UNIVERSITÀ
DEGLI STUDI
DI PADOVA

DICEA

Department of Civil, Environmental and Architectural Engineering

Introduction to Parallel Computing

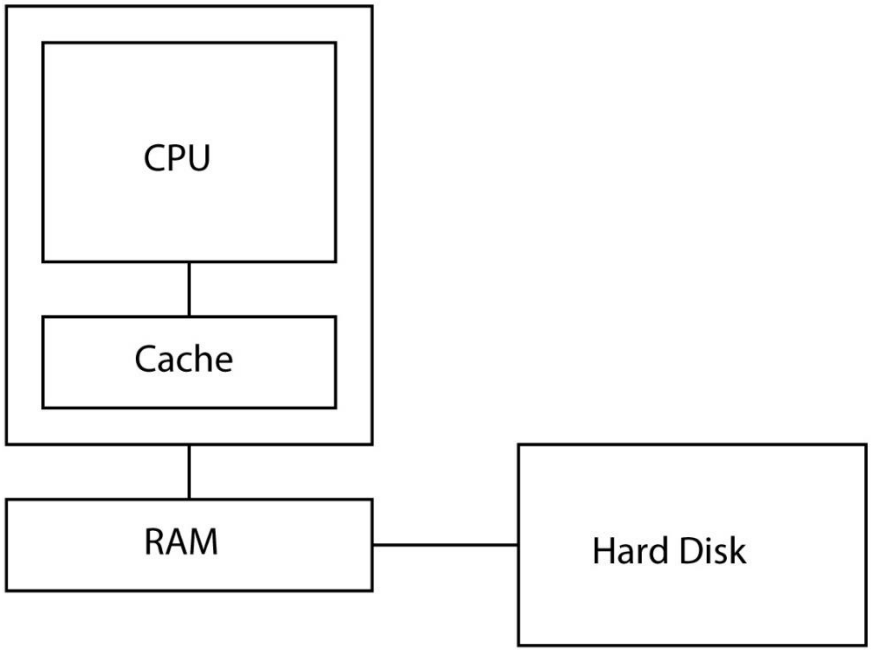
Massimiliano Ferronato

Introduction

- Growing availability of *parallel computers* yields an increase of:
  - Problem size
  - Computation speed
- Numerical algorithms are strictly related to the computational architecture
- *Parallel numerical analysis* is a novel research field aiming at the optimization of the computational performances on parallel computers:
  - Gaining as much parallelism as possible from existing implementations
  - Designing new algorithms specifically developed for parallel computations

Parallel architectures

- Traditional structure of a single-processor computer (*von Neumann machine*)



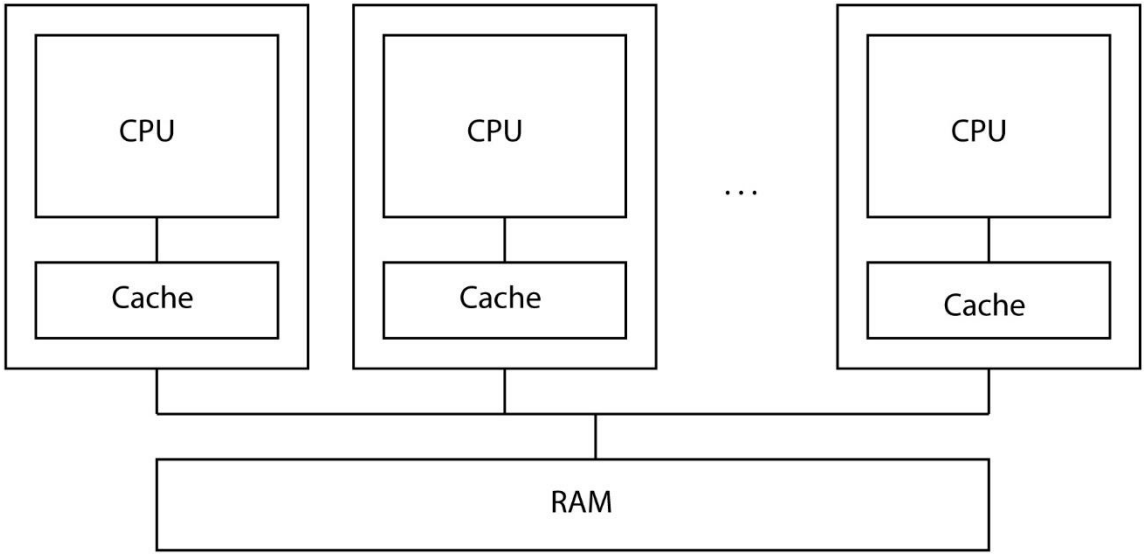
- Algorithms are built thinking that the operations are performed *sequentially*
- A *parallel computer* is equipped with more than one processor and is able to execute several operations *simultaneously*
- There are different parallel architectures according to how the cuncurrent operations are performed and the data are stored

