Advanced Topics in Numerical Analysis: High Performance Computing

Cross-listed as MATH-GA.2012-001 and CSCI-GA 2945.001

Benjamin Peherstorfer
Courant Institute, NYU
pehersto@cims.nyu.edu

Spring 2020, Monday, 1:25–3:15PM, WWH #512

February 24, 2020

Slightly adapted from Georg Stadler's lectures.

Today

Last lecture

- Single core performance
- Vectorization and pipelining

Today

- More on data layout
- Performance models work depth
- Parallel programming models
- Performance models Amdahl's law
- Shared memory parallelization and threads
- Tool: git

Announcements and upcoming

- New homework posted (due March 9, 2020)

Feedback

What you like

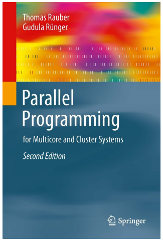
- General impression is that class is going OK
- Short programs and demos are interesting/instructive

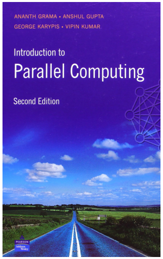
Suggestions

- Reading material
- Large applications/more complex examples that use HPC
- Cheat sheets
  - OpenMP: `https://www.openmp.org/wp-content/uploads/OpenMPRef-5.0-111802-web.pdf`
  - MPI: `http://www.netlib.org/utk/people/JackDongarra/WEB-PAGES/SPRING-2006/mpi-quick-ref.pdf`

Feedback: Literature

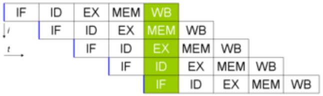
- T. Rauber and G. Rünger: *Parallel Programming for Multicore and Cluster Systems*, Springer, 2nd edition 2013. *Available online on NYU Campus.*
- A. Grama, A. Gupta, G. Karypis and V. Kumar: *Introduction to Parallel Computing*, Pearson, 2003.
- To refresh C programming language: Z. Shaw: *Learn C the hard way*, 2016.





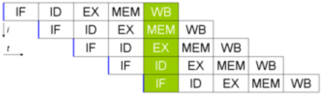
Levels of parallelism

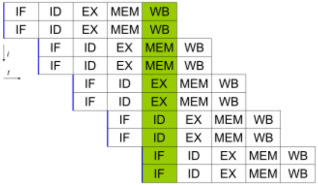
- Parallelism at the bit level (64-bit operations)



Levels of parallelism

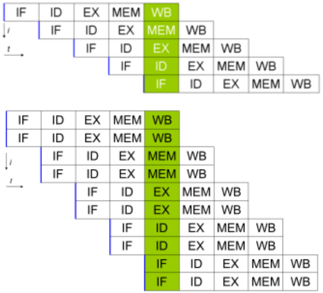
- Parallelism at the bit level (64-bit operations)
- Parallelism by pipelining (overlapping of execution of multiple instructions); "assembly line" parallelism, Instruction-Level-Parallelism (ILP); several operators per cycle





Levels of parallelism

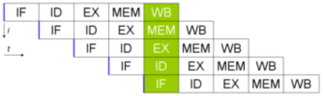
- Parallelism at the bit level (64-bit operations)
- Parallelism by pipelining (overlapping of execution of multiple instructions); "assembly line" parallelism, Instruction-Level-Parallelism (ILP); several operators per cycle
- multiple functional units parallelism: ALUs (algorithmic logical units), FPUs (floating point units), load/store memory units,...

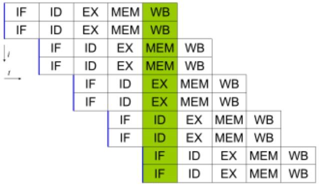


all of the above assume single sequential control flow

Levels of parallelism

- Parallelism at the bit level (64-bit operations)
- Parallelism by pipelining (overlapping of execution of multiple instructions); "assembly line" parallelism, Instruction-Level-Parallelism (ILP); several operators per cycle
- multiple functional units parallelism: ALUs (algorithmic logical units), FPUs (floating point units), load/store memory units,...





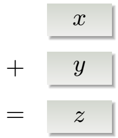
all of the above assume single sequential control flow

- process/thread level parallelism: independent processor cores, multicore processors; parallel control flow

Vectorization

Standard processing mode of a processor is *scalar*:

$$z := \mathrm{op}_1(x) \quad \text{or} \quad z := \mathrm{op}_2(x, y) \quad \text{or} \quad \cdots$$



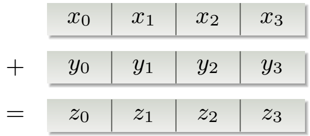
with $\mathrm{op}_i : \mathbb{R}^i \to \mathbb{R}$ and costs

$$t_{\mathrm{load}} + t_{\mathrm{op}}$$

Here, t_{load} denotes the time to load the data from memory.

With *vectorization* we have $\mathrm{op}_i : \mathbb{R}^{i \cdot n} \to \mathbb{R}^n$, e.g.

$$\begin{pmatrix} z_1 \\ z_2 \\ \vdots \\ z_n \end{pmatrix} := \mathrm{op} \begin{pmatrix} x_1 & y_1 \\ x_2 & y_2 \\ \vdots & , \ \vdots \\ x_n & y_n \end{pmatrix}$$



with costs

$$n \cdot t_{\mathrm{load}} + 1 \cdot t_{\mathrm{op}}$$

vectorization is a realisation of an SIMD parallel architecture with algorithms employing the data parallel model.

Vectorization

Let $x, y, z \in \mathbb{K}^n$ with n *large*. Compute

$$z_i := e^{z_i + x_i \cdot y_i}, \quad \forall i \leq n$$

Standard implementation

```
for ( size_t  i = 0; i < n; ++i )
  z[i] = std::exp( z[i] + x[i]*y[i] );
```

Auto vectorization with Intel compiler (similar statments for gcc)

```
#pragma simd
for ( size_t  i = 0; i < n; ++i )
  z[i] = std::exp( z[i] + x[i]*y[i] );
```

Speedup: 2.79x ($\mathbb{K} = \mathbb{R}$, Xeon E5-2640), 5.01x ($\mathbb{K} = \mathbb{C}$, XeonPhi 5110P)

Manual vectorization

```
for ( size_t  i = 0; i < n; i += 4 ) {
  const __m256d  vx = _mm256_load_pd( & x[i] );
  const __m256d  vy = _mm256_load_pd( & y[i] );
  __m256d        vz = _mm256_load_pd( & z[i] );

  vz = _mm256_exp_pd( _mm256_add_pd( vz, _mm256_mul_pd( vx, vy ) ) );
  _mm256_store_pd( & z[i], vz );
}
```

Speedup: 3.20x ($\mathbb{K} = \mathbb{R}$, Xeon E5-2640), 6.93x ($\mathbb{K} = \mathbb{C}$, XeonPhi 5110P)

Auto vectorization

Most C/C++ compilers will *automatically* use vector instructions for handling suitable data, e.g. the loop

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

will be automatically converted into

```
for ( int  i = 0; i < n; ++i )
   z[i:i+3] = z[i:i+3] + x[i:i+3]*y[i:i+3];
```

on a vector CPU with four entries per register.

