition

Linear onestep stationary iterative method is a process where the relation between the two subsequent iterates $x, x^+ \in R^n$ is expressed as

$$x^+ = Sx + M^{-1}b. \tag{79}$$

$S, M \in R^{n \times n}$; M regular. Matrix S is called the **iteration matrix**.

- Briefly called **stationary iterative methods**.
- **Consistence** of an iterative method is expressed by

$$x^* = Sx^* + M^{-1}Ax^*,$$

- This implies

$$S = I - M^{-1}A,$$

where x^* is a solution of $Ax = b$.

Approximate factorizations, splitting and preconditioning

- Another expression

$$x^{+} = x - M^{-1}Ax + M^{-1}b \equiv (I - M^{-1}A)x + M^{-1}b. \tag{80}$$

- Or

$$M(x - x^{+}) = Ax - b. \tag{81}$$

- Different choices of M imply different iterative methods.
- Choosing M from

$$A = M - R \equiv M - (M - A) \tag{82}$$

  for some $R \in R^{n}$ is called a choice by **splitting** of A.
- The choice $M = I$ is sometimes called **simple iteration**.
- Matrix M can be called a **preconditioning** of the simple iteration.

Approximate factorizations, splitting and preconditioning

Definition

Stationary iterative method for solving

$$Ax = b,\ A \in R^{n\times n},\ x \in R^{n},\ b \in R^{n} \tag{83}$$

is **convergent** if the sequence of its iterates converges to the problem solution x^{*} independently of the choice of the initial approximation x_0.

- Remind that the **spectral radius** of $S \in R^{n\times n}$ is given as

$$\lim_{k\to\infty} \parallel S^{k} \parallel^{1/k}, \tag{84}$$

- Another equivalent expression:

$$\rho(S) = \max\{|\lambda_i| \,|\, \lambda \in \sigma(A)\}, \tag{85}$$

Approximate factorizations, splitting and preconditioning

Theorem

Stationary iterative method (79) with iteration matrix S is **convergent** *iff*

$$\rho(S) < 1,$$

where $\rho(S)$ is the spectral radius of S.

Approximate factorizations, splitting and preconditioning

Preconditioning as a general transformation

- $Ax = b$, M regular
- $$M^{-1}Ax = M^{-1}b. \tag{86}$$
- $$x^{+} + M^{-1}Ax = x + M^{-1}b. \tag{87}$$
- $$x^{+} = (I - M^{-1}A)x + M^{-1}b, \tag{88}$$

Approximate factorizations, splitting and preconditioning

- Construct $M^{-1}A$ or not?

Desirable properties of preconditioning

- small
$$\| M - A \|$$
- small
$$\| I - M^{-1}A \| .$$
Note that these norms may be very different
- Stable application of composed preconditioners as $M = M_1 M_2$
- Useful for the specific target computer architecture.

Approximate factorizations, splitting and preconditioning

Left, right or split preconditioning

$$\begin{aligned} M^{-1}Ax &= M^{-1}b \\ AM^{-1}y &= b, x = My \\ M_1^{-1}AM_2^{-1}y &= M_1^{-1}b, x = M_2y, M = M_1M_2 \end{aligned}$$

Theorem

*Let ϵ and Δ are positive numbers. Then for every $n \geq 2$ there are regular matrices $A \in R^n$ and $X \in R^n$ such that all entries of $XA - I$ have magnitudes less than ϵ and all entries of $AX - I$ have magnitudes larger than Δ [**?**].*

Approximate factorizations, splitting and preconditioning

Let A be SPD. Then the system preconditioned from both sides

$$L_M^{-1} A L_M^{-T} y = L_M^{-1} b, \; x = L_M^T y \tag{89}$$

where $M = L_M L_M^T$ has SPD system matrix $L_M^{-1} A L_M^{-T}$ and can be solved by the CG method.

Theorem

Consider solving $Ax = b$ with SPD preconditioning matrix M. Then

- *$M^{-1}A$ is self-adjoint in the dot product $(.,.)_M = (M.,.)$.*
- *AM^{-1} is self-adjoint in the dot product $(.,.)_{M^{-1}} = (M^{-1}.,.)$.*

Approximate factorizations, splitting and preconditioning

Proof.

$$\begin{aligned}(M^{-1}Ax, y)_M &= (Ax, y)\\ &= (x, Ay)\\ &= (x, MM^{-1}Ay)\\ &= (Mx, M^{-1}Ay)\\ &= (x, M^{-1}Ay)_M\end{aligned}$$

$$(AM^{-1}x, y)_{M^{-1}} = (AM^{-1}x, M^{-1}y) = (M^{-1}x, AM^{-1}y) = (x, AM^{-1}y)_{M^{-1}} \tag{90}$$

□

Approximate factorizations, splitting and preconditioning

Corollary

CG method preconditioned from the left based on the dot product $(.,.)_M$*, CG method preconditioned from the right based on the dot product* $(.,.)_{M^{-1}}$ *and CG method using standard dot product and preconditioned from both sides as above (89) provide in the exact arithmetic the same iterates.*

Accelerating iterative methods: preconditioning

- Many standard preconditioners are based on approximate **LU/Cholesky factorization** and also called **incomplete factorizations**. Problems in parallel computational environment.
- Factorization constructions have often to be reformulated to become more efficient. Although the construction is often rather **cheap**, preconditioners may not have large and dense blocks.
- But they may have more of structural parallelism (analogy of the tree paralellism) being often more sparse.
- Also the challenge of **solve steps** may be more suitable here (often more possible parallel branches due to the sparsity)

Accelerating iterative methods: preconditioning

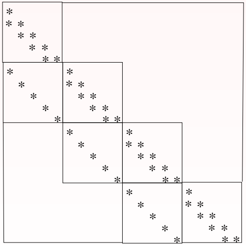
- Main directions in **parallel and parallelized preconditioners**
  - Specific approximation techniques or modifications of factorizations.
  - Specific **reorderings**.
  - Finding and exploiting blocks in the system matrix.
  - A posteriori reorderings to enhance matrix-vector multiplications or the solve steps.
  - Approximating **directly** $M^{-1} \approx A^{-1}$.