Parallel architectures

- Parameters defining a parallel architecture:
  - Type and number of processors
  - Level of control on the cuncurrent operations: SIMD (Single Instruction-Multiple Data) with a *host* processor and a number of *slave* processors; MIMD (Multiple Instruction-Multiple Data) where each processor is simultaneously host and slave
  - Synchronization: barriers and alignment of processors, or execution of *asynchronous* algorithms
  - Connection among the processors
- The extreme models for a parallel computer are the *Shared Memory* and the *Distributed Memory* architectures

Parallel architectures

Shared Memory Computers

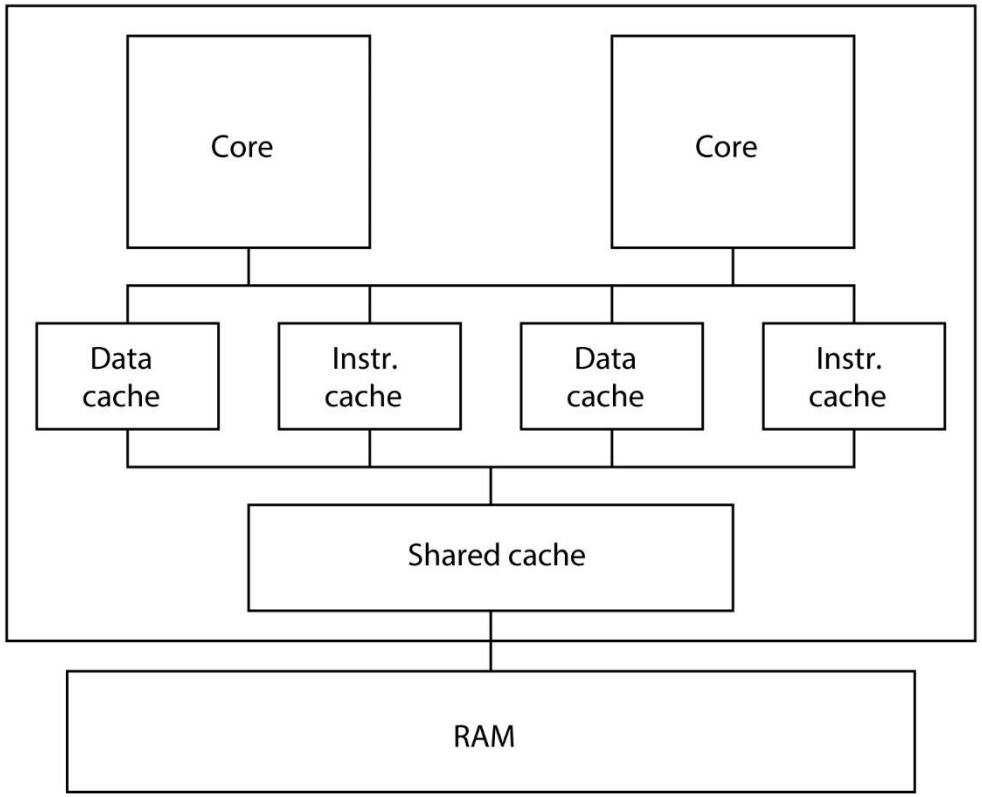


- Use of *OpenMP directives* to manage the access to global memory, the definition of private variables and the different operations performed by the processors
- Easy to code and to transform a sequential program into a parallel one

