Modelling and implementation of algorithms in applied mathematics using MPI

Lecture 1: Basics of Parallel Computing

G. Rapin

Brazil
March 2011

Outline

Outline

Structure of Lecture

Thu: Basics of Parallel Computing, Fan-in method, parallel treatment of vectors and matrices

Fri: Poisson Problem and Finite Differences; First Steps with MPI

Fri: Exercises in MPI - Part I

Tue: Linear Systems: Simple Iterative Methods and their parallelization, Prgramming MPI

Tue: Exercises in MPI - Part II

Wed: Conjugate Gradient (CG) method and preconditioning

Contents of the Lecture

- Basics of Parallel Computing
- Introduction to MPI
- Algorithms of Numerical Linear Algebra
- Paralellization of some algorithms
- Poisson Problem as prototype application
- Modern Iterative Solvers, like Krylov methods

Goals of the Lecture

- Understanding and Usage of MPI
- Explaining ideas of High Performance Computing
- Discussion of the Interplay of Algorithms and their Implementation
- Give some insight in Numerical Linear Algebra
- Show basic mathematical techniques to prove convergence

Literature - MPI

- Message Passing Interface Forum, *MPI: A message-passing interface standard*. University of Tennesse, Knoxville. download from *http://www.mpi-forum.org*.
- P.S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann Publishers. San Francisco. 1997.
- W. Gropp, E. Lusk, A.Skjellum. *Using MPI. Partable Parallel Programming with the Message-Passing-Interface*. Second Edition. MIT Press. Cambridge. 1999
- Blaise Barney, Introduction to Parallel Computing, On–Site tutorial, *https://computing.llnl.gov/tutorials/parallel_comp*
- Blaise Barney, Introduction to Parallel Computing, On–Site tutorial,*https://computing.llnl.gov/tutorials/mpi/*

Literature - Scientific Computing

- G.H. Golub, J.M. Ortega: Scientific Computing: An Introduction with Parallel Computing, Academic Press, 1993
- A. Quarteroni, A. Valli. *Domain Decomposition Methods for Partial Differential Equations*. Oxford University Press. Oxford. 1999
- J.W. Demmel, Applied Numerical Linear Algebra, SIAM, 1997
- Y. Saad, Iterative Methods for Sparse Linear Systems, Second Edition, SIAM, 2003
- T. Mathew, Domain Decomposition Methods for the Numerical Solution of Partial Differential Equations, Springer, 2008

Outline

Serial/ Parallel Computing

Serial Computing

- Software runs on a single computer with a single CPU.
- Problem is broken into a set of instructions.
- Instructions are executed one after another.
- There is only one execution at the same time.

Parallel Computing

- To be run on multiple CPUs
- A problem is broken into discrete parts that can be solved concurrently.
- Each part is further broken down to a series of instructions.
- Instructions from each part execute simultaneously on different CPUs.

Why Parallel Computing?

- Save Time
  speed-up tasks
- Save Money
  faster hardware is quite expensive; for parallel computing cheaper components can be used
- Compute Larger Problems
  Bottlenecks like limited memory can be circumvented
- Combine existing Computational Resources
  Existing Clusters can be used for parallel computing.

High Performance Computing (HPC)

Definition HPC (Wikipedia)

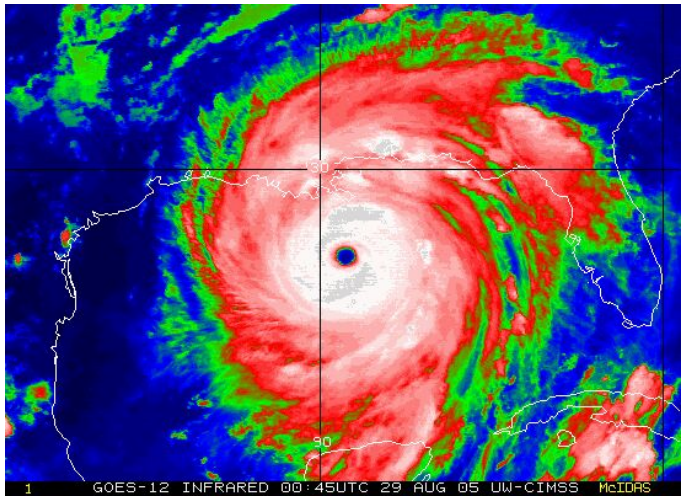
High-performance computing (HPC) uses supercomputers and computer clusters to solve advanced computation problems.