To explicitly activate this auto-vectorization, different compiler flags are used:

Intel Compiler

```
> icpc -O2 -msse2 -vec -c f.cc
> icpc -O2 -mavx -vec  -c f.cc
> icpc -O2 -mmic -vec  -c f.cc
```

GNU Compiler

```
> g++ -O2 -ftree-vectorize -msse2 -c f.cc
> g++ -O2 -ftree-vectorize -mavx  -c f.cc
```

For gcc

- Flag "-ftree-vectorize" turns auto-vectorization on
- Flag "-mavx" and "-msse2" tells the compile what is supported
- Flag "-fopt-info" gives information about vectorization (cryptic)
- If "-O3", then automatically vectorizes
- Both compilers will only vectorize for optimisation levels -O2 or higher

Data structure layout

The layout of data structures may have a large impact on the efficiency of vectorization.
Consider:

```
struct  vector_t {
  double    x, y, z;
};

struct  particle_t {
  double     mass;
  vector_t   pos;
};

void f ( int n, particle_t * p, vector_t & t ) {
  for ( int i = 0; i < n; ++i ) {
    p[i].pos.x += t.x;
    p[i].pos.y += t.y;
    p[i].pos.z += t.z;
  }
}
```

The memory layout of an `particle_t` array is

...	mass	x	y	z	mass	x	y	z	mass	x	y	z	mass	x	y	z	...

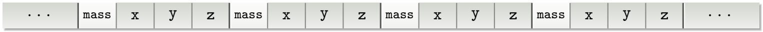
In function `f`, each fourth element (`mass`) is unused during the computations, leading to *gaps* in the memory stream. Hence, only three values can be loaded simultaneously.

Data structure layout

The layout of data structures may have a large impact on the efficiency of vectorization.
Consider:

```
struct  vector_t {
  double    x, y, z;
};

struct  particle_t {
  double    mass;
  vector_t  pos;
};

void f ( int n, particle_t * p, vector_t & t ) {
  for ( int i = 0; i < n; ++i ) {
    p[i].pos.x += t.x;
    p[i].pos.y += t.y;
    p[i].pos.z += t.z;
  }
}
```

The memory layout of an `particle_t` array is

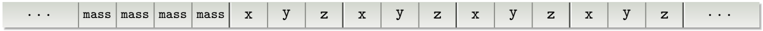


In function `f`, each fourth element (`mass`) is unused during the computations, leading to *gaps* in the memory stream. Hence, only three values can be loaded simultaneously.

The first optimisation approach is to decouple `mass` and `pos` for all particles:

```
struct  vector_t     { double x, y, z; };
struct  particles_t {
  double *     mass; // array of masses
  vector_t *  pos;  // array of positions
};

void f ( int n, particles_t & p, vector_t & t ) {
  for ( int i = 0; i < n; ++i ) {
    p.pos[i].x += t.x;
    p.pos[i].y += t.y;
    p.pos[i].z += t.z;
  }
}
```

This yields the modified data layout



without any gaps in the memory stream of the function `f`. But it is still inefficient for loading data into vector registers for *individual operations*, e.g. for `x` alone.

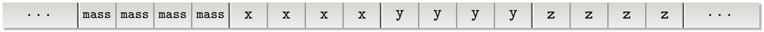
Speedup compared to first algorithm:

SSE2 (Xeon X5650)	AVX (Xeon E5-2640)	MIC (XeonPhi 5110P)
1.58x	1.59x	1.68x

The second approach is to *fully* decouple all data elements:

```
struct  particles_t {
  double *  mass; // array of masses
  double *  x;    // individual arrays for
  double *  y;    // all coordinates
  double *  z;
};

void f ( int n, particles_t & p, vector_t & t ) {
  for ( int i = 0; i < n; ++i ) {
    p.x[i] += t.x;
    p.y[i] += t.y;
    p.z[i] += t.z;
  }
}
```

leading to separate memory blocks for each value type:



Data can now be loaded directly into vector registers for each sub-operation.

Speedup compared to

	SSE2 (Xeon X5650)	AVX (Xeon E5-2640)	MIC (XeonPhi 5110P)
second algorithm:	1.06x	1.18x	2.65x
first algorithm:	1.67x	1.88x	4.45x

Array-of-Structures

The data structures used in the initial algorithm follow the *Array-of-Structures* principle (AOS), which

- is good for computations affecting data within a single item (good data locality),
- but has bad data locality for computations affecting all items, e.g. via vectorization,
- has good code structure, e.g. all data for single item packed together (*Object Oriented* approach),

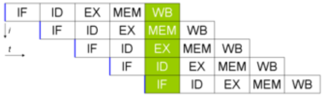
Structure-of-Arrays

The data structures of the last algorithm follow the *Structure-of-Arrays* principle (SOA), which

- is good for computations affecting all items,
- but has bad data locality for computations affecting data within a single item

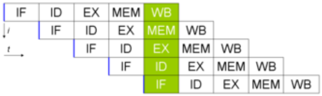
Levels of parallelism

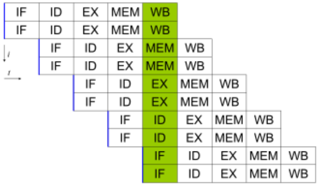
- Parallelism at the bit level (64-bit operations)



Levels of parallelism

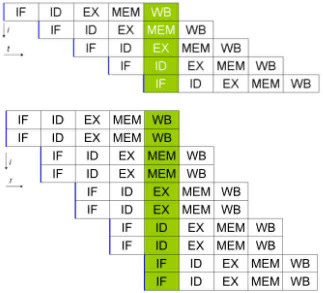
- Parallelism at the bit level (64-bit operations)
- Parallelism by pipelining (overlapping of execution of multiple instructions); "assembly line" parallelism, Instruction-Level-Parallelism (ILP); several operators per cycle





Levels of parallelism

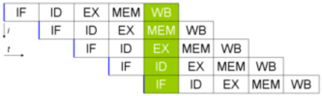
- Parallelism at the bit level (64-bit operations)
- Parallelism by pipelining (overlapping of execution of multiple instructions); "assembly line" parallelism, Instruction-Level-Parallelism (ILP); several operators per cycle
- multiple functional units parallelism: ALUs (algorithmic logical units), FPUs (floating point units), load/store memory units,. . .

