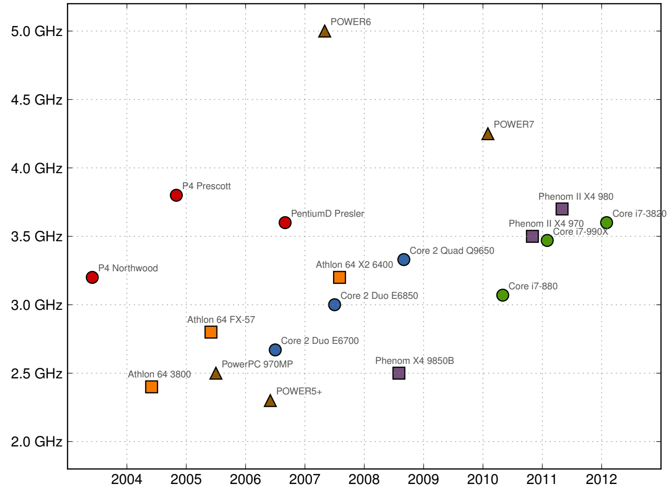
Introduction to Parallel Programming

Ronald Kriemann

Introduction

Why Go Parallel

Clock speed of processors in the last 10 years:



No longer hope for “free” performance increase by increased clock speeds (Sutter 2005).

Why Go Parallel

Instead of clock speed, processors have more and more *cores*:

Year	Processor	#cores
2006	Core 2 Duo	2
2007	Core 2 Quad	4
2010	Core i7-9xx	6
2010	Xeon 7xxx	8
2012	Xeon Phi 5110P	60
2005	Athlon 64 X2	2
2007	Phenom X4	4
2010	Phenom II X6	6
2010	Opteron 6100	8/12
2011	Opteron 6200	16

To speed up software, programmers have to use more execution paths simultaneously, e.g. design and implement *parallel algorithms*.

Why Go Parallel

List of Top 500 super computers of the last years (Wikipedia):

Nov 2012	18,688 CPUs (16 cores) + 18,688 GPUs	17.59 PFLOPS
Jun 2012	1,572,864 cores	13.32 PFLOPS
Jun 2011	88,128 CPUs (8 cores)	10.51 PFLOPS
Nov 2010	14,336 CPUs (6 cores) + 7,168 GPUs	2.57 PFLOPS
Nov 2009	37,376 CPUs (6 cores) + 14,336 CPUs (4 cores)	1.75 PFLOPS
Jun 2008	12,960 CPUs + 6,480 CPUs (2 cores)	1.04 PFLOPS
Nov 2004	106,496 CPUs (2 cores)	0.60 PFLOPS
Jun 2002	5120 vector processors	0.13 PFLOPS

Supercomputing Centers in Germany:

Jülich	: JUQUEEN, 458,752 cores, 4.1 PFLOPS
LRZ	: SuperMUC, 19,252 CPUs (8 cores / 10 cores), 2.9 PFLOPS
HLRS	: CRAY XE6, 7,104 CPUs (16 cores), 0.8 PFLOPS

Lecture Topics

Outline of the lecture:

1. Theoretical Foundations:
   - Parallel Programming Platforms
   - Parallel Algorithm Design
   - Performance Metrics
2. Vectorisation
   - Auto- and Manual Vectorisation
3. Programming Shared-Memory Systems
   - OpenMP
   - Thread Building Blocks
4. Programming Distributed-Memory Systems
   - Message Passing Interface

The lecture is (partly) based on A. Grama et al. 2003.

Parallel Programming Platforms

Parallel Programming Platforms

In contrast to sequential platforms, parallel platforms come in a great variety.

A basic classification may be according to

Control Flow

- Single-Instruction-Multiple-Data (SIMD),
- Single-Program-Multiple-Data (SPMD),

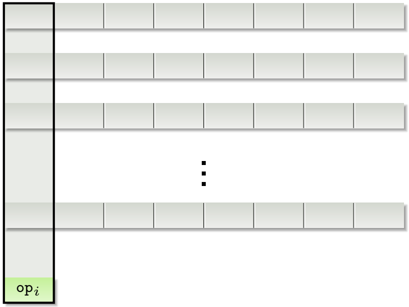
or

Communication Model

- Shared-Memory Machines,
- Distributed-Memory Machines.

SIMD

In a *single instruction multiple data* (SIMD) architecture, a central control unit performs the *same* instruction on *different* data streams.

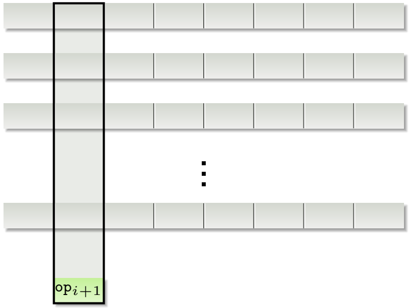


Here, the instruction stream goes synchronous with the data stream on all processing units.

An example for such an architecture are *vector machines* or *vector units* in processors. Handling the latter is discussed in the section about *Vectorisation*.

SIMD

In a *single instruction multiple data* (SIMD) architecture, a central control unit performs the *same* instruction on *different* data streams.

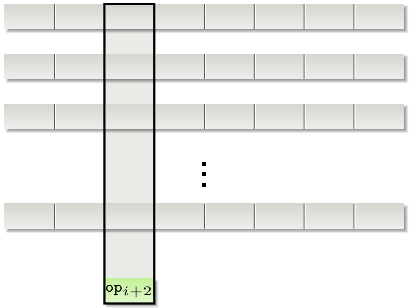


Here, the instruction stream goes synchronous with the data stream on all processing units.

An example for such an architecture are *vector machines* or *vector units* in processors. Handling the latter is discussed in the section about *Vectorisation*.

SIMD

In a *single instruction multiple data* (SIMD) architecture, a central control unit performs the *same* instruction on *different* data streams.

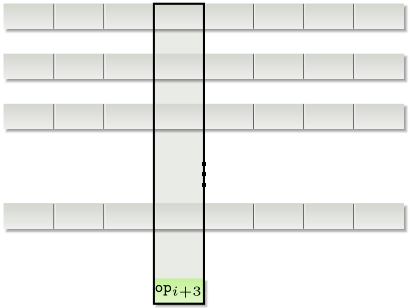


Here, the instruction stream goes synchronous with the data stream on all processing units.

An example for such an architecture are *vector machines* or *vector units* in processors. Handling the latter is discussed in the section about *Vectorisation*.

SIMD

In a *single instruction multiple data* (SIMD) architecture, a central control unit performs the *same* instruction on *different* data streams.



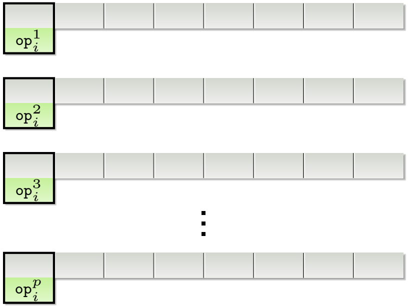
Here, the instruction stream goes synchronous with the data stream on all processing units.

An example for such an architecture are *vector machines* or *vector units* in processors. Handling the latter is discussed in the section about *Vectorisation*.

SPMD

Having multiple instructions working on multiple data stream is referred to as *MIMD* architectures, which can easily be implemented in a *single program* starting with various `if` blocks, each for a different *task*, leading to *single-program multiple-data* machines (SPMD).

Each parallel execution path in such a program may be completely independent from all other paths, but there may be various synchronisation points, e.g. for data exchange.

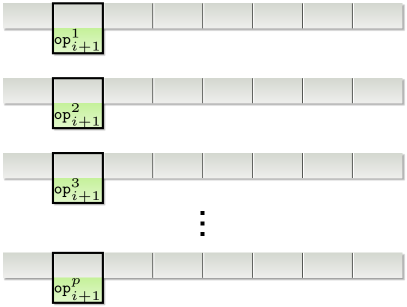


Most programs using multiple *threads* (see OpenMP/TBB) or *message passing* (see MPI) are based on the SPMD scheme.

SPMD

Having multiple instructions working on multiple data stream is referred to as *MIMD* architectures, which can easily be implemented in a *single program* starting with various `if` blocks, each for a different *task*, leading to *single-program multiple-data* machines (SPMD).