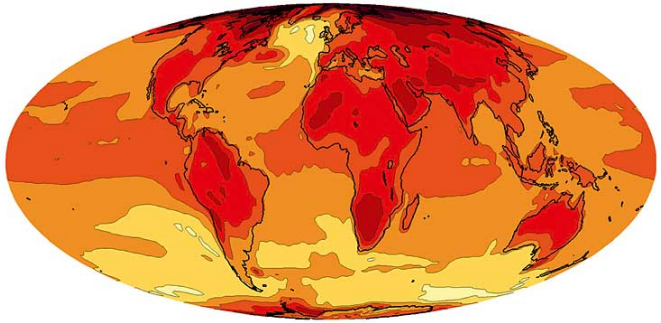
Characteristics of typical HPC applications

- parallelised
- large demand of memory
- requires large computing capacities
- applications mostly run on clusters and supercomputers
- handling of large data sets

Typical Applications for HPC

- Simulation of fluid flows
- Weather Forecasts
- Climate Modelling
- Chemical Models (Molecular dynamics, Combustion)
- Data Mining
- Physics (material science)





Classification of Parallel Computers

- Popular Classification is **Flynn's Taxonomy** (1966)
- distinguish with respect to Instructions and Data
- Instructions and Data have two categories Single or Multiple
- Single instruction: only one instruction stream is being acted on by the CPU during any one clock cycle
- Single data: only one data stream is being used as input during any one clock cycle

Flynn's Taxonomy

Examples for the four Possibilities

SISD older generation mainframes, minicomputers and workstations; most single CPU PCs

SIMD most modern computers, particularly those with graphics processor units (GPUs), vector pipelines like Cray Y-MP

MISD very rare, experimental Carnegie-Mellon C.mmp computer (1971).

MIMD most current supercomputers, networked parallel computer clusters and "grids", multi-core PCs

Memory Architecture

Shared Memory

- All processors have access to a global memory address space
- Advantages: user-friendly programming models, Data sharing between tasks is fast due to the proximity of memory to CPUs
- Disadvantages: lack of scalability between memory and CPUs; the programmer is responsible for a synchronized access to the data

Distributed Memory

- Processors have their own local memory. Data transfer requires a communication network.
- Advantages: memory is scalable with number of processors; cost effectiveness
- Disadvantages: Programming is complicated; complex data handling; almost all algorithms have to be adapted

Parallel Programming Models

Parallel programming models are on an abstraction level above hardware and memory architecture; NOT specific to a particular type of machine or memory architecture.

- Shared memory
  In a shared memory model, parallel tasks share a global address space which they read and write to asynchronously. This requires protection mechanisms such as locks and semaphores to control concurrent access. Example is OpenMP.
- Message passing model
  Parallel tasks exchange data through passing messages to one another. Example is MPI.
- Implicit model
  In an implicit model, no process interaction is visible to the programmer. The compiler is responsible for performing it. Examples are MATLAB or High Performance Fortran

Message Passing Interface (MPI)

- MPI is a specification for the developers and users of message passing libraries.
- It is NOT a library itself.
- The described interface should be portable, practical, efficient and flexible.
- Interface descriptions exist for C/C++ and Fortran.
- MPI is the 'de facto' industry standard for message passing.
- There exist a couple of implementations. The most popular MPI versions are OpenMPI, mpich2, IntelMPI or platformMPI.
- MPI is a single program with multiple data (SPMD). The same program is started on all processors.