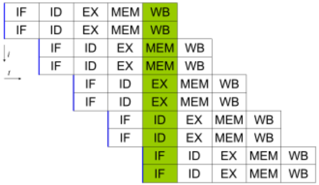


all of the above assume single sequential control flow

Levels of parallelism

- Parallelism at the bit level (64-bit operations)
- Parallelism by pipelining (overlapping of execution of multiple instructions); "assembly line" parallelism, Instruction-Level-Parallelism (ILP); several operators per cycle
- multiple functional units parallelism: ALUs (algorithmic logical units), FPUs (floating point units), load/store memory units,...





all of the above assume single sequential control flow

- process/thread level parallelism: independent processor cores, multicore processors; parallel control flow

Work Depth Model

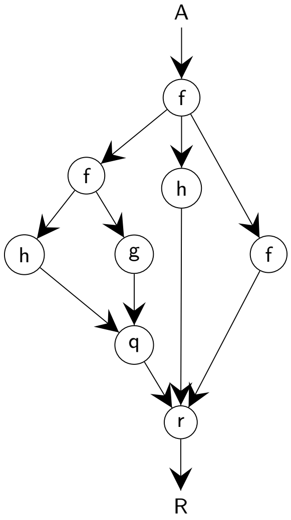
Is this parallel?

```
B=f(A);
C=f(B);
D=h(B);
G=h(C);
F=g(C);
E=f(B);
H=q(G, F);
R=r(H,D,E);
```

Work Depth Model

Is this parallel?

B=f(A);
C=f(B);   D=h(B);
G=h(C);   F=g(C);   E=f(B);
H=q(G, F);
R=r(H,D,E);
How about now?



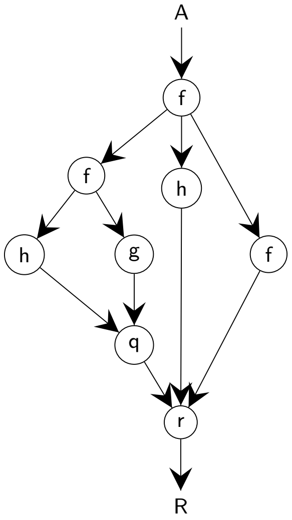
Work Depth Model

Is this parallel?

B=f(A);
C=f(B); D=h(B);
G=h(C); F=g(C); E=f(B);
H=q(G, F);
R=r(H,D,E);
How about now?

Directed Acyclic Graph (DAG)

Work = #-of-nodes
Depth = #-of-levels
Parallelism = Work/Depth
Connected: Otherwise have isolated notes with useless computation
acyclic: Otherwise infinite loops

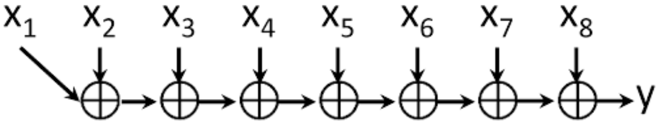


Work Depth Model

Reduction: y $= x_0 + x_1 + x_2 + \cdots + x_n \rightarrow$ board

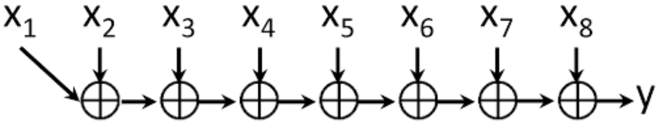
Work Depth Model

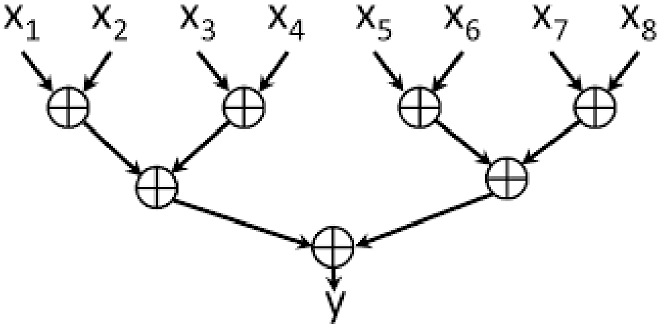
Reduction: y $= x_0 + x_1 + x_2 + \cdots + x_n \rightarrow$ board
work $= \mathcal{O}(n)$
depth $= \mathcal{O}(n)$ or $\mathcal{O}(\log n)$



Work Depth Model

Reduction: y $= x_0 + x_1 + x_2 + \cdots + x_n \rightarrow$ board
work $= \mathcal{O}(n)$
depth $= \mathcal{O}(n)$ or $\mathcal{O}(\log n)$





Work Depth Model

Reduction: y $= x_0 + x_1 + x_2 + \cdots + x_n \rightarrow$ board

Matrix-Matrix Multiplication with parfor: $\rightarrow$ **board**

Work Depth Model

Reduction: y $= x_0 + x_1 + x_2 + \cdots + x_n \rightarrow$ board

Matrix-Matrix Multiplication with parfor: $\rightarrow$ **board**
work $= \mathcal{O}(n^3)$
depth $= \mathcal{O}(n)$
parallelism $= \mathcal{O}(n^2)$

Work Depth Model

Reduction: y $= x_0 + x_1 + x_2 + \cdots + x_n \rightarrow$ board

Matrix-Matrix Multiplication with parfor: $\rightarrow$ board
work $= \mathcal{O}(n^3)$
depth $= \mathcal{O}(n)$
parallelism $= \mathcal{O}(n^2)$

Matrix-Matrix Multiplication:
work $= \mathcal{O}(n^3)$
depth $= \mathcal{O}(\log n)$
parallelism $= \mathcal{O}(n^3/\log n)$