Sparse linear algebraic solvers

Accelerating iterative methods: specific preconditioning techniques

1. Partial vectorization

- Strategy that exploits the vectorization potential in case of structured matrices. The fill-in limited to **original diagonals and possibly few others.**
- Useful mainly in special cases, e.g., for matrices from structured (regular) grids.



Sparse linear algebraic solvers

Accelerating iterative methods: specific preconditioning techniques

2. Forced aposteriori annihilation

- Based on dropping such entries of the incomplete factors that prohibit efficient (partial) vectorization and/or parallelization.

$$\begin{pmatrix} * & & & & & & & & & \\ * & * & & & & & & & & \\ & * & * & & & & & & & \\ & & * & * & & & & & & \\ & & & * & * & & & & & \\ & & & & * & * & & & & \\ & & & & & * & * & & & \\ & & & & & & * & * & & \\ & & & & & & & * & * & \\ & & & & & & & & * & * \end{pmatrix} \to \begin{pmatrix} * & & & & & & & & & \\ * & * & & & & & & & & \\ & * & * & & & & & & & \\ & & & * & & & & & & \\ & & & * & * & & & & & \\ & & & & * & * & & & & \\ & & & & & * & * & & & \\ & & & & & & & * & & \\ & & & & & & & * & * & \\ & & & & & & & & * & * \end{pmatrix}$$

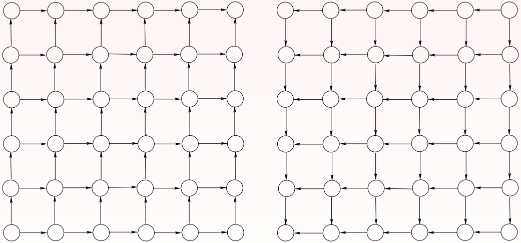
- Finding entries to be dropped may be difficult.
- **Slowdown** of convergence often faced.

Sparse linear algebraic solvers

Accelerating iterative methods: specific preconditioning techniques

3. Wavefront processing.

- Again, originally introduced for factorization of matrices from structured grids.
- Parallel potential of this approach similar of the one of the fine grain implementations of simple stationary iterative methods.



- Can be generalized to other stencils, even to unstructured problems.

Accelerating iterative methods: specific reorderings

Red-black reorderings and their multilevel extensions

Discretized 2D Poisson equation (24). grid points: rows and columns of the matrix. **Natural** matrix ordering gives the depicted matrix: bijection β between the grid points $(i,j) \in \{1,\dots,nx\} \times \{1,\dots,ny\}$ of the $nx \times ny$ grid and row/column indices given by

$$\beta : (i,j) \longleftrightarrow i + nx * (j-1). \tag{91}$$

$$\begin{pmatrix}
4 & -1 & & -1 & & & & & \\
-1 & 4 & -1 & & -1 & & & & \\
 & -1 & 4 & & & -1 & & & \\
-1 & & & 4 & -1 & & -1 & & \\
 & -1 & & -1 & 4 & -1 & & -1 & \\
 & & -1 & & -1 & 4 & & & -1 \\
 & & & -1 & & & 4 & -1 & \\
 & & & & -1 & & -1 & 4 & -1 \\
 & & & & & -1 & & -1 & 4
\end{pmatrix} \tag{92}$$

Accelerating iterative methods: specific reorderings

1. Red-black reordering

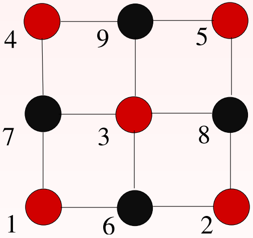
Red-black reordering is the reordering of the grid points /matrix rows and columns allowing the permuted matrix to be written in the block form as

$$A = \begin{array}{c|cc} & \text{red points} & \text{black points} \\ \hline \text{red points} & D_1 & F \\ \text{black points} & E & D_2 \end{array} \tag{93}$$

- D_1 and D_2 are diagonal matrices.
- $E = F^T$ if matrix is symmetric
- Coloring in the figure below.

Accelerating iterative methods: specific reorderings

1. Red-black reordering (continued)



$$\begin{pmatrix}
4 & & & & & -1 & -1 & & \\
& 4 & & & & -1 & -1 & & \\
& & 4 & & & -1 & -1 & -1 & -1 \\
& & & 4 & & & -1 & & -1 \\
& & & & 4 & & & -1 & -1 \\
-1 & -1 & -1 & & & 4 & & & \\
-1 & & -1 & -1 & & & 4 & & \\
& -1 & -1 & & -1 & & & 4 & \\
& & -1 & -1 & -1 & & & & 4
\end{pmatrix}$$

Figure: *Red-black reordering and the system matrix reordered such that the red nodes come first.*

Accelerating iterative methods: specific reorderings

1. Red-black reordering (continued)

Red-black reordering exists if and only if the adjacency graph of the matrix is **bipartite**. Considering the factorization, one its block step based on the pivoting diagonal block D_1 provides

$$A = \begin{pmatrix} D_1 & F \\ E & D_2 \end{pmatrix} = \begin{pmatrix} I & \\ ED_1^{-1} & I \end{pmatrix} \begin{pmatrix} D_1 & \\ & D_2 - ED_1^{-1}F \end{pmatrix} \begin{pmatrix} I & D_1^{-1}F \\ & I \end{pmatrix}. \quad (94)$$

- Partial elimination of the rows and columns that correspond to this step results just to **scaling of the offdiagonal blocks** by the diagonal block D_1
- Since D_2 is diagonal, this simplifies the computation of the Schur complement

$$S = D_2 - ED_1^{-1}F. \quad (95)$$

- In our case of discretized Laplace operator the matrix can be considered as a **block tridiagonal** matrix and this property is transferred into the Schur complement.