History of MPI

- Before 1990 numerous approaches and libraries for parallel computing exist. Many hardware producers provide the customer with specialized implementations. Codes were not portable and differ significantly in performance.
- Starting point of MPI is a Workshop on Standards for Message Passing in a Distributed Memory Environment in Williamsburg, USA, in April 1992. A team working on a proposal was founded.
- First version of MPI was released in November 1992. Official releases are published in June 1994 (MPI 1.0) and June 1995 (MPI 1.1).
- Foundation of the MPI Forum in 1995. The MPI Forum publishes MPI 1.2 in 1997
- Essential Extensions like I/O-interfaces and C++-interfaces are defined in MPI 2. MPI 2.0 appeared 1998 in MPI 2.2 in 2009. The MPI 3 project was started in 2010.

Outline

Degree of Paralel ization

Definition

The *degree of parallelisation* of an algorithm is defined as the maximum number of parallel tasks.

Examples:

- Most parallel algorithms for the addition of two vectors with n components have a degree of parallelisation of n. Each component of the vector can be summed in parallel.
- We consider an iterative sequence $(x_n)_n$ of the type

$$x_{k+1} = f(x_k), k \in \mathbb{N}, \quad x_0 \in \mathbb{R}$$

with $f : \mathbb{R} \to \mathbb{R}$. The degree of parallelisation is 1 for standard algorithms.

Speed-up

Definition

Let T_1 be the time spent for the algorithm for the solution of a given problem on a single processor and T_p the time, which is needed for the algorithm on a parallel computer with p processors of the same type. Then, the *speed-up* is defined as

$$S = \frac{T_1}{T_P}.$$

Remark

- In the literature T_1 can also be defined as the time spent on a single processor for the solution of a given problem using the best known algorithm.

Incremental Speed-up

In many applications it is not possible to compute the problem on a single processor. Then, the *incremental Speed-up* can be used:

$$S_i(p) := \frac{\text{Runtime on } \frac{p}{2} \text{ processors}}{\text{Runtime on } p \text{ processors}}.$$

In the optimal case $S_i(p)$ is equal to 2. Normally, it holds. $2 \geq S_i(p) \geq 1$.

Parallel Efficiency

Definition

The Parallel Efficiency e of a parallel algorithm using p processors is defined by

$$e := \frac{S}{p} = \frac{T_1}{T_P p}$$

where S is the speed-up.

- It holds $0 \leq e \leq 1$. For e next to 1 the algorithm is quite efficient.
- If it is not possible to compute the solution on a single processor, the parallel efficiency can be approximated by