- Potential problems:
  - Memory conflicts: e.g., two processors access simultaneously the same variable
  - Data consistency: e.g., a processor requires a variables that is being modified by another processor
- Solutions:
  - Variable duplication and definition of private data
  - Use of barriers that enforce the alignment of all processors

Parallel architectures

Shared Memory Computers

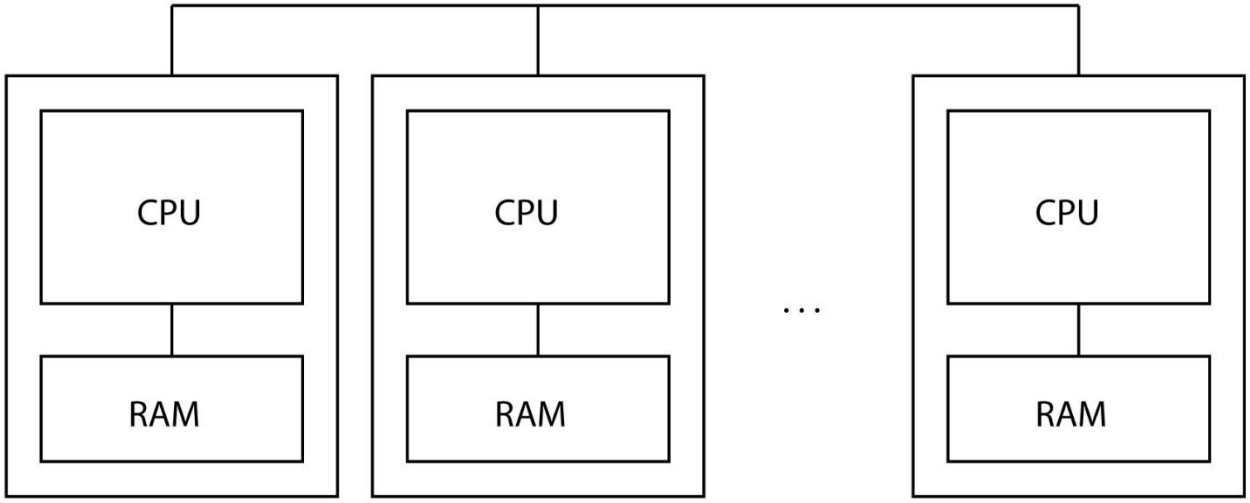


- *Multi-core processors* are a particular kind of Shared Memory Computers
- As some resources are shared among the cores, e.g., the buses or part of the cache, a loss of performance could be expected

Parallel architectures

Distributed Memory Computers

- While the number of processors that can be linked in a shared memory architecture and the size of the memory are physically limited, with distributed memory architectures they are not



- Use of the *Message Passing Interface* (MPI) paradigm

Parallel architectures

Distributed Memory Computers

- Most modern supercomputers are hybrid machines, where each CPU (*node*) is actually a *multi-core processor* that locally can be programmed as a Shared Memory computer
- Writing a code using MPI typically implies a complete revision of the overall algorithm
- The main bottleneck of any simulation on a Distributed Memory architecture is the amount of *interprocessor communications*
- Optimal parallel algorithms for Distributed Memory architectures minimize the number of communications

Computational performance

- The ultimate objective of using a parallel computer is to accelerate the execution of any code
- The quality of the parallelization can be measured in different way and can provide surprising results

- Speed-up: measure of the *computational gain* using *p* processors

$$S_p = \frac{T_1}{T_p}$$

T_p = wall-clock time elapsed using *p* processors

Computational performance

- Efficiency: measure of the fraction of wall-clock time in which a processor is really working, i.e., it is not *idle*

$$E_p = \frac{S_p}{p} \quad \Rightarrow \quad E_p = \frac{T_1}{pT_p}$$

E_p = 1 means *ideal speed-up*

- Total cost: measure of the total quantity of operations performed by *p* processors

$$C_p = pT_p$$

C_p = constant for any *p* means *ideal speed-up*

Computational performance

- Effectiveness: measure of the overall quality of a parallel algorithm

$$F_p = \frac{S_p}{C_p} \quad \Rightarrow \quad F_p = \frac{S_p}{pT_p} = \frac{E_p}{T_p} = \frac{E_p S_p}{T_1}$$