Each parallel execution path in such a program may be completely independent from all other paths, but there may be various synchronisation points, e.g. for data exchange.

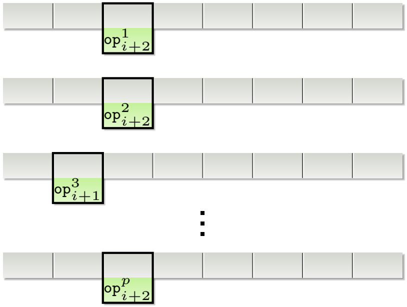


Most programs using multiple *threads* (see OpenMP/TBB) or *message passing* (see MPI) are based on the SPMD scheme.

SPMD

Having multiple instructions working on multiple data stream is referred to as *MIMD* architectures, which can easily be implemented in a *single program* starting with various `if` blocks, each for a different *task*, leading to *single-program multiple-data* machines (SPMD).

Each parallel execution path in such a program may be completely independent from all other paths, but there may be various synchronisation points, e.g. for data exchange.

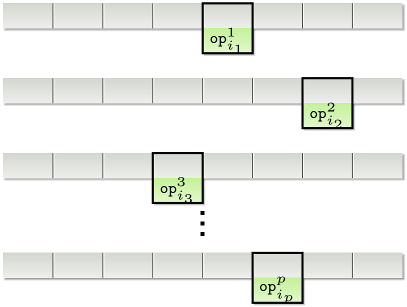


Most programs using multiple *threads* (see OpenMP/TBB) or *message passing* (see MPI) are based on the SPMD scheme.

SPMD

Having multiple instructions working on multiple data stream is referred to as *MIMD* architectures, which can easily be implemented in a *single program* starting with various `if` blocks, each for a different *task*, leading to *single-program multiple-data* machines (SPMD).

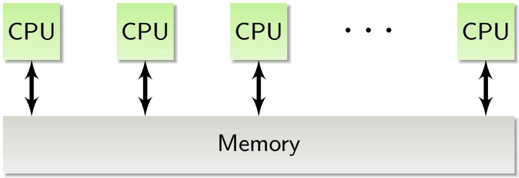
Each parallel execution path in such a program may be completely independent from all other paths, but there may be various synchronisation points, e.g. for data exchange.



Most programs using multiple *threads* (see OpenMP/TBB) or *message passing* (see MPI) are based on the SPMD scheme.

Shared-Memory Machines

In a shared memory system, all processors are connected to a shared memory, i.e. every memory position is directly accessible (read/write) by all processors.



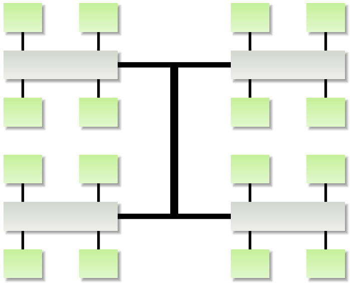
All communication between different processors is performed by changing data stored in the shared memory.

Remark

Instead of CPU one may also refer to each (logical) core within a physical processor.

Shared-Memory Machines

Often, the memory is constructed in some hierarchical form with *local* and *remote* (or global) memory for each processor.



If the time to access a single data in memory is independent from the memory position, e.g. local or remote memory, it is called *uniform memory access* (UMA). Otherwise, it is called *non uniform memory access* (NUMA). The difference between both types is important for the design of parallel algorithms.

Remark

Processor cache is not considered for the classification of UMA or NUMA.

Shared-Memory Machines

Shared-Memory vs. Shared-Address-Space

Different processes (programs) running on a shared memory usually do *not* have a shared address space, e.g. processes are not allowed to access the memory of other processes.

Threads (see OpenMP/TBB) are the most widely used form of having a shared address space for different execution paths.

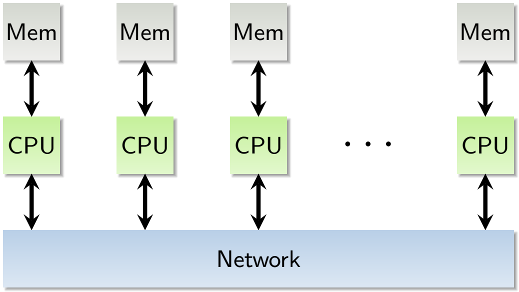
Another way is defined by "shared memory" in terms of *inter-process communication* (IPC, see W.R. Stevens 1998), a set of different methods for exchanging data between different processes. IPC is not discussed in this lecture!

Remark

In this lecture, shared memory is considered the same as shared address space.

Distributed-Memory Machines

In a distributed memory system, each processor has a *dedicated, local memory*, e.g. only the *local* processor may access data within this memory. All exchange of data is performed via a *communication network*.

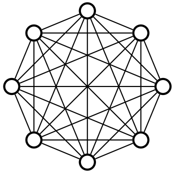


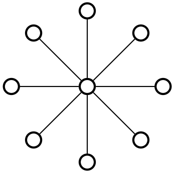
The standard communication paradigm on distributed memory platforms is *message passing* (see MPI).

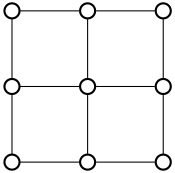
As before, distributed memory platforms are considered as *distributed address space* systems, e.g. may also refer to different processes on a shared memory system using the shared memory as the communication layer.

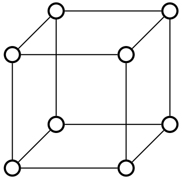
Distributed-Memory Machines

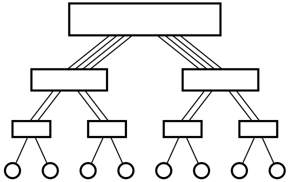
The communication network may come in a wide variety of different forms.


fully connected


Star


2D-Mesh


Hypercube


Fat Tree

The network *topology* has, in principle, a significant influance on the design and efficiency of parallel algorithms.

Distributed-Memory Machines

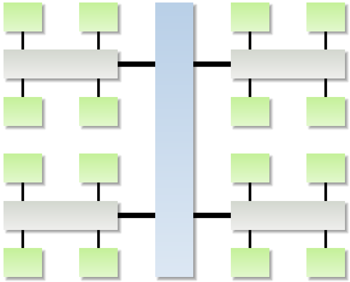
Distributed memory systems may also be programmed using a shared address space, which is implemented by some extra software, e.g.

- RDMA (*Remote Direct Memory Address*, also part of MPI),
- PGAS (*Partitioned Global Address Space*)

Such methods simplify programming, but each remote memory access involves a send/receive operation and hence, creates extra costs.

Hybrid Systems

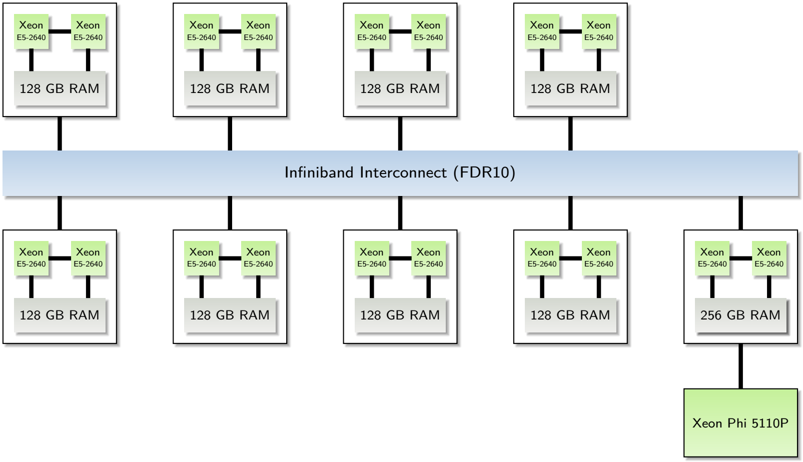
The majority of parallel systems follows a hybrid approach: a set of shared memory systems connected via a network.



Programming such systems can be done by using a combination of the above described techniques, e.g. threads + message passing, or with message passing alone, using IPC on shared memory sub-systems. The optimal approach is heavily dependent on the specific algorithm.