$$e \approx \frac{T_{p_{min}} p_{min}}{T_p p}.$$

Armdahl's Law

Theorem

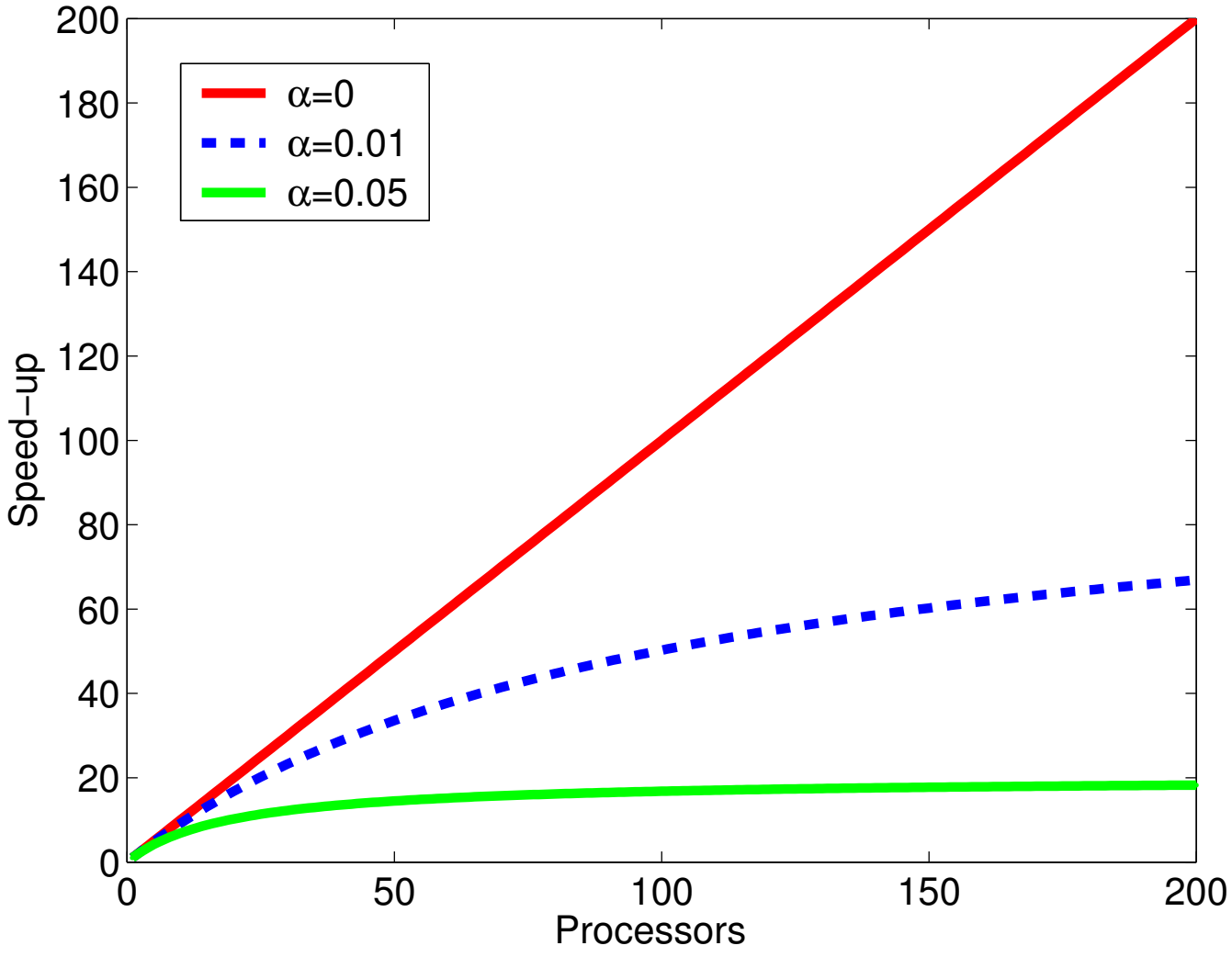
Let α be the sequential part of the algorithm. The remaining part $1-\alpha$ is executed on p processors with efficiency 1. Then the speed-up is

$$S = \frac{1}{\alpha + (1-\alpha)/p}.$$

Remarks

- The assumption, that the remaining part scales perfectly, is essential.
- Let us assume that 99% of the algorithm is parallelised. Thus, we get a maximum speed-up of 9.17 for 10 processors and a speed-up of less than 50 for 100 processors.

Visualization of Armdahl's Law



Proof of Armdahl's Law

- Let T_1 be the computational time on a single processor.
- The computational time for the computation on p processors assuming an efficiency of 1 for the parallel part is

$$T_p = \alpha T_1 + (1 - \alpha)\frac{1}{p} T_1.$$

- Then the speed-up is given by

$$S = \frac{T_1}{T_P} = \frac{1}{\alpha + (1 - \alpha)/p}.$$

Outline

Addition of Vectors

- Let $a = (a_1, \ldots, a_n)^T$ and $b = (b_1, \ldots, b_n)^T$ be two vectors.
- Goal: Compute the addition

$$c := a + b = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} = \begin{pmatrix} a_1 + b_1 \\ a_2 + b_2 \\ \vdots \\ a_n + b_n \end{pmatrix}$$

  in parallel.
- Usage of p processors. We assume $p \leq n$.

Decomposition of the problem

The n additions have to distributed to the p processors.

Definition

The system $I_R = (I_r)_r$ of sets I_r, $r = 1, \ldots, R$ is called *partition* of a set M, if

(i) $I_r \neq \emptyset$, $r = 1, \ldots, R$.