Sparse linear algebraic solvers

Accelerating iterative methods: specific reorderings

2. Red-black reordering and stability problems

- Red/black reorderings may enhance **factorization efficiency** but also decrease **factorizability** of the matrix in some incomplete factorizations.
- This is an additional adverse effect in addition to possible deterioration of convergence of the preconditioned iterative method.
- Consider a simple modified incomplete factorization MIC(0) that does not allow fill-in.
- Stencil depicted.

$$\begin{array}{ccccc} \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & -1 & \cdot & \cdot \\ \cdot & -1 & 4 & -1 & \cdot \\ \cdot & \cdot & -1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \end{array} \tag{96}$$

Accelerating iterative methods: specific reorderings

2. Red-black reordering and stability problems

Consider elimination of the **red** nodes (rows/columns) earlier than the black nodes. This is what is done in the red-black reordering. Then,

- First: black matrix entries (of internal vertices) in MIC(0) are modified by other black nodes as follows

$$a_{ii} = a_{ii} - \sum_{j_i=1}^{4} \frac{1}{a_{j_i j_i}} = 4 - 4 \times \frac{1}{4} = 3,\ j_i \text{ corresponds to a grid neighbor of i.} \tag{97}$$

- Each of these **four** black neighbors generates three fill-ins **among their other black neighbours** since these nodes would form in the complete LU a clique. The fill-in is in MIC(0) not removed but **subtracted from the diagonal entry** and we have

$$a_{ii} = a_{ii} - \sum_{j_i=1}^{4} 3 \times \frac{1}{a_{j_i j_i}} = 3 - 3 = 0,\ j_i \text{ corresponds to a grid neighbor of i} \tag{98}$$

Sparse linear algebraic solvers

Accelerating iterative methods: specific reorderings

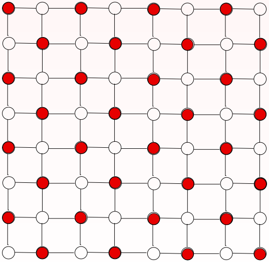
2. Red-black reordering and stability problems

- Clearly, **this** incomplete factorization and MIC(0) can **break down**.
- Consequently, **hunt for more parallelism can be counterproductive**.
- Summarizing: the red-black reordering combined with MIC(0) may significantly damage the **local connections** by
  - unnaturally separating points that were originally topologically close and
  - replacing fill-in by diagonal modification.

Accelerating iterative methods: specific reorderings

3. Recursive red-black reorderings

- Red-black reorderings may imply a significant asymptotic decrease of the condition number of the preconditioned matrix for some model problems
- There are more ways to do recurrent reordering that differ by the choice of the fill-in kept in the subsequent levels.

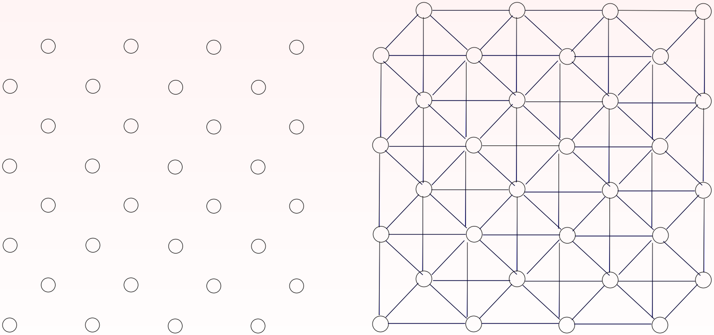


Sparse linear algebraic solvers

Accelerating iterative methods: specific reorderings

3. Recursive red-black reorderings (continued)

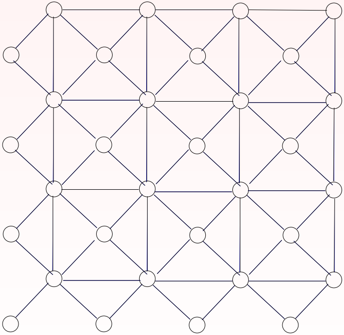
Left side: the black nodes that were originally not connected, **right side:** the "black" Schur complement with the fill-in.



Accelerating iterative methods: specific reorderings

3. Recursive red-black reorderings (continued)

One possibility is to **keep only** the following fill-in for the next level as proposed by Brand (1992), see also Brand, Heinemann (1989).



Accelerating iterative methods: specific reorderings

3. Recursive red-black reorderings (continued)

Possible to show that the ratio of the largest and smallest eigenvalue (of the preconditioned system) for an SPD model problem is after recurrent applications of the MIC preconditioner asymptotically:

$$O(h^{-1/2}). \tag{99}$$

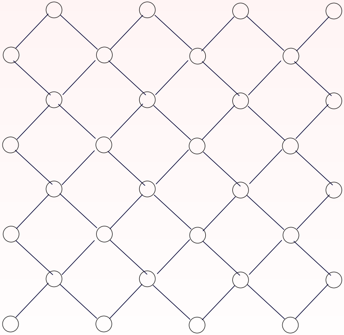
In the other words, this is the asymptotic dependency of the condition number of the symmetrically preconditioned matrix

$$M^{-1/2}AM^{-1/2}. \tag{100}$$

Accelerating iterative methods: specific reorderings

3. Recursive red-black reorderings (continued)

Similar proposals to keep well-chosen fill-in and use the rest to modify the diagonal as a MIC scheme proposed by Ciarlet, Jr. (1992):



Accelerating iterative methods: specific reorderings

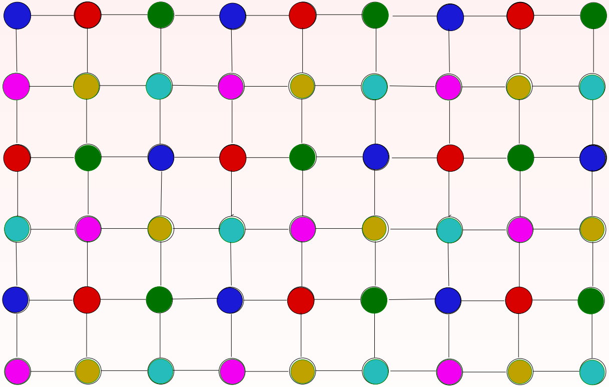
3. Recursive red-black reorderings (continued)

In this case the condition number of the recursively preconditioned system is asymptotically

$$O(h^{-1/3}). \tag{101}$$

- Experiments show that this may be true for more general problems and less structured problems.
- Note that the condition number **does not necessarily imply** a fast convergence of the preconditioned iterative method.
- Also there can be more powerful preconditioners than those from the MIC class.