Work Depth Model

Reduction: y $= x_0 + x_1 + x_2 + \cdots + x_n \rightarrow$ board

Matrix-Matrix Multiplication with parfor: $\rightarrow$ **board**
work $= \mathcal{O}(n^3)$
depth $= \mathcal{O}(n)$
parallelism $= \mathcal{O}(n^2)$

Matrix-Matrix Multiplication:
work $= \mathcal{O}(n^3)$
depth $= \mathcal{O}(\log n)$
parallelism $= \mathcal{O}(n^3/\log n)$

Sorting:
work $= \mathcal{O}(n \log n)$
depth $= \mathcal{O}(\log^2 n)$
parallelism $= \mathcal{O}(n/\log n)$

Parallel architectures (Flynn's taxonomy)

Characterization of architectures according to Flynn:

SISD:	Single instruction, single data. This is the conventional sequential model.
SIMD:	Single instruction, multiple data. Multiple processing units with identical instructions, each one working on different data. Useful when a lot of completely identical tasks are needed.
MIMD:	Multiple instructions, multiple data. Multiple processing units with separate (but often similar) instructions and data/memory access (shared or distributed). We will mainly use this approach.
MISD:	Multiple instructions, single data. Not practical.

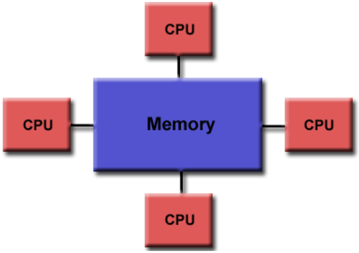
Programming model must reflect architecture

Example: Inner product between two (very long) vectors: $a^T b$:

- Where are a, b stored? Single memory or distributed?
- What work should be done by which processor?
- How do they coordinate their result?

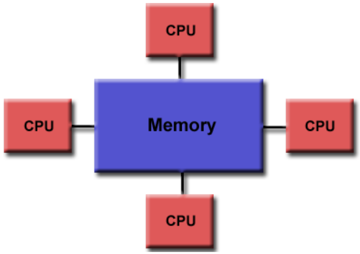
Shared memory programming model

- Program is a collection of control threads, that are created dynamically
- Each thread has private and shared variables
- Threads can exchange data by reading/writing shared variables
- Danger: more than 1 processor core reads/writes to a memory location: race condition



Shared memory programming model

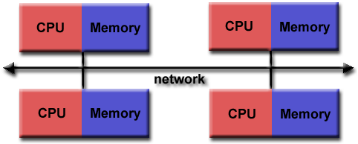
- Program is a collection of control threads, that are created dynamically
- Each thread has private and shared variables
- Threads can exchange data by reading/writing shared variables
- Danger: more than 1 processor core reads/writes to a memory location: race condition



Programming model must manage different threads and avoid race conditions.
OpenMP: Open Multi-Processing is the application interface (API) that supports shared memory parallelism: `www.openmp.org`

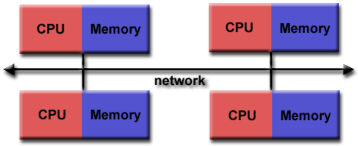
Distributed memory programming model

- Program is run by a collection of named processes; fixed at start-up
- Local address space; no shared data
- logically shared data is distributed (e.g., every processor only has direct access to a chunk of rows of a matrix)
- Explicit communication through send/receive pairs



Distributed memory programming model

- Program is run by a collection of named processes; fixed at start-up
- Local address space; no shared data
- logically shared data is distributed (e.g., every processor only has direct access to a chunk of rows of a matrix)
- Explicit communication through send/receive pairs



Programming model must accommodate communication.

MPI: Massage Passing Interface (different implementations: LAM, Open-MPI, Mpich, Mvapich), http://www.mpi-forum.org/

Hybrid distributed/shared programming model

- Pure MPI approach splits the memory of a multicore processor into independent memory pieces, and uses MPI to exchange information between them.
- Hybrid approach uses MPI across processors, and OpenMP for processor cores that have access to the same memory.
- A similar hybrid approach is also used for hybrid architectures, i.e., computers that contain CPUs and accelerators (GPGPUs, MICs).

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

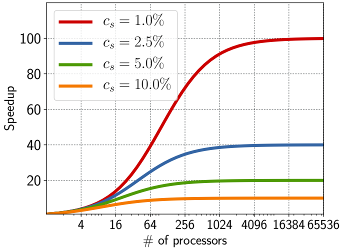
$$t(p) = c_s t(1) + \frac{(1-c_s)}{p} t(1)$$