(ii)

$$I_r \cap I_q = \emptyset \quad \text{for } r \neq s,\ r, s \in \{1, \ldots, R\}.$$

(iii)

$$\bigcup_{r=1}^{R} I_r = M.$$

Parallel Addition of Two Vectors

Let (I_p) be a partition of $\{1, \ldots, n\}$.

Algorithm 1: **Parallel Addition of Two Vectors**

Solve on processor k, $k \in \{1, \ldots, p\}$

$$c_i = a_i + b_i, \quad i \in I_k.$$

Remark

- The subsets of the partition should almost have the same size.

Addition of the entries of a vector

- Let $x = (x_1, \ldots, x_n)^T$ be a vector.
- Compute the sum

$$s = \sum_{i=1}^{n} x_i.$$

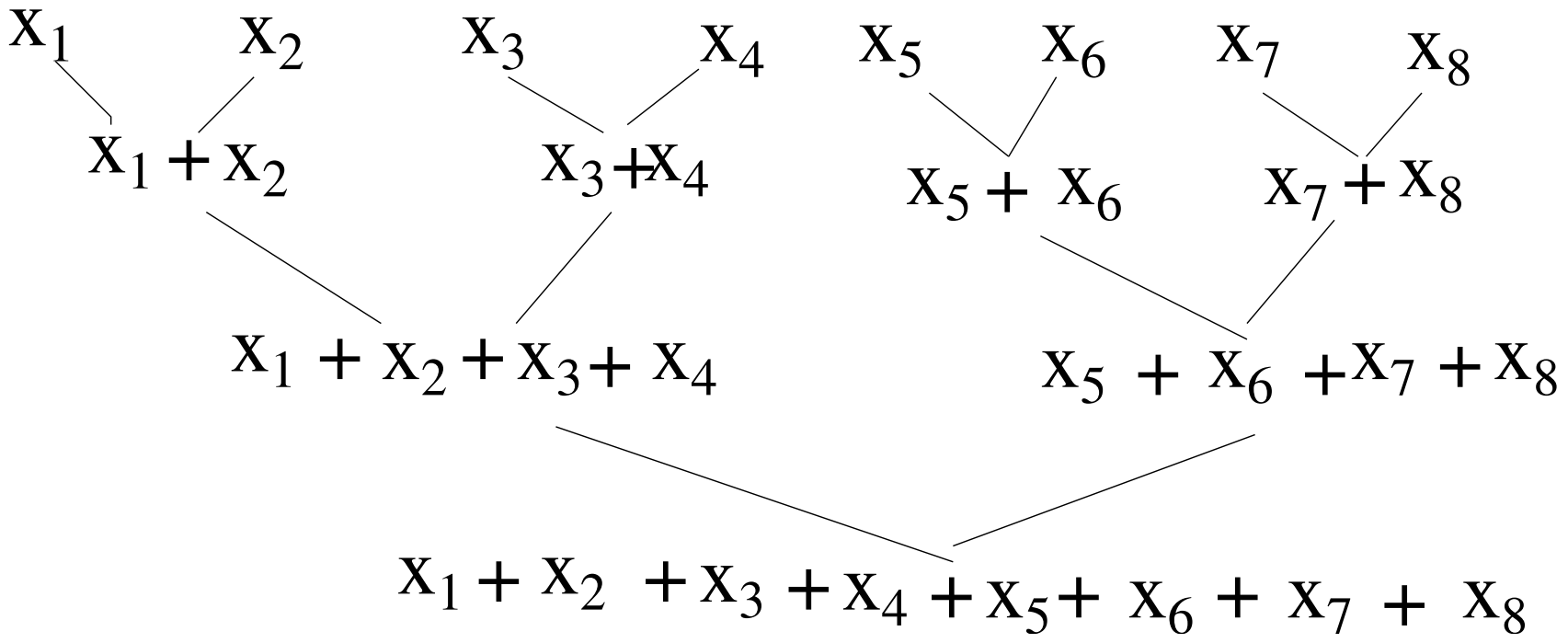
- Sequential Algorithm is almost trivial

Algorithm 2: **Sequential Algorithm**

$$\begin{aligned} s_1 &:= x_1, \\ s_{k+1} &:= s_k + x_{k+1}, \quad k = 1, \ldots, n-1. \\ s &= s_n \end{aligned}$$

Idea of Parallel Algorithm

Compute $s = \sum_{i=1}^{8} x_i$!