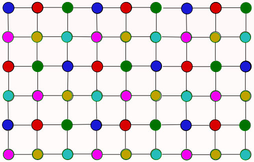
Multicoloring



Accelerating iterative methods: specific reorderings

4. Multicolorings

- Why reorderings based on more colors called **multicolorings** should be used instead of red-black reorderings.
  - The red-black reordering combined with a specific incomplete factorization of the MIC type may lead to stability problems.
  - Multicoloring **naturally generalizes** the red-black concept.
  - Does not necessarily destroy so many global connections in the preconditioner and may not influence convergence so much.



Multicoloring implies block structure with **diagonal diagonal blocks.**
More colors naturally restricts potential parallelism.

Sparse linear algebraic solvers

Accelerating iterative methods: specific reorderings

5. Red-black reordering and multilevel approaches

- If the graph of the matrix structure is not bipartite we can still use two colors and look for a reordering such that **only** the matrix D_1 is diagonal. Such reordering can be obtained considering just the underlying graph.
- A related graph problem: to find an **independent set** in a (typically undirected) graph.
- The multilevel component is in recurrent repetition of the search of independent sets in the subsequent Schur complements that, depending on the type of approximate factorization can have the sparsity structure known or fully **general**.

$$A = \begin{pmatrix} D_1 & F \\ E & C \end{pmatrix} = \begin{pmatrix} D_1^{1/2} & \\ E & I \end{pmatrix} \begin{pmatrix} I & \\ & C - ED_1^{-1}F \end{pmatrix} \begin{pmatrix} D_1^{1/2} & F \\ & I \end{pmatrix}. \qquad (102)$$

Accelerating iterative methods: specific reorderings

5. Red-black reordering and multilevel approaches (continued)

- While there is not much to be saved if the principal leading blocks are chosen as diagonal, if they are block diagonal or even more general, **storing unscaled blocks can decrease memory** at the expense of a slight increase in the algorithm (sequential) efficiency.
- This is what we often do in case of these so-called **multilevel factorizations**.
- Multilevel approaches to compute preconditioners may not only **enhance parallelism** in the construction, but they can be also more efficient due to **less cache faults** despite they may influence convergence.

Sparse linear algebraic solvers

Approximating directly $M^{-1} \approx A^{-1}$

- Substitutions → matvecs
- More possibilities in this class: **factorized, non-factorized**, provided in the form of **polynomial** and so on.
- The inverse of a matrix with a **strongly connected** adjacency matrix is generally **fully dense**.
- But: **Standard incomplete factorizations** with limited nonzero counts represent the matrix very locally since the nonzeros in M correspond to nonzero **edges** of the adjacency graph of A or to the local fill-in. In contrast to this, nonzeros in A^{-1} correspond to **paths** in this graph and this global information may be transferred also to M^{-1}.

Sparse linear algebraic solvers

Approximate inverses

Preconditioning by minimization in the Frobenius norm

Consider $A \in R^{n \times n}$, positive definite and possibly nonsymmetric $W \in R^{n \times n}$ and a constraint in the form of **a prescribed sparsity pattern** $\mathcal{S}$ that has to be satisfied by an approximate matrix inverse $G \in R^{n \times n}$.

$$\text{minimize } F_W(G, A) = \|I - GA\|_W^2 = tr\left[(I - GA)W(I - GA)^T\right]$$

For $W = I$ we get the **least-squares approximate inverse** (AI) that **decouples** to solving n simple least-squares problems.

$$\text{Minimize } F_I(G, A) = \|I - GA\|_F^2 = \sum_{i=1}^{n} \|e_i^T - \tilde{g}_i^T A\|_2^2,$$

where

$$\tilde{g}_i^T, i = 1, \dots n$$

are rows of the approximation G based on the **prescribed sparsity pattern** S.

Sparse linear algebraic solvers

Preconditioning by minimization in the Frobenius norm (continued)

The positive definiteness of W implies that the functional is $F_W(G, A)$ nonnegative and its minima satisfy

$$(GAWA^T)_{ij} = (WA^T)_{ij},\ (i,j) \in \mathcal{S}. \qquad (103)$$

This can be obtained through (since we know that $tr(AB) = \sum_i \sum_j a_{ij} b_{ji}$)

$$\begin{aligned} F_W(G) &= tr\left[(I - GA)W(I - GA)^T\right] \\ &= tr(W) - tr(GAW) - tr(WA^TG^T) + tr(GAWA^TG^T) \\ &= tr(W) - \sum_{i,j} g_{ij}\left[(AW)_{ji} + (WA^T)_{ij}\right] + tr(GAWA^TG^T) \end{aligned}$$

Setting

$$\frac{\partial F_W(G)}{\partial g_{ij}} = 0,\ (i,j) \in \mathcal{S} \qquad (104)$$

we get

$$-(AW)_{ji} - (WA^T)_{ij} + (AWA^TG^T)_{ji} + (GAWA^T)_{ij},\ (i,j) \in \mathcal{S} \qquad (105)$$

Approximate inverses

Preconditioning by minimization in the Frobenius norm (continued)

This implies

$$(GAWA^T)_{ij} = (WA^T)_{ij},\ (i,j) \in \mathcal{S} \tag{106}$$

Its special case where A is also SPD with $W = A^{-1}$ is called the **direct block method** (DB) and it leads to solving

$$\text{Solve } [GA]_{ij} = \delta_{ij},\ (i,j) \in\ \mathcal{S}.$$

Sparse linear algebraic solvers

Approximate inverses (continued)

More sophisticated proposal called SPAI changes the sparsity pattern **dynamically** in outer iterations.