Hybrid Systems

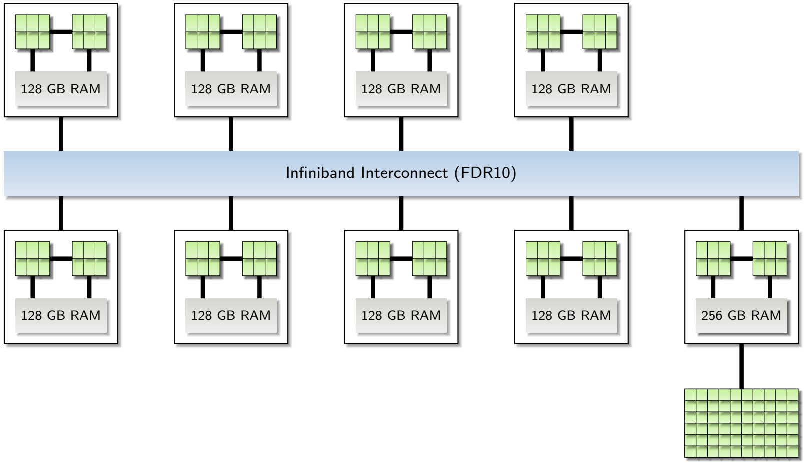
Our locally installed compute cluster also follows the hybrid approach:



Compute cluster at “Rechenzentrum Garching” of the MPG has (almost) the same configuration but with *610* nodes.

Hybrid Systems

Our locally installed compute cluster also follows the hybrid approach:



Compute cluster at "Rechenzentrum Garching" of the MPG has (almost) the same configuration but with *610* nodes.

Parallel Algorithm Design

Tasks and DAGs

Decomposition

For a parallel computation, the work has to be divided into smaller parts, which then may be executed in parallel. This process is called *decomposition*.

Task

Each part of the computation as defined by the programmer due to decomposition is called a *task*. A task is considered an *atomic* computational unit, i.e. is not divided any further.

The goal is to define as many *independent* tasks as possible to compute as much as possible in parallel.

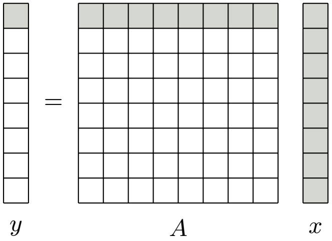
Different tasks of the same computation may be of different size.

Tasks and DAGs

As an example, consider the dense matrix-vector multiplication $y = Ax$, with $A \in \mathbb{R}^{n \times n}$:

$$y_i := \sum_j a_{ij} x_j$$

which may be split into tasks for each dot-product of the rows $A_{i,\bullet}$ with the vector x:

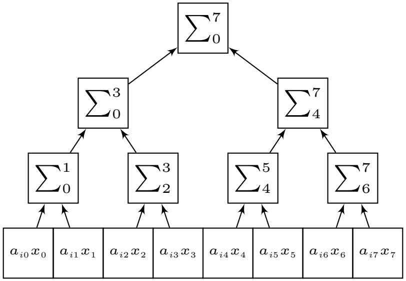


Each of these n tasks may be computed in parallel as no dependency exists between them.

Tasks and DAGs

The set of tasks could be further enlarged by dividing the dot-product computation into smaller tasks, i.e. one task per $a_{ij}x_j$. That way, we end up with n^2 independent tasks and may utilise even more parallel resources.

But, the computation of all products $a_{ij}x_j$ is only the first step in the computation of the actual dot-product as all results have to be summed up. Consider the following summation procedure:

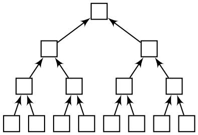


Tasks for the computation of $d^i_{k\ell}$ *depend* on the results of other tasks. The dependency connection between all tasks forms a *directed acyclic graph* (DAG), the *task-dependency graph*.

Tasks and DAGs

Handling each sum in a separate task results in an additional $n-1$ tasks per row.

But due to the dependencies, not all these tasks can be executed in parallel, which reduces the *concurrency* of the parallel program.



Concurrency

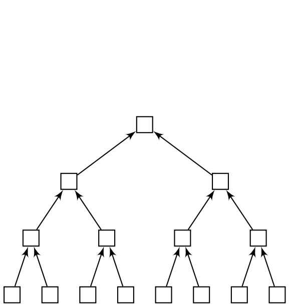
For a given set of tasks with dependencies, the maximal number of tasks which can be executed simultaneously is known as the *maximal degree of concurrency* (maximal concurrency).

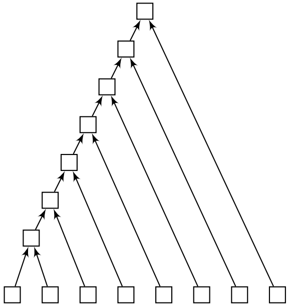
The maximal concurrency is usually *less* then the number of tasks.

Oten more important is the *average degree of concurrency* (average concurrency), i.e. the average number of tasks to run simultaneously during the runtime of the program.

Tasks and DAGs

Below are two task-dependency graphs for the dot-product with an equal maximal concurrency, but a different average concurrency.





For a general computation, the optimal DAG is dependent on the costs per tasks (see below).

Tasks and DAGs

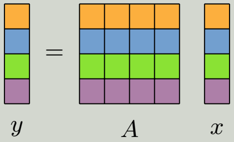
Task Interaction

Tasks dependence usually exists because the output of one task is the input of another task.

But even between independent tasks, a hidden dependence may exist, e.g. if (initial) input data is stored on different tasks.

Example: Dense Matrix-Vector Multiplication

Consider the dense matrix-vector multiplication $y = Ax$ on four processors, with row-wise tasks and a index-wise decomposition of the vectors.



Each task needs the entries of the vector x stored on all other tasks for the local computations. Hence, on a distributed memory system an initial send/recieve step is neccessary, before the actual computations may start.

Such *task interaction* may limit the gain of using parallel resources.

Tasks and DAGs

Granularity

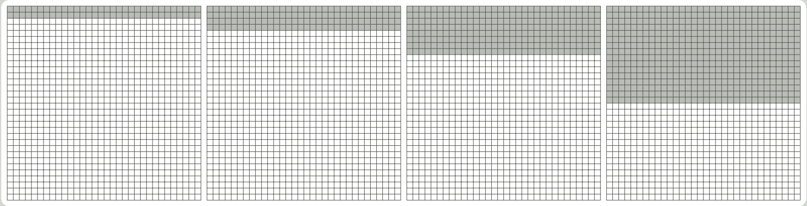
Handling all multiplications $a_{ij}x_j$ in a different task results in the maximal number of tasks possible for computing the matrix-vector product.

The opposite extreme would be the computation of the whole product in a single task.

The number and size of tasks determines the *granularity* of the decomposition. The more tasks the more *fine-grained* the decomposition is, whereas a small number of tasks yields a *coarse-grained* decomposition.

Example: Dense Matrix-Vector Multiplication

A block approach with each task computing the dot-product for several rows leads to a different granularity:



Decomposition Techniques

Designing a parallel algorithm starts with the *decomposition* of the work into tasks.

The specific way, in which the work is decomposed is heavily dependent on the algorithm and the computer architecture. Furthermore, different decompositions may lead to different runtime or complexity of the final program.

Two general techniques have proven helpful as a starting point:

recursive decomposition:	based on the structure of the algorithm,
data decomposition:	based on the layout of the data

The following two techniques may help for special problems:

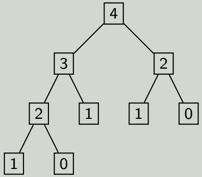
exploratory decomposition:	to *explore* a search space,
speculative decomposition:	to handle different branches in parallel

Recursive Decomposition

Recursive Decomposition uses a *divide-and-conquer* strategy to decompose the problem into *independent* sub-problems, to which the decomposition is then recursively applied.

Example: Fibonacci number

```
long  fib ( long  n ) {
  if ( n < 2 )
    return 1;
  else {
    long  f1 = spawn_task( fib( n−1 ) );
    long  f2 = spawn_task( fib( n−2 ) );

    return f1+f2;
  }
}
```



Recursive decomposition creates independent and therefore concurrent tasks.

Furthermore, overhead associated with task creation may also be distributed among the parallel computing resources.

Recursive Decomposition

Recursive decomposition may often also be applied, if the standard sequential algorithm does not use recursion.