Parallel Algorithm for $n = 2^k$ summands

Algorithm 3: **Fan-in Summation**

1. *Set* $a_i^0 := x_i$ *for* $i = 1, \ldots, 2^k$.
2. *Compute for* $j = 1, \ldots, k$, $i = 1, \ldots, 2^{k-j}$

$$a_i^j := a_i^{j-1} + a_{i+2^{k-j}}^{j-1}.$$

3. *The solution is* $s := a_1^k$.

Remarks for Fan-in Summation

- Degree of Parallelisation is 2^{k-1}.
- The average degree of parallelisation is

$$\frac{n-1}{\log_2 n},$$

since

1. Step	degree of parallelisation 2^{k-1}
2. Step	degree of parallelisation 2^{k-2}
$\vdots$	$\vdots$
$(k-1)$. Step	degree of parallelisation 2^1
k. Step	degree of parallelisation 2^0

Therefore we get

$$\frac{1}{k}\sum_{i=0}^{k-1} 2^i = \frac{1}{k}\frac{2^k-1}{2-1} = \frac{1}{k}\left(2^k-1\right) = \frac{1}{\log_2 n}(n-1).$$

Remarks for Fan-in Summation

- Neglecting communication time the sum of $n = 2^k$ numbers with 2^{k-1} processors can be computed in a time, which needs one processor for $k = \log_2 n$ additions.
- Assume that communication needs the time $\kappa\tau$. τ is the time for one addition ($\kappa > 1$). Then, 2^{k_p} processors need for the sum of $n = 2^k$ numbers (with $k_p < k$) the time

$$t(k_p) = \left(2^{k-k_p} - 1\right)\tau + (\kappa\tau + \tau)k_p.$$

The function $t(k_p)$ has a global minimum at

$$k_0 := k - \frac{1}{\log 2} \log\left(\frac{\kappa + 1}{\log 2}\right).$$

For $\kappa = 10$ we get $k_0 \approx k - 3.99$.

Other Applications for Fan-in

More general the Fan-in method can be applied to arbitrary associative, commutative and binary operations

$$x = x_1 \circ x_2 \circ \cdots \circ x_n.$$

Some Examples

- $\prod_{i=1}^{n} a_i$ with $a_1 \circ a_2 = a \cdot b$,
- $\max\{a_i | i = 1, \ldots, n\}$ with $a_1 \circ a_2 = \max\{a_1, a_2\}$,
- $\min\{a_i | i = 1, \ldots, n\}$ with $a_1 \circ a_2 = \min\{a_1, a_2\}$,
- l_p norm, $1 < p < \infty$, i.e. $\left(\sum_{i=1}^{n} |a_i|^p\right)^{\frac{1}{p}}$ with $a_1 \circ a_2 = (|a_1|^p + |a_2|^p)^{\frac{1}{p}}$,
- $gcd\{r_i | i = 1, \ldots, n\}$ of numbers $r_1, \ldots, r_n$ with $r_1 \circ r_2 = gcd(r_1, r_2)$.

Scalar products

- Let $x = (x_1, \ldots x_n)^T$ and $y = (y_1, \ldots, y_n)^T$ be two vectors.
- The scalar product is given by

$$(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^{n} x_i y_i, \qquad \mathbf{x}, \mathbf{y} \in \mathbb{R}^n$$

- *Algorithm 4:* **Parallel scalar product**

1. *Define a partition* $I_P = (I_r)_r$ *of* $\{1, \ldots, n\}$
2. *Compute* $d_k = \sum_{i \in I_k} x_i y_i$ *for* $k \in \{1, \ldots, p\}$.
3. *Compute with Fan-in*

$$d = \sum_{i=1}^{p} d_i.$$

Product of Matrices

- Let two matrices be given:

$$A = (a_{ij}) \in \mathbb{R}^{n\times m}, \quad B = (b_{ij}) \in \mathbb{R}^{m\times q}.$$