F_p = *max* means that the speed-up is as large as possible at a small total cost

- According to the selected measure and the ultimate objective of the parallelization, an algorithm can be evaluated in different ways

Computational performance

An example: the reduction operation

- Numerical example: the reduction operation. Compute in parallel with *p* processors

$$s = \sum_{i=1}^{16} a_i$$

p	*Tp*	*Cp*	*Sp*	*Ep*	*Fp*
1	15	15	1.00	1.00	0.07
2	8	16	1.88	0.94	0.12
4	5	20	3.00	0.75	0.15
8	4	32	3.75	0.47	0.12

Computational performance

Time complexity

- An important parameter is the size *n* of the problem
  - Strong scalability: *n* is constant and *p* varies
  - Weak scalability: *n* varies in the same ratio as *p*
- Time complexity: the *best performance* that can be obtained for a problem with fixed size *n* and an arbitrary number of processors

$$T_\infty = \min_{p \geq 1} T_p$$

There exists an optimal number *p** of processors such that the wall-clock time does not decrease for any $p > p^*$

Computational performance

Time complexity

- For the reduction operation we have:

$$T_\infty = \log_2 n \qquad \text{and} \qquad p^* = n/2$$

- As $T_1 = n-1$, the efficiency for p^* processors reads:

$$E_{p^*} = \frac{2(n-1)}{n \log_2 n}$$

- The efficiency tends to 0 as n grows to infinity, so the reduction of n scalars is efficient only if using a number of processors much smaller than $n/2$

Computational performance

Time complexity

- Communication penalty: ratio between the real wall-clock time and the ideal time elapsed if there were no communications
- Large values for the communication penalty mean that the number of processors is close to p^* and the parallelization is no longer efficient
- *Amdahl law:* if the sequential part of a code is a fraction f of the total number of operations, then an upper bound for the speed-up exists such that:

$$S_p \leq \frac{1}{f + (1-f)/p} < \frac{1}{f}$$

for any p larger or equal than 1

Parallel algorithms

- The theoretical complexity of a parallel algorithm and the expected performance can be analyzed using the *graph theory*
- A *graph*:

$$G = (N, A)$$

  is a non-empty set made of *nodes N* and *arches A* that link a pair of nodes

- If each pair of nodes linked by an arch has an *order*, than the graph is said to be *direct*
- Two nodes linked by an arch are *adjacent* or *incident*
- The *degree* of a node is the number of arches arriving to or departing from it

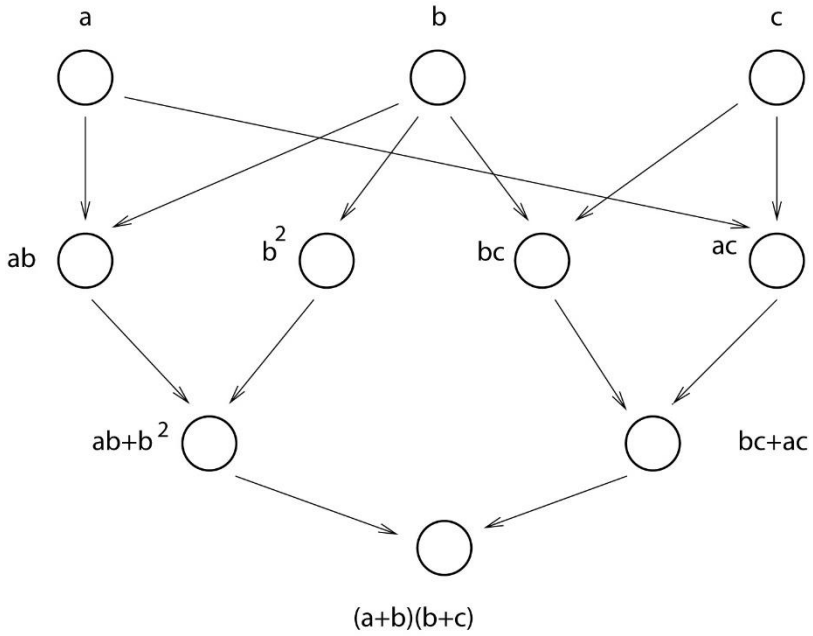
Parallel algorithms

- If a finite number of nodes n_1, n_2, …, n_k can be reached following the arches connecting two nodes we say that a *path* links n_1 to n_k
- If $n_1 = n_k$, with $k > 2$, the path is a *cycle*
- A graph is said to be *connected* if for any node *i* there exists a path that brings to any other node *j*