Example: dot-product

The standard sequential implementation uses a single loop:

```
double dot_product ( int n, double * x, double * y ) {
  double  d = 0.0;

  for ( int  i = 0; i < n; ++i )
    d = d + x[i]*y[i];

  return d;
}
```

which can be reformulated using recursion by splitting the vector in half:

```
double dot_product ( int n, double * x, double * y ) {
  if ( n == 1 )
    return x[0]*y[0];
  else {
    double d1 = spawn_task( dot_product( n/2, x, y ) );
    double d2 = spawn_task( dot_product( n/2, x+n/2, y+n/2 ) );

    return d1+d2;
  }
}
```

Data Decomposition

Decomposition based on the involved data is done in two steps:

1. define the actual decomposition of the data, e.g. onto different processors, and then
2. decompose the computation based on the data decomposition into tasks.

Different algorithm data may be used to define the data decomposition:

Output Data:	decompose the results of the computation,
Input Data:	decompose the input data,
Intermediate Data:	decompose auxiliary data appearing during the computation.

Furthermore, combinations of the above may be used for the decomposition.

Data Decomposition

Partitioning Output Data

Consider the matrix multiplication

$$A \cdot B = C$$

with matrices $A, B, C \in \mathbb{R}^{n \times n}$.

The multiplication shall be performed *block-wise*

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

with sub matrices $A_{ij}, B_{ij}, C_{ij} \in \mathbb{R}^{n/2 \times n/2}$ of A, B and C, respectively.

Thus, the multiplication decomposes into the multiplications

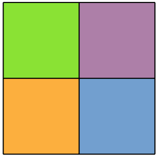
$$\begin{aligned} C_{11} &:= A_{11}B_{11} + A_{12}B_{21}, \\ C_{12} &:= A_{11}B_{12} + A_{12}B_{22}, \\ C_{21} &:= A_{21}B_{11} + A_{22}B_{21}, \\ C_{22} &:= A_{21}B_{12} + A_{22}B_{22} \end{aligned}$$

Data Decomposition

Partitioning Output Data

The output of the matrix multiplication is the matrix C, which computation defines four tasks:

Task 1: $C_{11} := A_{11}B_{11} + A_{12}B_{21}$,
Task 2: $C_{12} := A_{11}B_{12} + A_{12}B_{22}$,
Task 3: $C_{21} := A_{21}B_{11} + A_{22}B_{21}$,
Task 4: $C_{22} := A_{21}B_{12} + A_{22}B_{22}$



“Task Layout” of C

Remark

Using the eight sub multiplications, the work may even be split with a finer granularity:

$$\begin{array}{ll} C_{11} \leftarrow A_{11}B_{11}, & C_{11} \leftarrow A_{12}B_{21}, \\ C_{12} \leftarrow A_{11}B_{12}, & C_{12} \leftarrow A_{12}B_{22}, \\ C_{21} \leftarrow A_{21}B_{11}, & C_{21} \leftarrow A_{22}B_{21}, \\ C_{22} \leftarrow A_{21}B_{12}, & C_{22} \leftarrow A_{22}B_{22} \end{array}$$

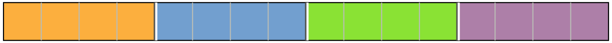
with $\leftarrow$ denoting either an initialisation $C_{ij} := A_{ik}B_{kj}$ or an update $C_{ij} := C_{ij} + A_{ik}B_{kj}$ of the destination matrix.

Data Decomposition

Partitioning Input Data

Consider the dot-product computation: the output is just a single value. In comparison, the input data may be arbitrary large.

Instead of the previous recursion, the input data may be decomposed explicitly into blocks of size n/p:



The computation starts at the per-process input blocks and combines the results afterwards:

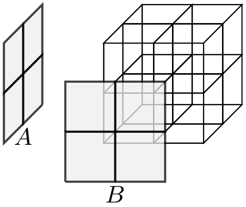
```
double dot_product ( int n_loc, double * x_loc, double * y_loc ) {
  double  d_loc = 0;

  for ( int i = 0; i < n_loc; ++i )
    d_loc = d_loc + x_loc[i]*y_loc[i];

  combine( d_loc );
}
```

Data Decomposition

Partitioning Intermediate Data

Back at the matrix multiplications: the sub multiplications of $A \cdot B = C$

$$\begin{aligned}
D_{111} &= A_{11}B_{11}, & D_{121} &= A_{12}B_{21}, \\
D_{112} &= A_{11}B_{12}, & D_{122} &= A_{12}B_{22}, \\
D_{211} &= A_{21}B_{11}, & D_{221} &= A_{22}B_{21}, \\
D_{212} &= A_{21}B_{12}, & D_{222} &= A_{22}B_{22}
\end{aligned}$$



form a $2 \times 2 \times 2$ tensor D with $C_{ij} := \sum_k D_{ikj}$.

Each of the eight $D_{ikj} := A_{ik}B_{kj}$ defines a task. In addition, the sums $C_{ij} := \sum_k D_{ikj}$ lead to another four tasks.

Although the memory consumption is increased, the granularity is finer and the concurrency is higher compared to the previous approaches.

Remark

In contrast to the decomposition based on the output data, no interaction exists between the eight D_{ikj} tasks.

Task Mapping and Load Balancing

Up to now, the individual tasks for the computation of the problem have been defined.

Next step is the *mapping* of these tasks onto the processors, with a given objective. Typical objectives are minimal

- *total runtime* (by far the most important!) or
- *memory consumption*, e.g. with limited memory per node in a distributed memory system.

Task interaction or dependence results in *overhead*:

- data has to be exchanged between processors or,
- processors are *idle* because input data is not yet computed.

Furthermore, processors may be idle due to bad *load balancing*, e.g. different computing costs per task may lead to some processors finished before others.

Task Mapping and Load Balancing

For a minimal runtime

- reducing interaction and
- reducing idling

are the main objectives for the task mapping.

Remark

Minimal interaction and idling are usually conflicting with each other, e.g. minimal interaction may be achieved by assigning all tasks to one processor, leaving all others idle.

Mapping techniques may be classified by the time when the mapping is applied:

Static Mapping	:	maps tasks to processors *before* algorithm execution or
Dynamic Mapping	:	maps tasks to processors *during* algorithm execution.

Static Mapping

As the mapping is computed a priori, knowledge of task size, task interaction and dependence should be available. As a rule: the better the knowledge, the better the mapping.

Unfortunately, even for known task sizes, the mapping problem is *NP-hard* (Garey and Johnson 1979), but good heuristics are available, e.g. List (Garey and Johnson 1979) or Multifit scheduling (Coffman, Garey, and Johnson 1978).

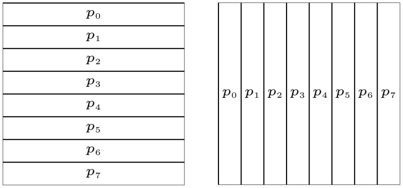
Beside the general mapping algorithms, concrete properties of the algorithm or the involved data may be used for mapping the tasks, e.g.

- data decomposition induces mapping (or vice versa!) and
- task dependence/interaction defines mapping

Array Distributions

A given d-dimensional array is partitioned such that each process receives a *contiguous* block along a subset of the array dimensions.

For $A \in \mathbb{R}^{n \times n}$, one-dimensional distributions may either be according to the rows or the columns of the matrix:



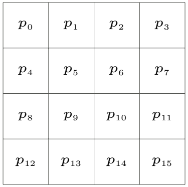
row-wise column-wise

In the latter distribution, processor $p_i, 0 \leq i < p$, will hold the columns

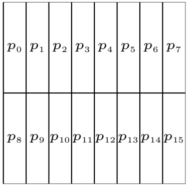
$$\left[\begin{array}{c|ccc|c} \cdots & i \cdot \frac{n}{p} & \cdots & (i+1) \cdot \frac{n}{p} - 1 & \cdots \end{array} \right]$$

Array Distributions

In two-dimensional distributions the partitioning of the rows and columns may be independent:


p_0 p_1 p_2 p_3
p_4 p_5 p_6 p_7
p_8 p_9 p_{10} p_{11}
p_{12} p_{13} p_{14} p_{15}

4×4 distribution


p_0 p_1 p_2 p_3 p_4 p_5 p_6 p_7
p_8 p_9 p_{10} p_{11} p_{12} p_{13} p_{14} p_{15}

2×8 distribution

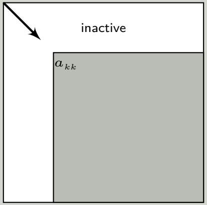
Let b_r and b_c the number of block rows and columns, then each processor will hold $n/b_r \times n/b_c$ entries of the matrix.