Algorithm

SPAI approximate inverse computation

1. *Choose an initial sparsity pattern and iterate the following steps*
2. *Compute the reduced least squares problem*
3. *Evaluate residual*
4. *Add new rows that correspond to largest residual components*
5. *Add new columns crossed by these rows*
6. *Update the decomposition*

Approximate inverses (continued)

- Later improvements considered, for example, on **more accurate residual evaluations** (Gould, Scott, 1995)
- or high-quality **initial pattern predictions** (Huckle, 1999, 2001; Chow, 2000).
- The approach is **procedurally parallel**, but it may be difficult to distribute A such that all processors have their data even when the prescribed pattern dynamically changes.

Sparse linear algebraic solvers

Approximate inverses (continued)

Another possibility inherently parallel is to use **simple stationary iterative method** to evaluate individual columns c_i by solving systems of the form

$$Ac_i = e_i,\ i = 1, \ldots, n$$

- Simple but typically not very efficient.
- Putting more data dependency to computations **a la Gauss-Seidel**.

Approximate inverses (continued)

- A specific approach: computation of an **approximate inverse Cholesky factor** of an SPD matrix using the Frobenius norm paradigm.
- Minimization is constrained by prescribing the sparsity pattern of the factor.

$$\bar{Z} = \arg\min_{G_L \in \mathcal{S}} F_I({G_L}^T, L) = \arg\min_{G_L \in \mathcal{S}_{\mathcal{L}}} \|I - {G_L}^T L\|_F^2, \text{ where } A = LL^T.$$

Applying (106) to this minimization problem ($A \to L$) with $W = I$, we have

$$(G_L L L^T)_{ij} = (L^T)_{ij}, \ (i,j) \in \mathcal{S}_{\mathcal{L}} \tag{107}$$

that is when plugging in the matrix A

$$(G_L A)_{ij} = (L^T)_{ij}, \ (i,j) \in \mathcal{S}_{\mathcal{L}}. \tag{108}$$

Approximate inverses (continued)

We can see that the pattern $\mathcal{S}_{\mathcal{L}}$ is **lower triangular** and L^T is upper triangular. If we know diagonal entries of L, we get G_L from

$$(G_L A)_{ij} = \begin{cases} (L^T)_{ij} & i = j, \\ 0 & i \neq j. \end{cases} \tag{109}$$

Otherwise, we can find (generally different) factor $\hat{G}_L$ from

$$(\hat{G}_L A)_{ij} = \delta_{ij}, (i,j) \in \mathcal{S}_{\mathcal{L}} \tag{110}$$

and set $G_L = D\hat{G}_L$ such that

$$(D\hat{G}_L A \hat{G}_L^T D)_{ii} = 1, \quad , i = 1, \ldots, n. \tag{111}$$

The procedure can be extended to the nonsymmetric matrix A.

Preconditioning by direct factorization of the inverse

- Standard incomplete LU factorization of A:

$$A \approx LU \tag{112}$$

  and then these triangular factors are **inverted**, either exactly, or incompletely. If the incomplete factors L and U are sparse then even their **exact inverses can be sparse**. Then we can set

$$M^{-1} = U^{-1}L^{-1}. \tag{113}$$

  This strategy leads to a reasonably efficient preconditioner in some cases.

- Another possibility: construct the inverse directly A is strongly regular has the unique factors L, D a U such that L is unit lower triangular, U is unit upper triangular and D is diagonal. We have

$$A^{-1} = WD^{-1}Z^T \quad \Rightarrow \quad A = Z^{-T}DW^{-1} \tag{114}$$

Approximate inverses (continued)

- This can be rewritten into

$$Z^T AW = D. \tag{115}$$

For simplicity, consider a case when A is symmetric and positive definite. Then the columns Z and $W \equiv Z^T$ are mutually orthogonal in the A-scalar product

$$\langle . , . \rangle_A. \tag{116}$$

This represents to evaluate the factors.

- The algorithm to get these factor is **Gram-Schmidt orthogonalization** in this A-scalar product. This procedure is sometimes called A-orthogonalization.
- Its straightforward generalization to the nonsymmetric case (with less theoretical guarantees) is called **biconjugation**.

Approximate inverses (continued)

Algorithm

Inverse factorization via A-orthogonalization

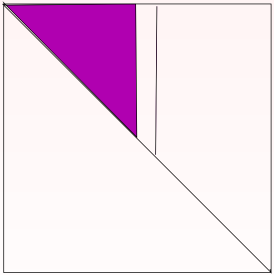
Input: *Sparse SPD matrix $A \in R^{n \times n}$.*

Output: *Unit upper triangular matrix Z such that $A^{-1} = ZD^{-1}Z^T$.*

```
1. for $i = 1 : n$ do
2.       $z_i = e_i - \sum_{k=1}^{i-1} z_k \frac{e_i^T A z_k}{z_k^T A z_k}$
3. end $i$
4. Set $Z = [z_1, \dots, z_n]$
```

Approximate inverses (continued)

The order of operations of Algorithm 9.2 is depicted in the following figure. In each step i of the algorithm for $i = 1, \ldots, n$ a column of Z as well as one diagonal entry of D are computed. This way to compute the factors we call **backward** left-looking) algorithm.