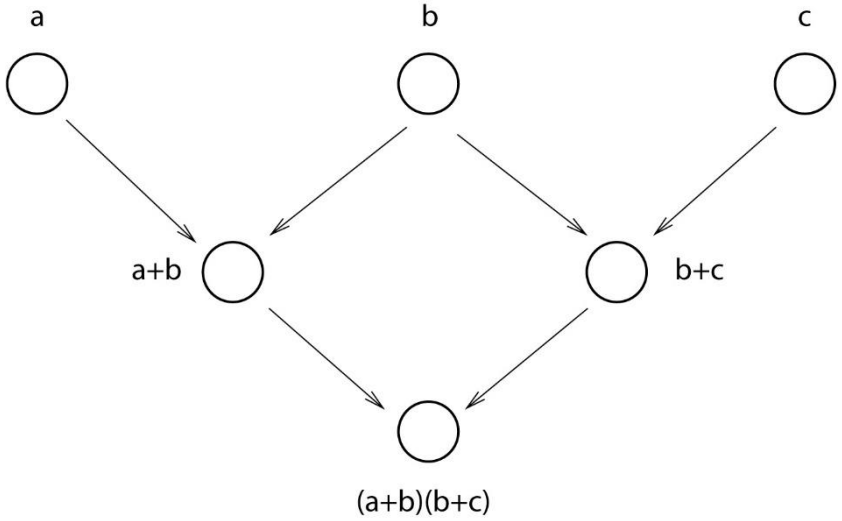
- A parallel algorithm can be theoretically represented by a *Direct A-cyclic Graph* (DAG)
- Every node is an *operation* and every arch denotes the *data dependencies*

Parallel algorithms

- Numerical example: compute $(a+b)(b+c)=ab+b^2+ac+bc$



Algorithm 1



Algorithm 2

- Numerical linear algebra is one of the fields of scientific computing where the use of parallel computers is particularly attractive
- For example, let's consider an iteration of the *Preconditioned Conjugate Gradient* method for solving a Symmetric Positive Definite linear system:

FOR k=0,… until convergence

$\mathbf{t} = A\mathbf{p}_k$

$\alpha_k = \mathbf{r}_k^T\mathbf{p}_k / \mathbf{p}_k^T\mathbf{t}$

$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k\mathbf{p}_k$

$\mathbf{r}_{k+1} = \mathbf{r}_k - \alpha_k\mathbf{t}$

$\mathbf{v} = M^{-1}\mathbf{r}_{k+1}$

$\beta_k = -\mathbf{v}^T\mathbf{t} / \mathbf{p}_k^T\mathbf{t}$

$\mathbf{p}_{k+1} = \mathbf{v} + \beta_k\mathbf{p}_k$

END FOR

Numerical kernels:

1. Vector update
2. Scalar Product
3. Matrix-vector product
4. Preconditioning

- Vector update:

$$\vec{x} \leftarrow \vec{x} + \alpha\vec{p}$$

is an *embarassingly parallel* operation

- Blocks of *n*/*p* consecutive components are assigned to each processor
- The update operations are independent each other

- Often, a good compiler is already able to exploit the presence of a multi-core processor

- Scalar product:

$$s = \vec{x}^T \vec{y} = \sum_{i=1}^{n} x_i y_i$$

is a *reduction* operation

- Blocks of *n*/*p* consecutive components are assigned to each processor
- Each processor computes its own contribute
- The scalar product is the result of a reduction of all the scalars computed by each processor

- The speed-up is close to be ideal if $n >> p$

- Matrix-vector product:

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

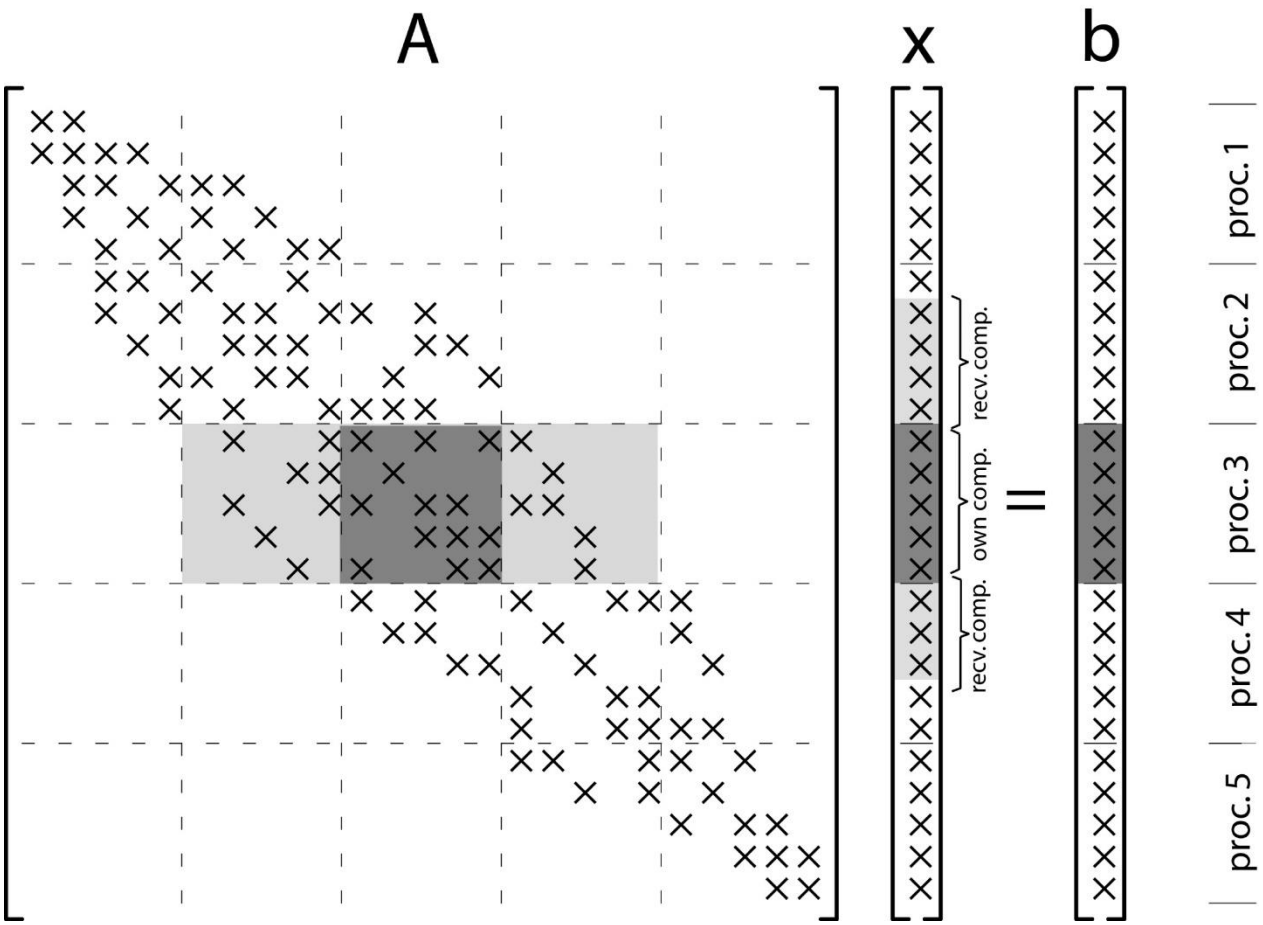
is the union of *n scalar products*

- Stripes of *n*/*p* consecutive rows and blocks of *n*/*p* consecutive components are assigned to each processor
- Each processor computes its own scalar products

- A possible limit is that a processor in principle may need to access the components of any other processor
- Communications are limited in case of linear systems arising from the Finite Element discretization of PDEs