Cyclic Array Distributions

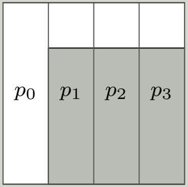
In many application, a contiguous mapping will lead to processor idling.

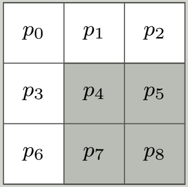
Example: LU Factorisation

```
void lu ( int n, Matrix & A ) {
   for ( int k = 0; k < n; ++k ) {
      // compute column
      for ( int j = k; j < n; ++j )
         A(j,k) = A(j,k) / A(k,k);
      // update trailing matrix
      for ( int j = k+1; j < n; ++j )
         for ( int i = k+1; i < n; ++i )
            A(i,j) −= A(i,k) / A(k,j);
  }
}
```



With previous mappings, the larger k the more processors will be idle:





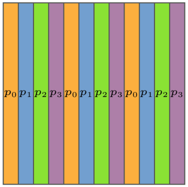
Cyclic Array Distributions

Alternatively, the array distribution is applied *cyclically*, with $1 \leq n_c \leq n/p$ being the number of cycles.

For the one-dimensional array distribution, the n rows/columns will be divided into n_c groups of n/n_c contiguous rows/columns. Each of these groups is then divided using the default array distribution, e.g. processor p_i will hold rows/columns

$$\left(j \frac{n}{n_c} + i \cdot \frac{n}{n_c p} \right) \quad \ldots \quad \left(j \frac{n}{n_c} + (i+1) \cdot \frac{n}{n_c p} - 1 \right)$$

for $0 \leq j < n_c$:



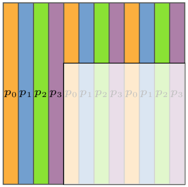
Cyclic Array Distributions

Alternatively, the array distribution is applied *cyclically*, with $1 \leq n_c \leq n/p$ being the number of cycles.

For the one-dimensional array distribution, the n rows/columns will be divided into n_c groups of n/n_c contiguous rows/columns. Each of these groups is then divided using the default array distribution, e.g. processor p_i will hold rows/columns

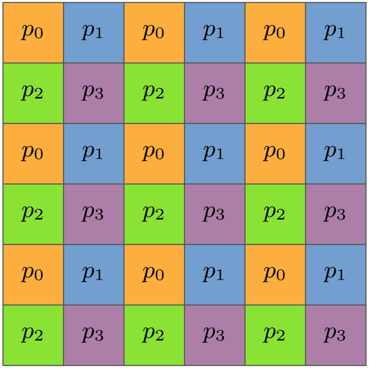
$$\left(j \frac{n}{n_c} + i \cdot \frac{n}{n_c p} \right) \quad \dots \quad \left(j \frac{n}{n_c} + (i+1) \cdot \frac{n}{n_c p} - 1 \right)$$

for $0 \leq j < n_c$:



Cyclic Array Distributions

For the two-dimensional case the cycles are applied to block rows and columns, i.e. divide the rows and columns in blocks of size n/n_c and apply the two-dimensional distribution to each subblock, i.e. split each $n/n_c \times n/n_c$ block into $b_r \times b_c$ subblocks.

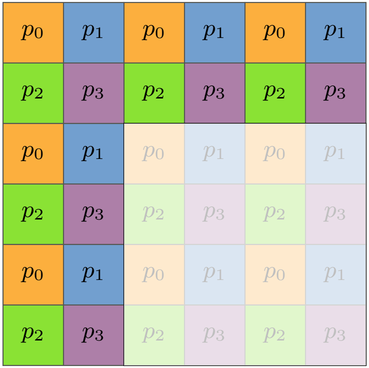


Remark

The cycle number n_c may also be different for the rows and columns.

Cyclic Array Distributions

For the two-dimensional case the cycles are applied to block rows and columns, i.e. divide the rows and columns in blocks of size n/n_c and apply the two-dimensional distribution to each subblock, i.e. split each $n/n_c \times n/n_c$ block into $b_r \times b_c$ subblocks.


p_0 p_1 p_0 p_1 p_0 p_1
p_2 p_3 p_2 p_3 p_2 p_3
p_0 p_1 p_0 p_1 p_0 p_1
p_2 p_3 p_2 p_3 p_2 p_3
p_0 p_1 p_0 p_1 p_0 p_1
p_2 p_3 p_2 p_3 p_2 p_3

Remark

The cycle number n_c may also be different for the rows and columns.

Cyclic Array Distributions

Since the data or work for each processor is spread over the *whole* array (matrix), an *automatic* load balancing is performed, reducing the chance for idling:

- for algorithms working on different parts of the data at different algorithms steps, e.g. LU factorisation, and
- for algorithms with different costs per array entry.

Remark

For some algorithms, a randomisation of the block assignment may lead to an even better load balancing.

Cyclic vs Block Cyclic Distribution

For $n_c = n$, only

- a single row (column) in the one-dimensional case or
- a single matrix entry for the two-dimensional distribution

is cyclically assigned to a processor.

This is originally known as the *cyclic distribution*, whereas all other cases are known as *block cyclic distributions*.

Graph Partitioning

The task interaction of algorithms may often be represented by a sparse graph. Furthermore, computations only apply to localised data. Typical examples are *mesh* based algorithms, e.g. PDEs.

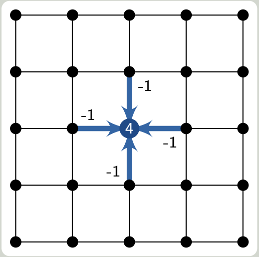
Example: Sparse Matrix-Vector Multiplication

Consider

$$-\Delta u = 0 \qquad \text{in } \Omega = [0,1]^2$$

on a uniform grid discretised with a standard 5-point stencil, and vector updates computed as

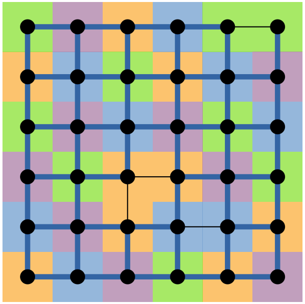
$$x'_{ij} := 4x_{ij} - x_{i-1,j} - x_{i+1,j} - x_{i,j-1} - x_{i,j+1}$$



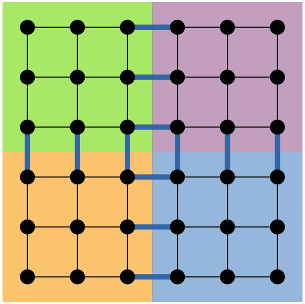
The interaction between all mesh nodes forms a graph, identical to the mesh itself.

Graph Partitioning

Applying a random distribution to the mesh nodes leads to a high amount of communication, whereas using a *localised* distribution only creates interaction at processor boundaries:



randomised



localised

A localised distribution can be computed using *graph partitioning*.

Graph Partitioning

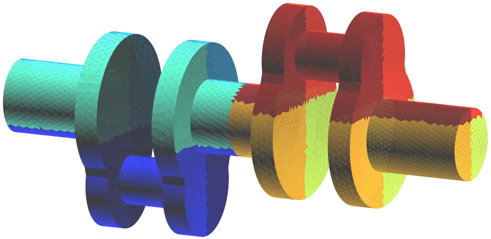
Given a graph $G = (V, E)$ *a partitioning* $P = \{V_1, V_2\}$, *with* $V_1 \cap V_2 = \emptyset, \#V_1 \sim \#V_2$ *and* $V = V_1 \cup V_2$, *of* V *is sought, such that*

$$\#\{(i,j) \in E \ : \ i \in V_1 \wedge j \in V_2\} \text{ is minimal.}$$

For a partitioning into more sets, graph partitioning may be applied recursively.

A small number of connecting edges directly translates into a small communication amount.