Z

Approximate inverses (continued)

The following result is easy to see

Lemma

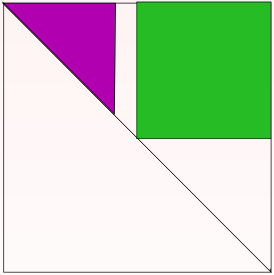
Columns of the factor Z from Algorithm 9.2 satisfy

$$D_{ii} = e_k^T A z_k \equiv z_k^T A z_k, 1 \le k \le n. \tag{117}$$

- Diagonal entries used to divide in Algorithm 9.2 can be computed by at least two ways. The one used here is called the **stabilized** computation of the diagonal entries. The reason for this is that we always have $z_k^T A z_k > 0$ even if the columns of Z are modified, for example, by dropping of offdiagonal entries in an incomplete factorization since A is positive definite.
- This does not need to be true for $e_k^T A z_k$ and the A-orthogonalization can break down in this case.
- This cannot happen for some special matrices, like M-matrices or H-matrices.

Approximate inverses (continued)

Another scheme, the **forward** (right-looking) variant of the algorithm, is given below.



Z

Approximate inverses (continued)

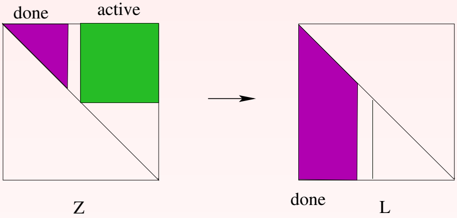
Side-effect of AINV: RIF

AINV can be a way to get the LDL^T factorization as well. The construction steps:

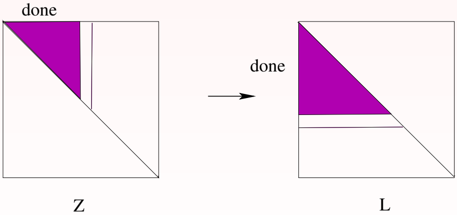
- Find the decomposition $Z^TAZ = D$, where Z is unit upper triangular and D is diagonal.
- The factor L of the decomposition $A = LDL^T$ is $L = AZD^{-1}$, and **it can be easily retrieved** from this inverse factorization.
- The procedure to compute factors in this way is sometimes called robust incomplete factorization (RIF).

The two following figures show data flow of the construction but we do not go into details.

Approximate inverses (continued)



Right-looking approach



Left-looking approach

Sparse linear algebraic solvers

Approximate inverses (continued)

- Approximate inverses or the RIF factors may enhance parallelism
- They can also help to solve systems of linear equations difficult to solve
- They can be used as auxiliary procedures to approximate blocks in block factorizations, see, e.g., Axelsson, Brinkkemper, Il'in, 1984; Concus, Golub, Meurant, 1985.

Sparse linear algebraic solvers

Other approaches to get an approximation to A^{-1}

Bordering scheme is based on the equivalence

$$\begin{pmatrix} Z^T & \\ -y^T & 1 \end{pmatrix} \begin{pmatrix} A & v \\ v^T & \alpha \end{pmatrix} \begin{pmatrix} Z & -y \\ & 1 \end{pmatrix} = \begin{pmatrix} D & \\ & \delta \end{pmatrix}$$

- Other techniques like getting inverse factors from the computed direct factors Alvarado, Dag, 1992
- Based on exploiting the Sherman-Morrison updating formula.

Global matrix iterations

The inverse matrix can be approximated iteratively by the so-called Schulz iterations (Schulz, 1933) based on the scheme

$$G_{i+1} = G_i(2I - AG_i),\, i = 1, \dots$$

The approach is derived from the Newton-Raphson iteration to get p where a given function f is zero. Consider the tangent equation for the function f.

$$y = f'(p_n)x + b. \tag{118}$$

The tangent passes through the point $(p_n, f(p_n))$ and this can be written

$$f(p_n) = f'(p_n)p_n + b \tag{119}$$

that gives

$$b = f(p_n) - f'(p_n)p_n. \tag{120}$$

Global matrix iterations (continued)

Searching for the **zero point in** p_{n+1} we have

$$0 = f'(p_{n+1})p_{n+1} + f(p_n) - f'(p_n)p_n \tag{121}$$

giving

$$p_{n+1} = p_n - \frac{f(p_n)}{f'(p_n)}. \tag{122}$$

Considering the function of the inverse

$$f(x) = 1/x - a$$

we get

$$p_{n+1} = p_n - \frac{1/p_n - a}{-1/p_n^2} = ap_n^2 = 2p_n - ap_n^2. \tag{123}$$

and this implies the matrix generalization of the Newton-Raphson iterations.

Sparse linear algebraic solvers

Polynomial preconditioning

Consider preconditioning of the systems of equations in the following transformation form

$$M^{-1}Ax = M^{-1}b. \tag{124}$$

One possibility to choose preconditioning is to consider its inversion in the form of polynomial $s(A)$ in A of degree k. A natural motivation is that the inverse matrix can be expressed using the characteristic polynomial $p(\lambda) = det(A - \lambda I)$ of A. Cayley-Hamilton theorem implies

$$p_n(A) \equiv \sum_{j=0}^{n} \beta_j A^j = 0. \tag{125}$$

For a regular A we have $\beta_0 = (-1)^n \det(A) \neq 0$ and we get after multiplying by A^{-1}

$$A^{-1} = -\frac{1}{\beta_0} \sum_{j=1}^{n} \beta_j A^{j-1}. \tag{126}$$

Sparse linear algebraic solvers

Polynomial preconditioning

Preconditioning: using the truncated characteristic polynomial

$$M^{-1} = s(A) = \sum_{j=0}^{k} c_j A^k. \tag{127}$$

- The idea of polynomial preconditioning can be found for the first time by Cesari in 1937 who used it to precondition the Richardson iterative method.
- Further development has been pushed by vector and parallel computations, see Stiefel, 1958, since the solve steps are **rich in matvecs operations** that profit parallel processing.
- Furthermore, A and $s(A)$ mutually commute and the evaluation can be based on the Horner scheme, see its more parallel variants by Estrin.