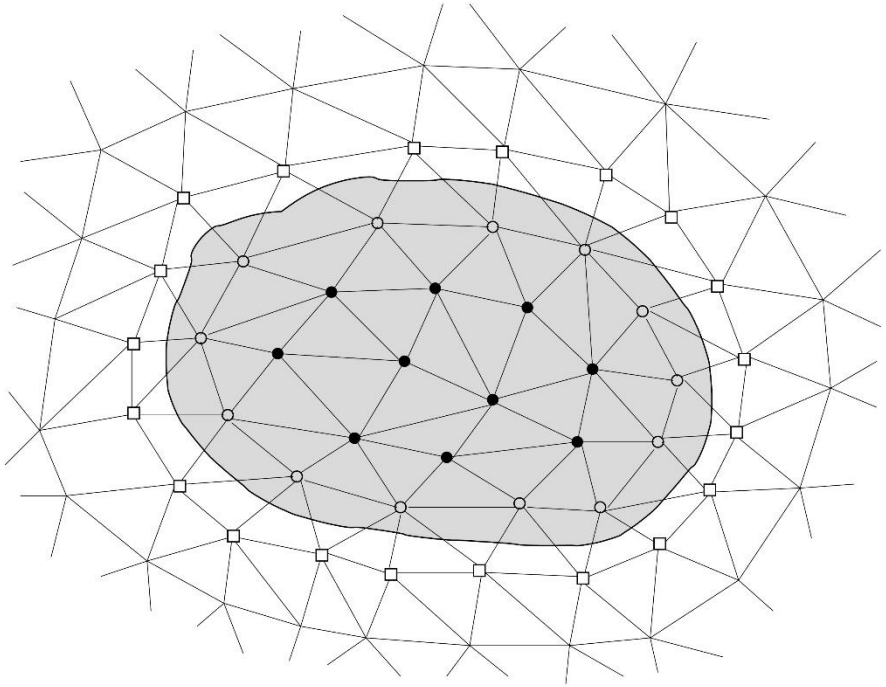
Parallel algorithms

Parallel numerical linear algebra



Parallel algorithms

Parallel numerical linear algebra



- If the linear system arises from the FE discretization of PDEs, each stripe of *A* refers to a *subdomain* of the computational grid
- The matrix partitioning coincides with a *domain decomposition* where *inner* and *boundary variables* can be identified

- The amount of interprocessor communications depends on the number of edges intercepted by each subdomain boundary
- An optimal domain decomposition can be obtained by appropriate *graph partitioning techniques* that minimize the boundaries shared by the subdomains

- Preconditioning:

$$\vec{v} = M^{-1}\vec{r}$$

where M^{-1} is the preconditioner

- The actual cost of this operation depends on the selected preconditioner
- This operation can be the actual bottleneck for a parallel linear system solution
- Sometimes, it is more convenient to use a non-optimal preconditioner, i.e., having a larger number of iterations to convergence, but highly parallel (e.g., Jacobi)
- The number of iterations to converge can change even significantly with the number of processors

- Example of preconditioning: incomplete Cholesky decomposition
- We need to solve a lower and an upper system:

$$L\vec{x} = \vec{b} \qquad \text{and} \qquad L^T\vec{v} = \vec{x}$$

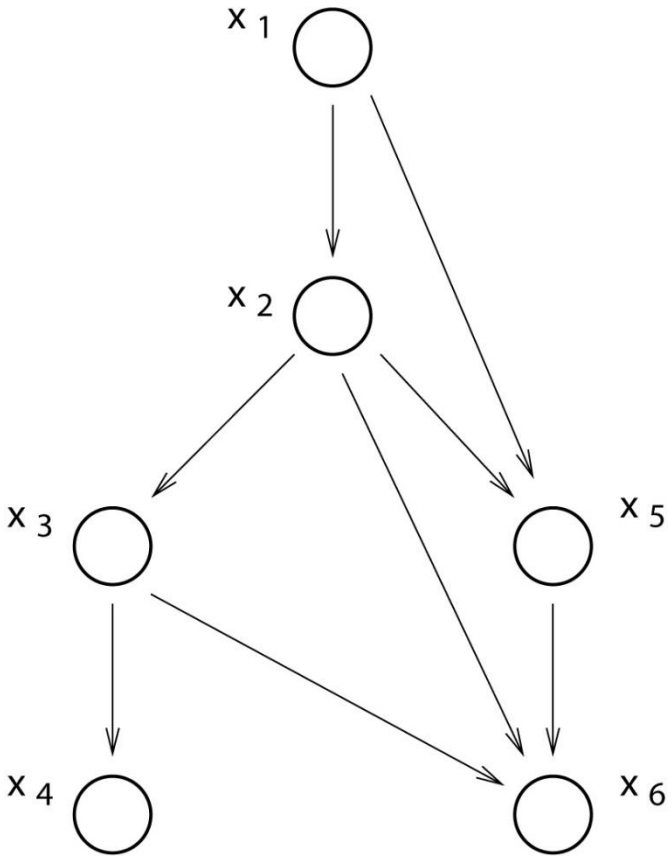
- If *L* were dense, a parallel solution is impossible
- However, *L* is sparse and some parallelism can be gained by *level scheduling*