For the NP-hard graph partitioning problem various approximation algorithms exist, e.g. based on breadth-first search or multilevel algorithms.



Dynamic Mapping

Schemes for dynamic mapping are:

Master-Slave:	A master process assignes work to slave processes (see below),
Online Scheduling:	Idle processes take tasks from a central work pool (same as List scheduling),
Rebalancing:	neighboured processes exchange tasks to rebalance work load (mesh refinement)

Parallel Algorithm Models

Parallel algorithms typically have a structure, which falls into one of the following categories:

- Data-Parallel Model,
- Task-Graph Model,
- Work Pool Model,
- Mast-Slave Model or
- Pipeline Model

These algorithm structures often also induce data decomposition and processor mapping.

Data-Parallel Model

In the *Data-Parallel Model* tasks perform

- similar or equal operations on
- different data.

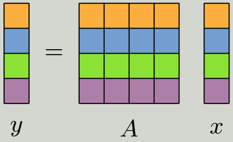
The tasks are typically statically mapped.

Often the algorithm can be split into several steps of data-parallel work.

Example: Dense Matrix-Vector Multiplication

Computing the product $y = Ax$ using a row-wise decomposition, each task is equal while working on a different row of A.

```
void  mul_vec ( int n, double * A, double * x, double * y ) {
  int  my_row = get_row();
  for ( int j = 0; j < n; ++j )
    y[my_row] += A[my_row*n * i] * x[j];
}
```



Task-Graph Model

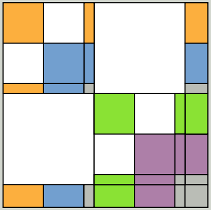
In the *Task-Graph Model* the task dependency graph directly defines the structure of the algorithm.

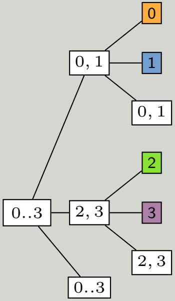
Often, the tasks work on large data sets, making task relocating expensive. Processor mapping is therefore usually static.

Example: Sparse LU factorisation

For sparse matrices, *nested dissection* can be used to decompose the data for an efficient parallel LU factorisation. In nested dissection, sub blocks of a sparse matrix are recursively *decoupled* by an *internal border*, such that, after reordering, large zero blocks result in the matrix.

```
void LU_nd ( BlockMatrix & A ) {
  int  id = my_id();

  // compute local part
  LU_nd( A( id, id ) );
  solve_L( A( id, id ), A( id, 2 ) );
  solve_U( A( id, id ), A( 2, id ) );
  update( A( 2, id ), A( id, 2 ),
          A( 2, 2 ) );
  // factorise global part
  if ( id == local_master() )
    LU( A( 2, 2 ) );
}
```





Work Pool Model

If the tasks can be cheaply relocated, e.g. involve little data, the *Work Pool Model* may be used. Here, the set of tasks is dynamically mapped onto the processors, e.g. the first free processor executes the first, not yet computed task.

Example: Chunk based Loop Parallelisation

If the cost per loop entry differs, the loop is split into *chunks* of a fixed size, which are then dynamically mapped to the processors:

```
// function per index with unknown cost
double  f ( int i );

// computation per task
void  compute_chunk ( int start, int chunk_size, double * array ) {
  for ( int i = start; i < start + chunk_size; ++i )
    array[i] = f( i );
}
// parallel computation of for loop
void  parallel_for ( int  n, int chunk_size, double * array ) {
  for ( int i = 0; i < n; i += chunk_size )
    spawn_task( compute_chunk( i, chunk_size, array );
}
```

The chunk size depends on the size of the array and the number of processors. It may even be adapted during the computation, e.g. if the costs per chunk are

- too low (too much management overhead) or
- too high (bad load balancing).

Master-Slave Model

The *Master-Slave Model* is similar to the work pool model but here, a dedicated *master process* creates and assigns tasks to several *slave processes*.

Remark

In the work pool model, any process may create (spawn) new tasks!

Tasks may be created and mapped

a priori:	if the task set and task sizes are known or
dynamically:	if new tasks are created due to the result of previous task computations.

The costs for managing the tasks at the master process may become a bottleneck. Alternatively, the model may be applied recursively, e.g. with several second-level masters managing a sub set of the tasks and slaves.

Example: Search Farm

A central server receives simultaneous *search requests* and assignes each request to a special slave, actually computing the search result.

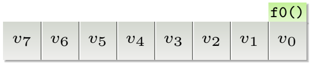
Pipeline Model

If the computation consists of several steps, where different computations are performed successively on each item of an input data stream, the computations may be executed in a *pipeline*.

The pipeline may work parallel w.r.t. the different computations or the input stream (or both).

Example

For an input vector $v \in \mathbb{R}^n$ compute $f_3(f_2(f_1(f_0(v))))$.



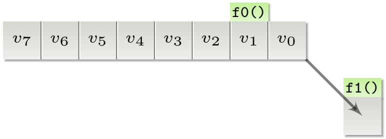
Pipeline Model

If the computation consists of several steps, where different computations are performed successively on each item of an input data stream, the computations may be executed in a *pipeline*.

The pipeline may work parallel w.r.t. the different computations or the input stream (or both).

Example

For an input vector $v \in \mathbb{R}^n$ compute $f_3(f_2(f_1(f_0(v))))$.



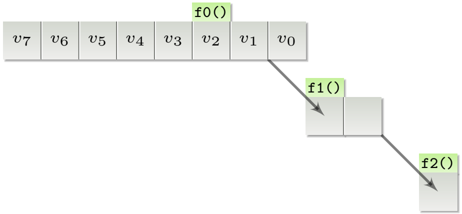
Pipeline Model

If the computation consists of several steps, where different computations are performed successively on each item of an input data stream, the computations may be executed in a *pipeline*.

The pipeline may work parallel w.r.t. the different computations or the input stream (or both).

Example

For an input vector $v \in \mathbb{R}^n$ compute $f_3(f_2(f_1(f_0(v))))$.



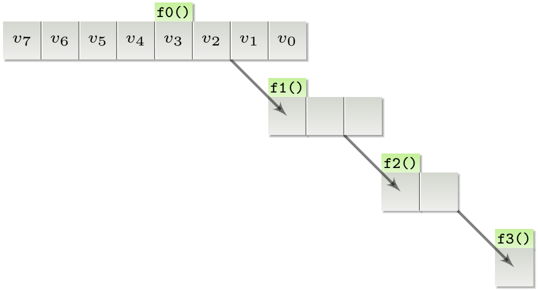
Pipeline Model

If the computation consists of several steps, where different computations are performed successively on each item of an input data stream, the computations may be executed in a *pipeline*.

The pipeline may work parallel w.r.t. the different computations or the input stream (or both).

Example

For an input vector $v \in \mathbb{R}^n$ compute $f_3(f_2(f_1(f_0(v))))$.



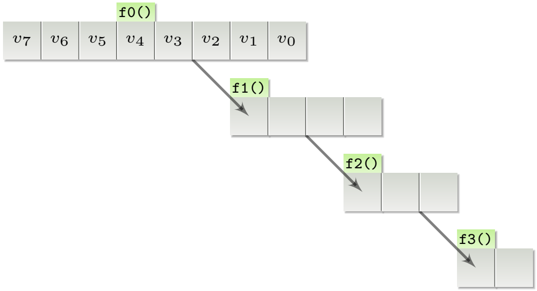
Pipeline Model

If the computation consists of several steps, where different computations are performed successively on each item of an input data stream, the computations may be executed in a *pipeline*.

The pipeline may work parallel w.r.t. the different computations or the input stream (or both).

Example

For an input vector $v \in \mathbb{R}^n$ compute $f_3(f_2(f_1(f_0(v))))$.



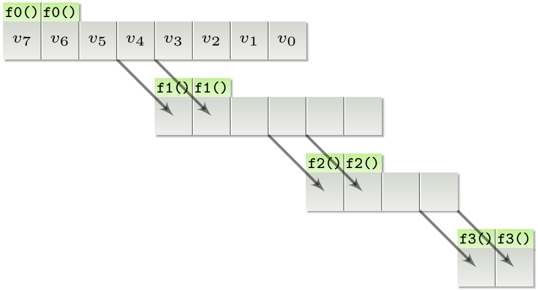
Pipeline Model

If the computation consists of several steps, where different computations are performed successively on each item of an input data stream, the computations may be executed in a *pipeline*.