This leads to

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

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

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



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;
  }
}
```

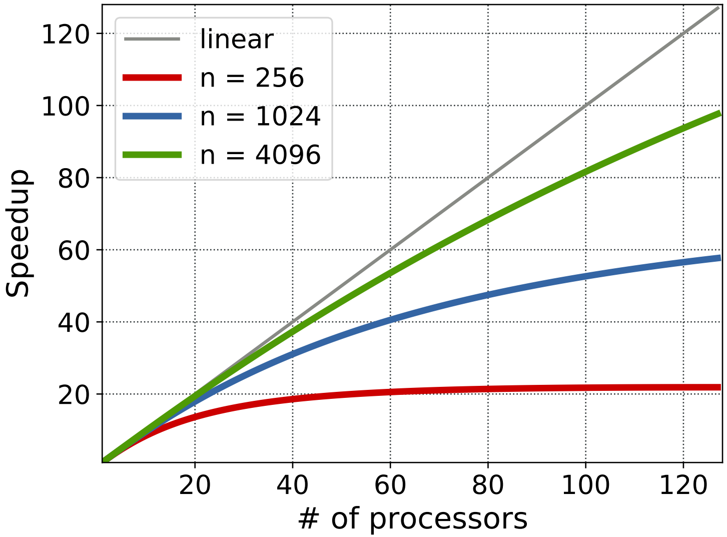
The parallel algorithm can use p processors at the lower $\log n - \log p$ levels, resulting in $\mathcal{O}(n/p)$ runtime. At the first $\log p$ levels $1, 2, 4, \ldots, p$ processors can be utilised. Hence, the total runtime is

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

$\mathcal{O}(\log p)$ 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:

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

Speedup of the parallel sum algorithm:



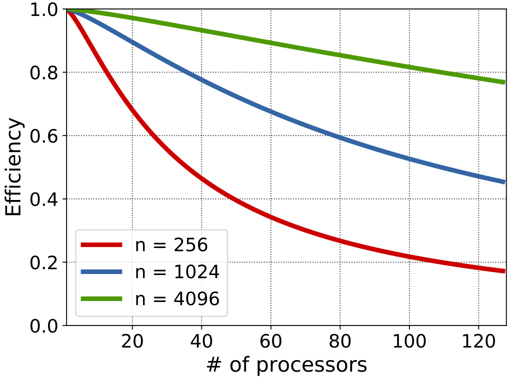
Efficiency

Tightly coupled with speedup is *Efficiency*:

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

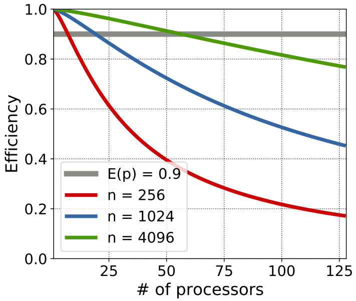
$$E(p) = \frac{S(p)}{p} = \frac{t(1)}{pt(p)}$$

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 *weakly scalable*.

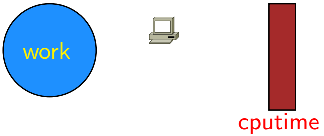
Remark

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.

Parallel scalability

Strong and weak scaling/speedup

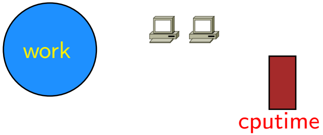
Strong scalability



Parallel scalability

Strong and weak scaling/speedup

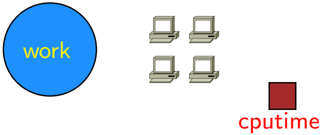
Strong scalability



Parallel scalability

Strong and weak scaling/speedup

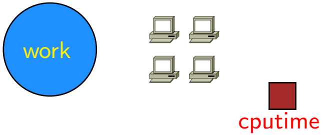
Strong scalability

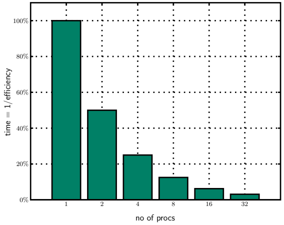


Parallel scalability

Strong and weak scaling/speedup

Strong scalability

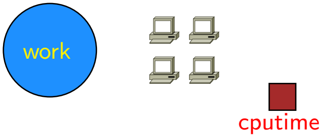


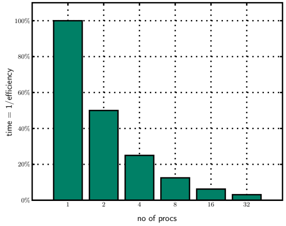


Parallel scalability

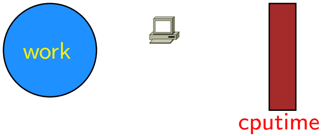
Strong and weak scaling/speedup

Strong scalability





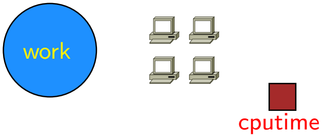
Weak scalability

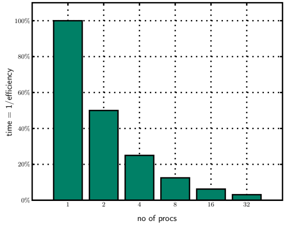


Parallel scalability

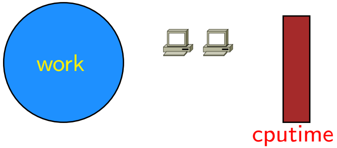
Strong and weak scaling/speedup

Strong scalability





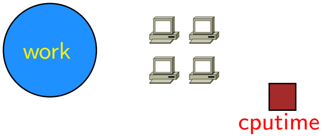
Weak scalability

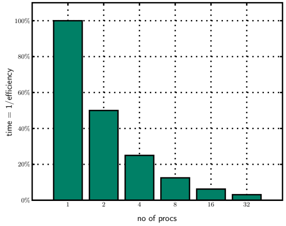


Parallel scalability

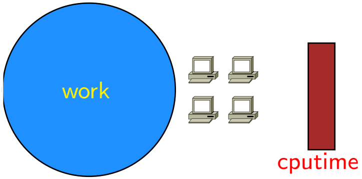
Strong and weak scaling/speedup

Strong scalability





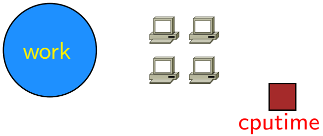
Weak scalability

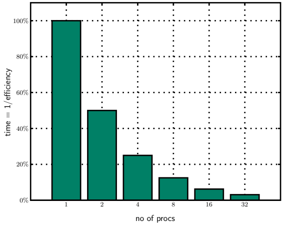


Parallel scalability

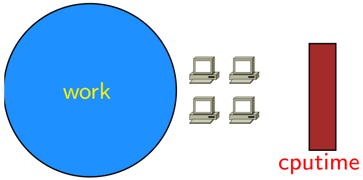
Strong and weak scaling/speedup

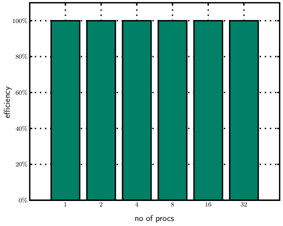
Strong scalability





Weak scalability



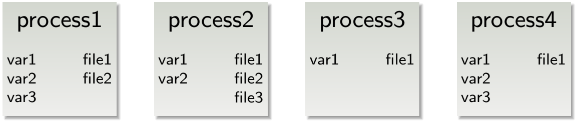


Threads

Threads

On a shared memory architecture, programs are executed as *processes* with their own

- address space,
- file handles,
- ...

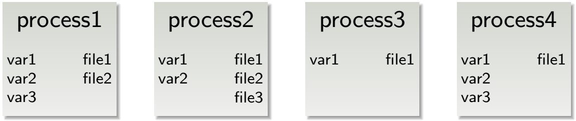


Processes may *not* directly access data in the address space of other processes. Communication between different processes needs extra tools, e.g. via pipes, files, sockets.

Threads

On a shared memory architecture, programs are executed as *processes* with their own

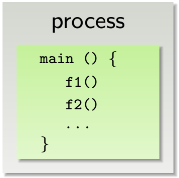
- address space,
- file handles,
- ...



Processes may *not* directly access data in the address space of other processes. Communication between different processes needs extra tools, e.g. via pipes, files, sockets.

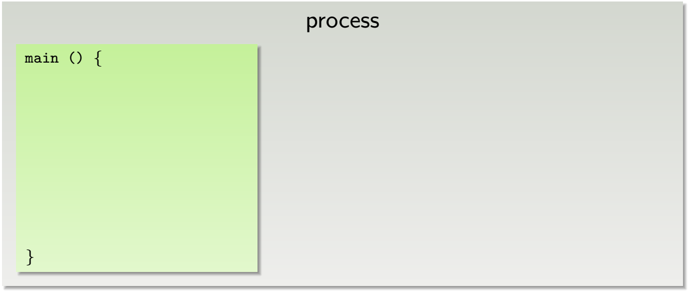
In a process execution starts with the `main` function and afterwards follows the control flow as defined by the programmer.

A normal process will have a single *computation path*.