Polynomial preconditioning and the conjugate gradient method

The conjugate gradient method with SPD system matrix searches in its step $k+1$ an approximation x_{k+1} of the solution vector in the form

$$x_{k+1} = x_0 + \mathcal{P}_k(A)r_0,\, k = 0, \ldots. \tag{128}$$

Here the polynomial $\mathcal{P}_k(A)$ minimizes the A-norm of the solution error

$$\|x_{k+1} - x^*\|_A = \sqrt{(x_{k+1} - x^*)^T A (x_{k+1} - x^*)} \tag{129}$$

among all polynomials of the degree k at most k for which $\mathcal{P}_k(0) = 1$. A reason why **another polynomial preconditioning** may be useful may be

- **Iteration count** can be even smaller although the total arithmetic complexity may be larger.
- Polynomial preconditioning can **reduce the number of dot products** that may be problematic on parallel computing architectures.
- **Much less memory**, very **straightforward** and may enable **matrix-free** implementation.

Similarly for polynomial preconditioning of other Krylov methods.

Preconditioning by Neumann series

Neumann series of a matrix $G \in R^{n\times n}$ is called the series

$$\sum_{j=0}^{+\infty} G^j. \tag{130}$$

We have the following theorem

Theorem

Neumann series of $G \in R^{n\times n}$ converges if and only if we have

$$\rho(G) \equiv \{|\lambda_1|, \ldots, |\lambda_n|\} < 1.$$

In this case we have

$$(I - G)^{-1} = \sum_{j=0}^{+\infty} G^j. \tag{131}$$

Let us note that $\rho(G) < 1$ is true if some multiplicative norm $|||G|||$ of G is less than 1.

Preconditioning by Neumann series (continued)

For simplicity we will distinguish two cases.

- Consider splitting A with regular matrix M_1.

  $$A = M_1 - R. \tag{132}$$

  Then

  $$A = M_1(I - M_1^{-1}R) = M_1(I - G). \tag{133}$$

  If

  $$\rho(G) = \rho(M_1^{-1}R) = \rho(I - M_1^{-1}A) < 1$$

  then

  $$A^{-1} = (I - G)^{-1}M_1^{-1} = \left(\sum_{j=0}^{+\infty} G^j\right) M_1^{-1} \tag{134}$$

  Preconditioning that approximates A^{-1} we get by considering only a **finite number** $k+1$ of terms in this expression for the inverse of A. In the other words, the inverse preconditioner M^{-1} is expressed by a **truncated** Neumann series.

Preconditioning by Neumann series (continued)

$$M^{-1} = (I - M_1^{-1}R)^{-1}M_1^{-1} = \left(\sum_{j=0}^{k}(M_1^{-1}R)^j\right)M_1^{-1}. \tag{135}$$

- In order to satisfy the **convergence condition**, scaling should be used. Consider the following splitting for ωA with a real nonzero parameter ω as follows

  $$\omega A = M_1 - R_1 = M_1 - (M_1 - \omega A) = M_1(I - M_1^{-1}(M_1 - \omega A)). \tag{136}$$

  Then

  $$(\omega A)^{-1} = (M_1(I - (I - \omega M_1^{-1}A)))^{-1} = (I - (I - \omega M_1^{-1}A))^{-1}M_1^{-1}. \tag{137}$$

  Parameter ω and regular matrix M_1 can be always chosen such that the matrix $G = (I - \omega M_1^{-1}A)$ has convergence radius less than one.

Preconditioning by Neumann series (continued)

- Matrix $(I - G)^{-1}$ can be approximated by truncated Neumann series

$$\left(\sum_{j=0}^{k} G^j\right). \tag{138}$$

Then we have

$$A^{-1} \approx \omega \left(\sum_{j=0}^{k} G^j\right) M_1^{-1}. \tag{139}$$

- Consequently,

$$M^{-1}A = \left(\sum_{j=0}^{k} G^j\right) M_1^{-1}A = \left(\sum_{j=0}^{k} G^j\right)(I - G) = (I - G^{k+1}). \tag{140}$$

- Another possibility

$$I + \gamma_1 G + \gamma_2 G^2 + \ldots \gamma_k G^k, \tag{141}$$

such that the additional degrees of freedom can be used to optimize it

Preconditioning based on Čebyšev polynomials

This type of polynomial preconditioning has been derived considering the following optimization task in the spectral norm.

$$||I - s(A)A|| = \max_{\lambda_i \in \sigma(A)} |1 - \lambda_i s(\lambda_i)|. \tag{142}$$

This implies a practical goal to find the polynomial s of a **given degree** k minimizing the expression

$$\max_{\lambda \in \sigma(A)} |1 - \lambda s(\lambda)| \tag{143}$$

among all polynomials of the given degree. This task may be relaxed looking for the polynomial that minimizes the expression on some set E that includes the matrix spectrum

$$\max_{\lambda \in E} |1 - \lambda s(\lambda)|, \text{ E includes spectrum} \tag{144}$$

among all polynomials of the given degree. If A is symmetric and positive definite, the set is an interval of R^{+}.

Preconditioning based on Čebyšev polynomials (continued)

Denoting this interval as

$$[a, b], \tag{145}$$

then the problem reduces to search of the polynomial s satisfying

$$s = \min_{p, deg(p) \le k} \max_{\lambda \in [a, b]} |1 - \lambda p(\lambda)|. \tag{146}$$

It is well-known that the solution can be expressed using **scaled and shifted Čebyšev polynomials** of the first kind

$$T_0(\lambda), T_1(\lambda), \ldots. \tag{147}$$

Preconditioning based on Čebyšev polynomials

In case of A SPD we can construct these Čebyšev polynomials in the following way, where δ and θ are $(a+b)/2$ and $(b-1)/2$, respectively.

$$
\begin{aligned}
\sigma_0 &= 1,\ \sigma_1 = \theta/\delta,\ \sigma_{k+1} = 2\theta/\delta\sigma_k - \sigma_{k-1} \\
T_0(\lambda) &= 1/\theta,\ T_1(\lambda) = (4\theta - 2\lambda)/(2\theta^2 - \delta^2) \\
T_k(\lambda) &= \frac{2\sigma_k}{\delta\sigma_{k+1}} + \frac{2\sigma_k(\theta-\lambda)}{\sigma_{k+1}\delta} T_{k-1}(\lambda) - \frac{\sigma_{k-1}}{\sigma_{k+1}} T_{k-2}(\lambda)
\end{aligned}
$$

Preconditioning based on Čebyšev polynomials

If we choose $\lambda_1 = a, \lambda_n = b$ then one can show (Johnson, Miccheli, Paul) that the preconditioned matrix

$$s(A)A \tag{148}$$

has minimum condition number among all such matrices where $s(A)$ has degree k at most.