- The goal is the computation of the matrix product

$$C = (c_{ij}) = AB \in \mathbb{R}^{n\times q}.$$

- We define the sums

$$c_{ij}^k := \sum_{l=1}^{k} a_{il} b_{lj}, \quad i \in \{1,\dots,n\},\ j \in \{1,\dots,q\},\ k \in \{1,\dots,m\}$$

  and $c_{ij}^0 := 0$, $i = 1,\dots,n$, $j = 1,\dots q$

- We can compute c_{ij} using

$$c_{ij}^k = c_{ij}^{k-1} + a_{ik} b_{kj},\ k = 1,\dots,m \quad \text{and} \quad c_{ij} = c_{ij}^m.$$

Matrix Computation - Sequential Algorithm

Algorithm 5: **Matrix Computation - ijk-Form**

1. *for* $i = 1$ *to* n
2. *for* $j = 1$ *to* q
3. *set* $c_{ij} = 0$
4. *for* $k = 1$ *to* m
5. $c_{ij} := c_{ij} + a_{ik} b_{kj}$
6. *end* k
7. *end* j
8. *end* i

Remark

- Depending on the storage pattern of the matrices a different order of the loops is better.

Parallel Matrix Multiplication

Idea :
Split the matrices in blocks

Assumptions

- Let (I_R) be a partition of $\{1, \ldots, n\}$, (K_S) a partition of $\{1, \ldots, m\}$ and (J_T) a partition of $\{1, \ldots, q\}$.
- Assume that $p = R \cdot S \cdot T$ processors are available.
- Assume that processor $P(r, s, t)$, $r \in \{1, \ldots, R\}$, $s \in \{1, \ldots, S\}$, $t \in \{1, \ldots, T\}$ has the data

$$a_{ik}, \qquad i \in I_r,\ k \in K_s$$
$$b_{kj}, \qquad k \in K_s,\ j \in J_T.$$

Parallel Matrix Multiplication

Algorithm 6: **Parallel Matrix Multiplication**

1. *Compute on each processor* $P(r, s, t)$
   (i) for $i \in I_r$
   (ii) for $j \in J_t$
   (iii) $\tilde{c}_{ij}^{s} := \sum_{k \in K_s} a_{ik} b_{kj}$
   (iv) end j
   (v) end i
2. *Compute for all* $r \in \{1, \ldots, R\}$, $t \in \{1, \ldots, T\}$ *the additive Fan-ins*

$$C_{rt} = \left(\sum_{s=1}^{S} \tilde{c}_{ij}^{s} \right)_{i \in I_r, j \in J_t}$$

with S processors.

Explanation of the Parallel Algorithm

- Processor $P(r, s, t)$ computes the matrix product of the blocks

$$A_{rs} = (a_{ik})_{i \in I_r, k \in K_s}, \quad B_{st} = (b_{kj})_{k \in K_s, j \in J_t}$$

  in step 1.
- During the Fan-in the blocks $C_{rt} = \sum_{s=1}^{S} A_{rs} B_{st}$ are computed using processors $P(r, \sigma, t)$, $\sigma = 1, \ldots, S$.
- Since each process contributes to exactly one Fan-in, all Fan-ins can be computed in parallel.
- The load balancing is for instance optimal when all sets I_r, K_s and J_t contains the same number of elements.

Special Choices for R, S and T

- Let be $S = 1$, thus $K_1 = \{1, \ldots, m\}$.
  Then there is no communication between the processors necessary. The Fan-in can be neglected.
- Let be $R = 1$, thus $I_1 = \{1, \ldots, n\}$.
  Each processor contains complete columns of A and a sub-block of B. There are T Fan-ins with S processors.
- Let be $R = 1$, $T = 1$, thus $I_1 = \{1, \ldots, n\}$, $J_1 = \{1, \ldots, q\}$.
  There is only one Fan-in with all processors.
- Let be $R = n$, $S = m$, $T = q$.
  Each processor computes a product of numbers. There are $n \cdot q$ parallel Fan-ins.