- The efficiency of the parallel implementation is strictly dependent on the *unknown numbering* and progressively worsens as the number of processors grows

Parallel algorithms

Parallel numerical linear algebra

$$\begin{bmatrix} l_{11} & 0 & 0 & 0 & 0 & 0 \\ l_{21} & l_{22} & 0 & 0 & 0 & 0 \\ 0 & l_{32} & l_{33} & 0 & 0 & 0 \\ 0 & 0 & l_{43} & l_{44} & 0 & 0 \\ l_{51} & l_{52} & 0 & 0 & l_{55} & 0 \\ 0 & l_{62} & l_{63} & 0 & l_{65} & l_{66} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \end{bmatrix}$$

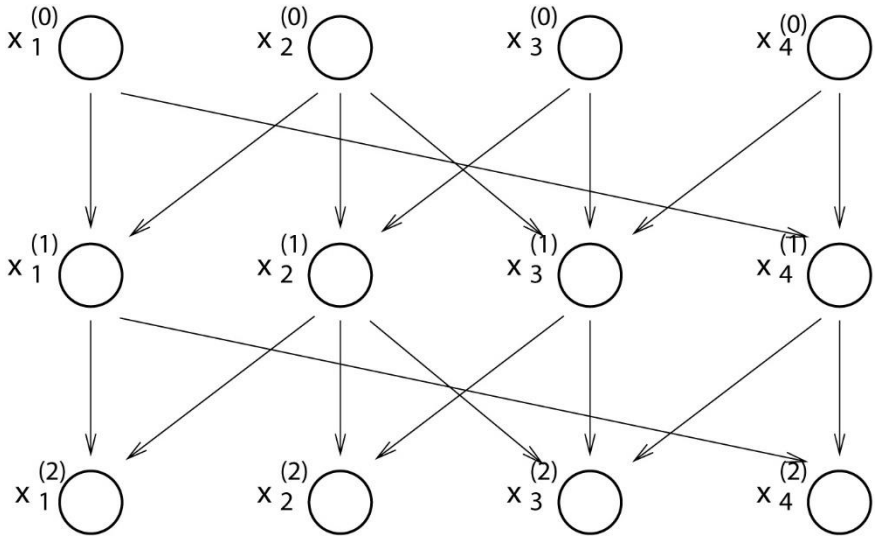


- Each level is made of the components that can be computed independently
- The maximum number of processors is equal to the maximum number of components belonging to a single level

- For a general iterative method in the form:

$$\vec{x}_{k+1} = f(\vec{x}_k)$$

parallelization can be performed by assigning a block of components to each processor with communications after each iteration



- A barrier with an alignment of all processors is necessary after each iteration, giving rise to a *synchronous method*
- The synchronization can be *global* or *local*
- For a better parallelization, we can use an *asynchronous* implementation
- Theoretical properties of the method are completely different

- Example: the Newton-Raphson iteration

$$x_{k+1} = x_k - f(x_k)/f'(x_k)$$

with an asynchronous implementation can become:

$$x_{k+1} = x_k - f(x_k)/f'(x_j) \qquad j \leq k$$

- Theoretical properties no longer hold, but the asynchronous method can be faster

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

DMMMSA

Department of Civil, Environmental and Architectural Engineering

End