Čebyšev polynomial preconditioning can be easily applied in the framework of the conjugate gradient method applying the polynomial to residuals r_i using the relation

$$r_i = T_i(A)r_0,\, i = 1, \ldots, n. \tag{149}$$

Even if A is symmetric and indefinite one can propose a polynomial preconditioning. Consider matrix spectrum inside the two intervals

$$[a,\, b] \cup [c,\, d],\, -\infty < a \le b < 0 < c \le d < +\infty, \tag{150}$$

of the same length. That is, we have

$$b - a = d - c. \tag{151}$$

Preconditioning based on least squares polynomials

The quality of the Čebyšev polynomials for SPD matrices strongly depends on the chosen interval/intervals. Straightforward use of the Geršgorin theorem does not need to be enough. There are some proposals to improve convergence. But there also other ways to construct polynomial preconditioners that can provide sometimes better results. One of these proposals is based on the least-squares polynomials and has been proposed by Johnson, Micchelli and Paul in 1983.

Consider the following scalar product of two functions p and q on the real axis

$$\langle p, q \rangle = \int_a^b p(\lambda) q(\lambda) w(\lambda) d\lambda, \tag{154}$$

where $w(\lambda)$ is nonnegative **weight function on** (a, b). The corresponding norm

$$||p||_w = \int_a^b |p(\lambda)|^2 w(\lambda) d\lambda, \tag{155}$$

we will call w-norm.

Preconditioning based on least squares polynomials (continued)

We will look for the preconditioner $s(A)$ in the form of a polynomial on an interval of the real axis that contains matrix eigenvalues. In particular, the polynomial will be a solution of the problem

$$\min_{s \in P_{k-1}} \|1 - \lambda s(\lambda)\|_w. \tag{156}$$

Assume A SPD. If we choose, for example, the weight function $w \equiv 1$ (Legendre weight function) or

$$w(\lambda) = (b - \lambda)^{\alpha} (\lambda - a)^{\alpha}, \alpha > 0, \beta \geq -\frac{1}{2}, \tag{157}$$

(Jacobi weight function), polynomial $s(A)$ can be explicitly computed. If in addition

$$\alpha - 1 \geq \beta \geq -\frac{1}{2}, \tag{158}$$

then even $s(A)A$ has all eigenvalues real and greater than zero.

Preconditioning based on least squares polynomials (continued)

Here we have the following least-squares polynomials $s_k(\lambda)$ of the degree k at most for $k = 1,\, 2,\, 3,\, \alpha = 1/2.\, \beta = -1/2.$

$$\begin{aligned} s_0(\lambda) &= \frac{4}{3} \\ s_1(\lambda) &= 4 - \frac{16}{5}\lambda \\ s_2(\lambda) &= \frac{2}{7}(28 - 56\lambda + 32\lambda^2)). \end{aligned}$$

Derivation of the polynomials can be based, for example, on the relation for kernel polynomials applied to the residual polynomial

$$R_k(\lambda) = 1 - \lambda s_k(\lambda) \tag{159}$$

or using the three-term polynomial recurrence that could be used for some weight functions. Both ways can be found in Stiefel, 1958. Another way is the explicit solution of the normal equations

$$\langle 1 - \lambda s_k(\lambda), \lambda Q_j(\lambda) \rangle_w,\ j = 0, \ldots, k-1, \tag{160}$$

Sparse linear algebraic solvers

Element-by-element preconditioning

Element is traditionaly called a submatrix A_e determined by its row and column indices that contributes to the system matrix in the following way

$$A = \sum_{e=1}^{n_e} A_e. \tag{162}$$

The operation **extend-add** is used to sum the contributions.
One of the possibilities to precondition a systems with this matrix is to propose the preconditioning in the form of elements as well. This could enable an efficient parallel implementation. The simplest proposal is to choose the preconditioning as a **sum of diagonal matrices.**

$$M_e = \sum_{e=1}^{n_e} diag(A_e). \tag{163}$$

Element-by-element preconditioning (continued)

A somewhat more sophisticated approach proposed by Hughes, Levit, Winget, 1983 and formulated for a matrix that is symmetrically scaled by its diagonal (Jacobi scaling) defines the preconditioner as follows

$$M = \sqrt{W}\Pi_{e=1}^{n_e} L_e \Pi_{e=1}^{n_e} D_e \Pi_{e=n_e}^{1} L_e^T \sqrt{W}, \tag{164}$$

where

$$W = diag(A),\ I + \sqrt{W}^{-1}(A_e - diag(A_e)\sqrt{W}^{-1} = L_e D_e L_e^T,\ e = 1, \ldots, n_e \tag{165}$$

Another way has been proposed by Gustafsson, Linskog, 1986. They set

$$M = \sum_{e=1}^{n_e}(\hat{L}_e + D_e)\left(\sum_{e=1}^{n_e} D_e)\right)^{-1}\sum_{e=1}^{n_e}(\hat{L}_e^T + D_e) \tag{166}$$

for

$$A_e = (L_e + D_e)D_e^{+}(L_e^T + D_e),\ e = 1, \ldots, n_e, \tag{167}$$

where D_e^+ is a pseudoinverse of the matrix D_e.