```
main () {
   f1()
   f2()
   ...
}
```

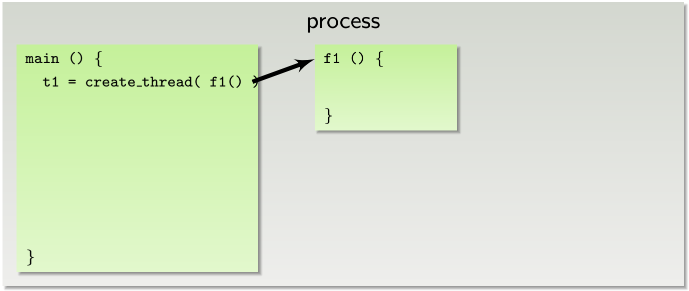
Threads provide a mechanism to have *several* computation paths within a single process.



```
main () {

}
```

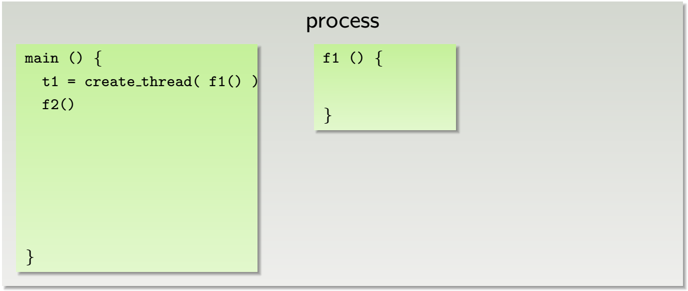
At any time during the runtime of a process, new threads may be spawned. Synchronising with the end of a thread is called *joining*.

Threads provide a mechanism to have *several* computation paths within a single process.



At any time during the runtime of a process, new threads may be spawned. Synchronising with the end of a thread is called *joining*.

Threads provide a mechanism to have *several* computation paths within a single process.

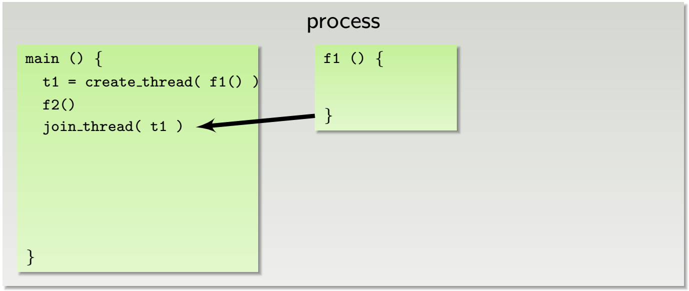


```
main () {
  t1 = create_thread( f1() )
  f2()

}
```

```
f1 () {

}
```

At any time during the runtime of a process, new threads may be spawned. Synchronising with the end of a thread is called *joining*.

Threads provide a mechanism to have *several* computation paths within a single process.

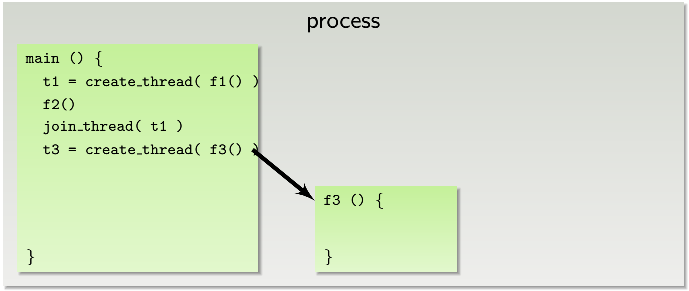


```
main () {
  t1 = create_thread( f1() )
  f2()
  join_thread( t1 )

}
```

```
f1 () {

}
```

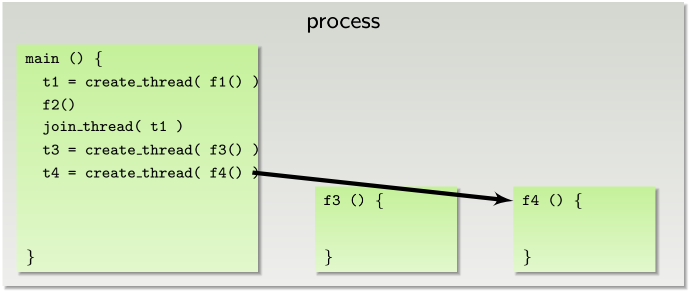
At any time during the runtime of a process, new threads may be spawned. Synchronising with the end of a thread is called *joining*.

Threads provide a mechanism to have *several* computation paths within a single process.



At any time during the runtime of a process, new threads may be spawned. Synchronising with the end of a thread is called *joining*.

Threads provide a mechanism to have *several* computation paths within a single process.



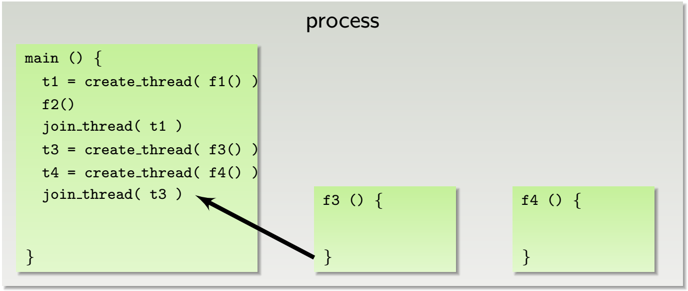
```
main () {
  t1 = create_thread( f1() )
  f2()
  join_thread( t1 )
  t3 = create_thread( f3() )
  t4 = create_thread( f4() )