The pipeline may work parallel w.r.t. the different computations or the input stream (or both).

Example

For an input vector $v \in \mathbb{R}^n$ compute $f_3(f_2(f_1(f_0(v))))$.



Performance Metrics and Complexity

Speedup

As the *runtime* of an algorithm is usually the most interesting measure, the following will focus on it. Another measure may be the memory consumption.

> *Let $t(p)$ be the runtime of an algorithm on p processors of a parallel system. If $p = 1$ then $t(1)$ will denote the runtime of the sequential algorithm.*

The most known and used performance measure of a parallel algorithm is the parallel *Speedup*:

$$S(p) := \frac{t(1)}{t(p)}$$

An *optimal* speedup is achieved, if $t(p) = t(1)/p$ and hence

$$S(p) = p$$

In most cases however, some form of *overhead* exists in the parallel algorithm, e.g. due to sub-optimal load balancing, which prevents an optimal speedup. This overhead $t_o(p)$ is given by

$$t_o(p) = pt(p) - t(s)$$

Amdahl's Law

Most parallel algorithm also contain *some sequential part*, i.e. where not all processors may be used.

Let $0 \leq c_s \leq 1$ denote this sequential fraction of the computation. Assuming the same algorithm for all p, one gets

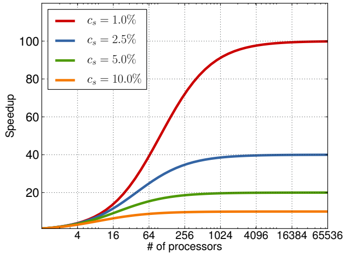
$$t(p) = c_s t(1) + \frac{(1-c_s)}{p} t(1)$$

This leads to

$$S(p) = \frac{1}{c_s + \frac{1-c_s}{p}}$$

which is also known as *Amdahl's Law* (see Amdahl 1967) and severely limits the maximal speedup by the sequential part of the parallel algorithm:

$$\lim_{p \to \infty} S(p) = \frac{1}{c_s}$$



Gustafson's Law

In Amdahl's Law the problem size is fixed, and only p is increased. In practise however, more computing resources usually lead to larger problems computed.

Assume that each processor handles a constant sized (sub-) problem, which costs time T_c to compute and let T_s denote the time for the remaining sequential part. Then we have

$$t(p) = T_s + T_c$$

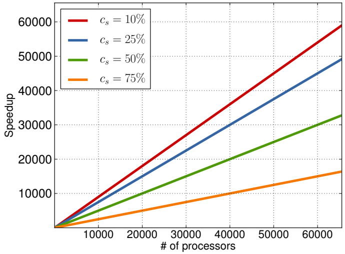
and a sequential runtime for the *same* problem of

$$t(1) = T_s + pT_c$$

With $c_s = T_s/(T_s + T_c)$ the speedup is given by

$$\begin{aligned} S(p) &= c_s + p(1 - c_s) \\ &= p - c_s(p - 1) \end{aligned}$$

which is known as *Gustafson's Law* (see Gustafson 1988).



Example: Parallel Sum

Summing up n numbers sequentially takes $\mathcal{O}(n)$ time.

```
double sum1 ( int i1, int i2, double * x ) {
  if ( i2 - i1 == 1 )
    return x[i1];
  else {
    double s1 = spawn_task( sum( i1, (i1+i2)/2, x ) );
    double s2 = spawn_task( sum( (i1+i2)/2, i2, x ) );

    return s1+s2;
  }
}
```

The parallel algorithm can use p processors at the lower $\log n - \log p$ levels, resulting in $\mathcal{O}(n/p)$ runtime. At the first $\log p$ levels $1, 2, 4, \ldots, p$ processors can be utilised. Hence, the total runtime is

$$t(p) = \mathcal{O}(n/p + \log p)$$

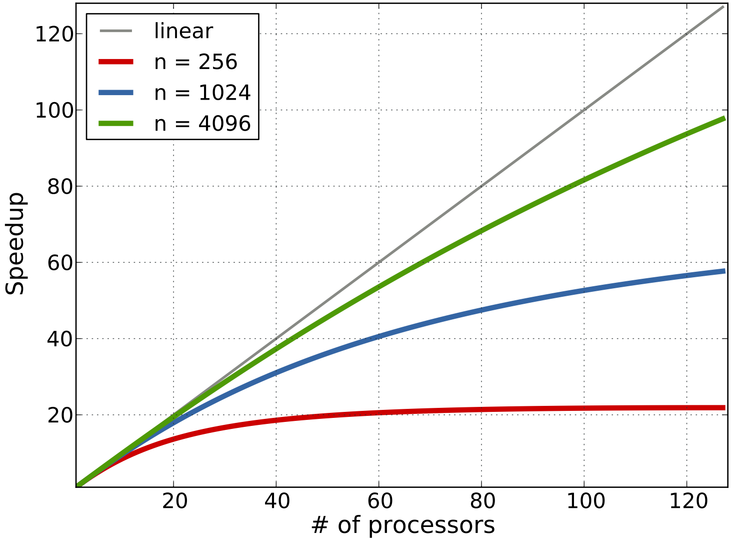
$\mathcal{O}(\log p)$ is the "sequential" part of the algorithm but if $n/p \geq \log p$ the runtime is dominated by the parallel part: $t(p) = \mathcal{O}(n/p)$. This yields an optimal speedup:

$$S(p) = \mathcal{O}\left(\frac{n}{n/p}\right) = \mathcal{O}(p)$$

Finally, for $p = n/2$ the maximal speedup of $\mathcal{O}(n/\log n)$ is achieved.

Example: Parallel Sum

Speedup of the parallel sum algorithm:



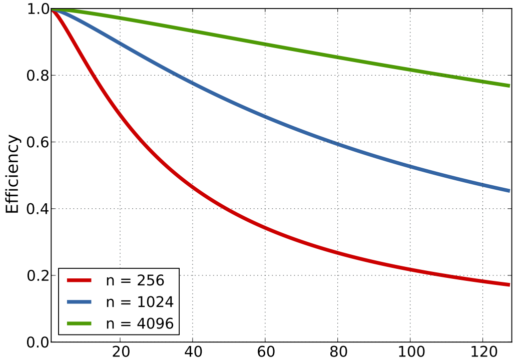
Efficiency

Tightly coupled with speedup is *Efficiency*:

The parallel efficiency $E(p)$ *of an algorithm is defined as*

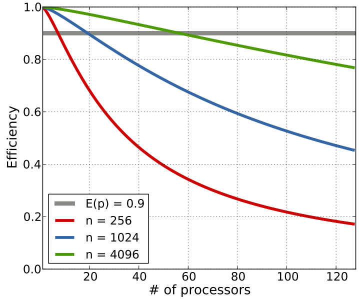
$$E(p) = \frac{S(p)}{p} = \frac{t(1)}{pt(p)} = \frac{1}{1 + \frac{t_o(p)}{t(1)}}$$

Efficiency of the parallel sum algorithm:



Scalability

Taking a closer look at the parallel efficiency of the (distributed) parallel sum:



n	$p = 8$	$p = 20$	$p = 60$
256	0.88	0.71	0.35
1024	0.97	0.90	0.68
4096	0.99	0.98	0.90

Although, for a *fixed* n the efficiency drops with an increasing p, the same level of efficiency can be maintained if p and n are increased *simultaneously*. Such algorithms are called *scalable*.

Remark

The scalability of an algorithm strongly depends on the underlying parallel computer system. Therefore, in principle both, the algorithm and the computer hardware, have to be considered.

Complexity Analysis

For the theoretical analysis of sequential programs, the *Random Access Machine* (RAM, see Cook and Reckhow 1973) model is used. It is based on

- an infinite memory with $\mathcal{O}(1)$ access time and
- a finite instruction set.

Practically all sequential computers are (finite) realisations of a RAM.

Using the RAM model, the complexity analysis of a program can be done independent on an actual computer system, e.g. the loop

```
for ( int  i = 0; i < n; ++i )
   a[i] := a[i] + 1;
```

has runtime $\mathcal{O}(n)$ on all sequential systems.