}
```

```
f3 () {

}
```

```
f4 () {

}
```

At any time during the runtime of a process, new threads may be spawned. Synchronising with the end of a thread is called *joining*.

Threads provide a mechanism to have *several* computation paths within a single process.



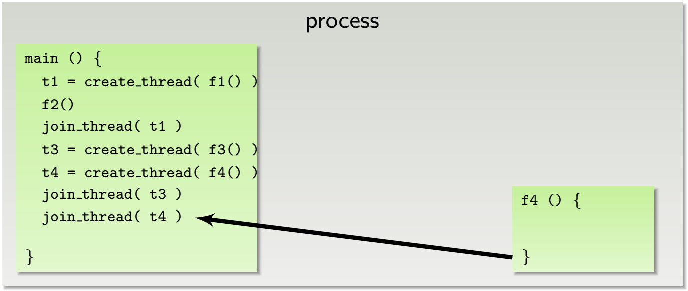
```
main () {
  t1 = create_thread( f1() )
  f2()
  join_thread( t1 )
  t3 = create_thread( f3() )
  t4 = create_thread( f4() )
  join_thread( t3 )


}
```

```
f3 () {

}
```

```
f4 () {

}
```

At any time during the runtime of a process, new threads may be spawned. Synchronising with the end of a thread is called *joining*.

Threads provide a mechanism to have *several* computation paths within a single process.

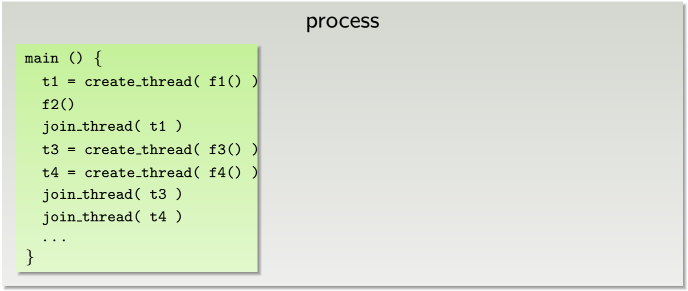


```
main () {
  t1 = create_thread( f1() )
  f2()
  join_thread( t1 )
  t3 = create_thread( f3() )
  t4 = create_thread( f4() )
  join_thread( t3 )
  join_thread( t4 )

}
```

```
f4 () {


}
```

At any time during the runtime of a process, new threads may be spawned. Synchronising with the end of a thread is called *joining*.

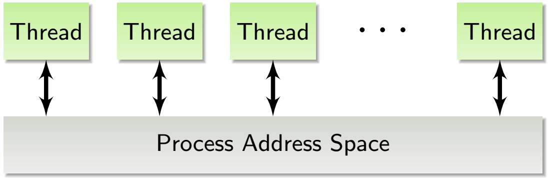
Threads provide a mechanism to have *several* computation paths within a single process.



```
main () {
  t1 = create_thread( f1() )
  f2()
  join_thread( t1 )
  t3 = create_thread( f3() )
  t4 = create_thread( f4() )
  join_thread( t3 )
  join_thread( t4 )
  ...
}
```

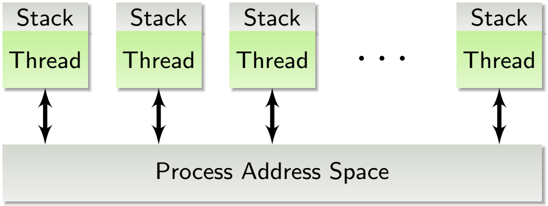
At any time during the runtime of a process, new threads may be spawned. Synchronising with the end of a thread is called *joining*.

All threads have a direct access to the address space of the process.



All communication between different threads may be accomplished by changing data in the common address space.

All threads have a direct access to the address space of the process.



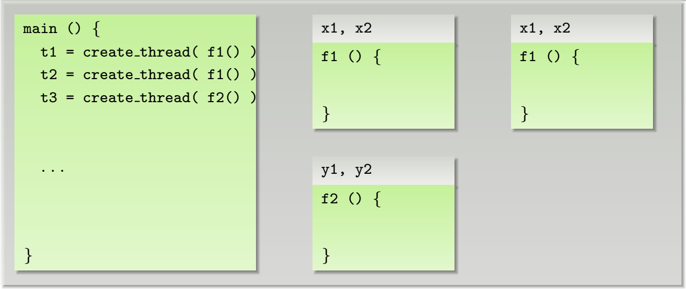
All communication between different threads may be accomplished by changing data in the common address space.

Furthermore, each thread executes a function which may have *local* variables stored on the *stack*. Such data is considered *thread-local* and must not be accessed by other threads.

As an example, the following threads execute functions with local variables x_1, x_2 in function `f1` and y_1, y_2 in function `f2`.

```
void f1 () {
  double  x1, x2;
  ...
}
```

```
void f2 () {
  double  y1, y2;
  ...
}
```


main () {
t1 = create_thread(f1())
t2 = create_thread(f1())
t3 = create_thread(f2())
...
}
x1, x2
f1 () {
}
x1, x2
f1 () {
}
y1, y2
f2 () {
}

These variables are local to the thread, the function is executed in.

As the variables are allocated *per function call*, the same function executed in different threads will create different variables.

Mutual exclusion

$\rightarrow$ code th01.cpp

Mutual exclusion

→ code th01.cpp

Consider the following algorithm for computing the sum $b = \sum_{i=0}^{3} a_i$.

```
double  a[4] = { 1, 2, 3, 4 };
double  b    = 0;
int main () {
  thread_t  t1 = create_thread( f1() );
  thread_t  t2 = create_thread( f2() );
  join_thread( t1 ); join_thread( t2 );
}
```