Complexity Analysis

For the theoretical analysis of sequential programs, the *Random Access Machine* (RAM, see Cook and Reckhow 1973) model is used. It is based on

- an infinite memory with $\mathcal{O}(1)$ access time and
- a finite instruction set.

Practically all sequential computers are (finite) realisations of a RAM.

Using the RAM model, the complexity analysis of a program can be done independent on an actual computer system, e.g. the loop

```
for ( int  i = 0; i < n; ++i )
   a[i] := a[i] + 1;
```

has runtime $\mathcal{O}(n)$ on all sequential systems.

No such universal model exists for parallel computers!

Hence, the complexity analysis of parallel programs can only be done for a specific computer, e.g. the runtime of the above loop may vary from $\mathcal{O}(n/p)$ to $\mathcal{O}(n \cdot p)$, depending on interconnect network.

PRAM

But there exist models, which *approximate* actual computer systems and allow hardware-independent analysis for programs.

For shared memory systems, the generalisation of the RAM is the *Parallel RAM* (PRAM, see Fortune and Wyllie 1978), with

- an infinite memory with $\mathcal{O}(1)$ access time for *all* processors and
- a finite instruction set.

Using the PRAM model, the loop

```
for ( int  i = 0; i < n; ++i )
   a[i] := a[i] + 1;
```

has a runtime of $\mathcal{O}(n/p)$ if each of the p processors handle n/p entries of the array.

Remark

UMA systems are optimal realisations of a PRAM, whereas NUMA systems are only approximations.

BSP Machine

For shared and distributed memory machines, a simplified model is the *Bulk Synchronous Parallel Machine* (BSP Machine, see Valiant 1990), which is defined by two parameters:

g : transfer cost for a single word through the network and
l : cost for a global synchronisation.

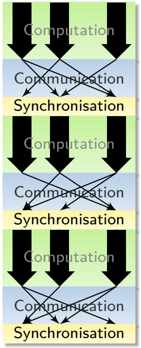
A BSP computation consists of several *supersteps*, each having a

computation phase:	compute on local data only,
communication phase:	send and receive data,
synchronisation:	synchronise global machine

The complexity of a single superstep has the form

$$w + h \cdot g + l,$$

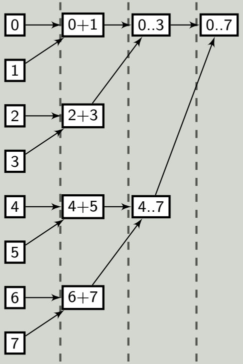
where w is the max. number of operations and h the max. data size sent/received by each processor. The total work is simply the sum over all supersteps.



BSP Machine

Example: Distributed Parallel Sum

```
double  par_sum ( int n_loc, double * x_loc ) {
  int      p     = bsp_nprocs(); // total proc. number
  int      id    = bsp_id();     // local proc. id
  double   s_loc = 0.0;

  // sum of local data
  for ( int i = 0; i < n_loc; ++i ) s_loc += x_loc[i];

  // sum up local data in log_2(p) steps
  for ( int i = 1; i < p; i *= 2 ) {
    // send local sum to master
    if ( id % (2*i) != 0 ) bsp_send( id-i, s_loc );

    // synchronise (wait for communication)
    sync();

    // get remote data and update local sum
    if ( bsp_nmsgs() > 0 ) s_loc +=  bsp_get();
  }
}
```



It the first step the local sum is computed in time $\mathcal{O}(n/p)$. In the following $\log p$ steps, one number is send/received and the local sum updated. This yiels a total complexity of

$$\mathcal{O}\left(\underbrace{n/p + \log p}_{\text{computation}} + \underbrace{g \cdot \log p}_{\text{communication}} + \underbrace{l \cdot \log p}_{\text{synchronisation}}\right)$$

General Message Passing

The BSP model is best suited for algorithms which can be decomposed into *global* single steps.

Together with corresponding software libraries, it can tremendously simplify parallel programming.

But idling may easily appear, e.g.

- if computations and data exchange only affect a local part in a superstep or
- if load is not optimally balanced.

Therefore, a more general model is considered, in which computations and communication may be fully decoupled from all processors.

General Message Passing

Again, two parameters will define the parallel machine, but now they describe a *single* communication:

t_s: startup time for a data transfer (latency),

t_w: time to transfer a single data word (inverse bandwidth)

Sending m words costs

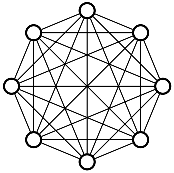
$$t_s + m \cdot t_w$$

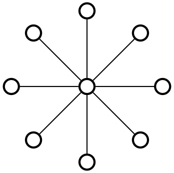
In the BSP model all characteristics of the network, especially the topology, are compressed into two parameters.

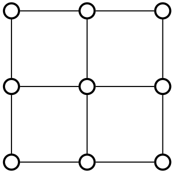
For the general message passing model such network properties have to be considered for analysing algorithm complexity.

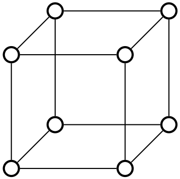
General Message Passing

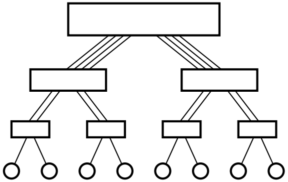
As an example, we consider the *one-to-all broadcast*.


fully connected
$t_s + t_w$


Star
$2(t_s + t_w)$


2D-Mesh
$2\sqrt{p}(t_s + t_w)$

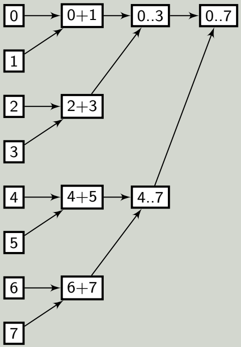

d-dim Hypercube
$d(t_s + t_w) = \log_2 p$


Fat Tree
$2\log_2(p)(t_s + t_w)$

General Message Passing

Example: Distributed Parallel Sum

```
double  par_sum ( int n_loc, double * x_loc ) {
  int      p     = get_nprocs(); // total proc. number
  int      id    = get_id();     // local proc. id
  double   s_loc = 0.0;

  // sum of local data
  for ( int i = 0; i < n_loc; ++i ) s_loc += x_loc[i];

  // sum up local data in log_2(p) steps
  for ( int i = 1; i < p; i *= 2 ) {
    if ( id % (2*i) != 0 ) {
      // send local sum to master and finish
      send( id-i, s_loc );
      break;
    }
    else {
      // recieve remote sum and update local data
      double  s_rem = 0;

      recv( id+i, s_rem );
      s_loc += s_rem;
    }
  }
}
```



The complexity is again

$$\mathcal{O}\left(n/p + \log p + \log p(t_s + t_w)\right)$$

Literature

A. Grama et al. (2003). *Introduction to Parallel Computing, Second Edition*. Pearson Education Limited.

Amdahl, G. (1967). "*Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities*". In: *AFIPS Conference Proceedings*, pp. 483–485.

Coffman, E.G., M.R. Garey, and D.S. Johnson (1978). "*An application of bin-packing to multiprocessor scheduling*". In: *SIAM J. Comput.* 7.1, pp. 1–17.

Cook, S. and R. Reckhow (1973). "*Time Bounded Random Access Machines*". In: *Journal of Computer and Systems Sciences* 7, pp. 354–375.

Fortune, S. and J. Wyllie (1978). "*Parallelism in random access machines*". In: *Proceedings of the tenth annual ACM symposium on Theory of computing*. San Diego, California, United States: ACM Press, pp. 114–118.

Garey, M.R. and D.S. Johnson (1979). *Computers and Intractability, A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company.

Gustafson, John L. (1988). "*Reevaluating Amdahl's Law*". In: *Communications of the ACM* 31, pp. 532–533.

Sutter, Herb (2005). "*The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software*". In: *Dr. Dobb's Journal*.

Valiant, L.G. (1990). "*A bridging model for parallel computation*". In: *Communications of the ACM* 33.8, pp. 103–111. ISSN: 0001-0782. DOI: `http://doi.acm.org/10.1145/79173.79181`.

W.R. Stevens (1998). *UNIX Network Programming, Volume 2: Interprocess Communications, Second Edition*. Prentice Hall.