```
1 void f1 () {
2   double  t = a[0]+a[1];
3   b = b + t;
4 }
```

```
void f2 () {
  double  t = a[2]+a[3];
  b = b + t;
}
```

On the level of processor instructions, line 3 consists of several steps:

1. load data from memory position of `b`,
2. compute `b + t`,
3. store the result at memory position of `b`

Both threads may *simultaneously* execute these instructions, yielding different results depending on which threads execute which instruction first. The final value of `b` may either be 3, 7 or 10, depending on the *scheduling* of the threads.

In contrast to this, the local computation of `t` is uncritical, since only thread-local data is changed.

Critical section

Critical Section

The update of the variable `b` in both functions is called a *critical section*.

At any time at most *one* thread must be inside a critical section. Otherwise, the result of the computation is undefined.

Critical section

Critical Section

The update of the variable `b` in both functions is called a *critical section*.

At any time at most *one* thread must be inside a critical section. Otherwise, the result of the computation is undefined.

Mutex

To enable the *mutual exclusion* of several threads in critical sections, *mutex locks* (mutexes) are provided by the corresponding programming interfaces.

A mutex is always in one of two states, *locked* or *unlocked*:

locked: If a thread tries to lock an already locked mutex, the thread will block further computation until the mutex is unlocked.

unlocked: Any thread may lock the mutex. If multiple threads try to lock a mutex simultaneously, only one thread will succeed and all others will block.

Furthermore, if multiple threads will block on a locked mutex, unlocking the mutex will always unblock only a *single* thread.

To protect critical sections with mutexes, a shared mutex is locked before the critical section (function `lock()`) and unlocked afterwards (function `unlock()`):

```
double  a[4] = { 1, 2, 3, 4 };
double  b    = 0;

int main () {
  mutex_t   mutex;
  thread_t  t1 = create_thread( f1( mutex ) );
  thread_t  t2 = create_thread( f2( mutex ) );
  join_thread( t1 ); join_thread( t2 );
}
```

```
void f1 ( mutex_t & mutex ) {
  double  t = a[0]+a[1];

  lock( mutex );
  b = b + t;
  unlock( mutex );
}
```

```
void f2 ( mutex_t & mutex ) {
  double  t = a[2]+a[3];

  lock( mutex );
  b = b + t;
  unlock( mutex );
}
```

→ demo th02.cpp

Thread scheduling

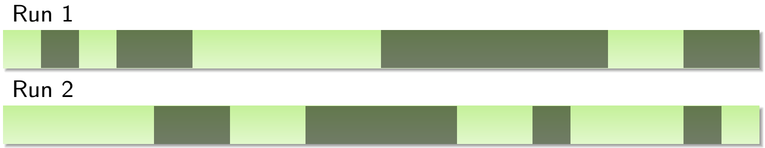
The mapping of threads to physical processors (or processor cores) is either performed by the operating system or by the software library providing the thread functionality.

Which thread is assigned to which processor depends on many factors, the main two being

- how many other processes or threads are running and
- topology of the processor configuration (cores in same processor),

Especially the number of other threads in the system is constantly changing. Hence, the mapping also changes constantly.

Furthermore, the times at which a thread is assigned CPU cycles are completely *undeterministic*.



Race Condition

If the outcome of a multi-threaded program changes with the scheduling of threads, a *race condition* is present.

Race conditions usually exist because of shared data and missing control of critical regions.

Thread interfaces

Threads can be accessed by different programming interfaces, e.g.:

- POSIX Threads,
- C++ Threads,
- OpenMP or
- Threading Building Blocks

Furthermore, many software libraries will also provide an interface for threads, usually based on one of the above frameworks.

They provide a different level of abstraction from the underlying thread implementation of the operating system.

Beside basic thread handling, e.g. creation and joining, functions for mutexes and mechanisms to alter thread scheduling are usually part of the thread interface.

POSIX threads

The most widely used thread interface are *POSIX* threads or *Pthreads*. Pthreads are designed to give the programmer almost full control over all aspects of threads, e.g. thread creation or thread scheduling, using a *low-level* interface, with a complex set of functions.

```
#include <pthread.h>

double  a[4] = { 1, 2, 3, 4 };
double  b    = 0;

void * f1 ( void * data ) {
  pthread_mutex_t * mutex = (pthread_mutex_t *) data;
  double            t     = a[0]+a[1];
  pthread_mutex_lock( mutex );
  b = b + t;
  pthread_mutex_unlock( mutex );
}
void * f2 ( void * data ) { ... }

int main () {
  pthread_t          t1, t2;
  pthread_attr_t     attr;
  pthread_mutex_t    mutex = PTHREAD_MUTEX_INITIALIZER;

  pthread_attr_init( & attr ); // change, e.g., scheduling policy, stack size
  pthread_create( & t1, & attr, f1, (void *) & mutex );
  pthread_create( & t2, & attr, f2, (void *) & mutex );
  pthread_attr_destroy( & attr );
  pthread_join( t1, NULL );
  pthread_join( t2, NULL );
}
```

C++ threads

With C++11, C++ provides data types for threads and mutexes, enabling simplified programming of thread parallel applications. ⇒ th03.cpp

C++ threads

With C++11, C++ provides data types for threads and mutexes, enabling simplified programming of thread parallel applications. ⇒ th03.cpp

```
#include <thread>
#include <mutex>

double  a[4] = { 1, 2, 3, 4 };
double  b    = 0;

void f1 ( std::mutex * mutex ) {
  double  t = a[0]+a[1];

  mutex->lock();
  b = b + t;
  mutex->unlock();
}
void f2 ( std::mutex * mutex ) { ... }

int main () {
  std::mutex    mutex;
  std::thread   t1( f1, & mutex );
  std::thread   t2( f2, & mutex );

  t1.join();
  t2.join();
}
```

C++ threads are best suited for simple thread programming, without the need for special scheduling or task handling. A typical example is a special I/O thread handling network communication.

OpenMP

OpenMP is a language extension to C, C++ and Fortran, providing special pragmas for handling parallel sections of a program.

```
double  a[4] = { 1, 2, 3, 4 };
double  b    = 0;

int main () {
  #pragma omp parallel
  {
    #pragma omp sections reduction (+:b)
    {
      #pragma omp section
      {
        double  t = a[0] + a[1];
        b = b + t;
      }
      #pragma omp section
      {
        double  t = a[2] + a[3];
        b = b + t;
} } } }
```

OpenMP also provides automatic task definition and scheduling when handling loops. It is even possible to map code sections to special targets, e.g. external accelerator cards.

When converting sequential to parallel programs, OpenMP often yields a good parallel efficiency with only a few changes to the source code and preserving most of the code structure.

Side effects of hardware and software

Certain features of parallel computers have a large influence on the behaviour of parallel programs.

This behaviour is often not directly visible to the programmer, especially, since all communication between different processors are performed using shared memory and the hardware providing the shared memory.

Such problems appear either directly due to hardware, especially memory caches, e.g.

- False Sharing and ⇒ lecture 2
- Atomic Operations

or indirectly by software, e.g.

- Thread Scheduling or
- Memory Allocation.

The most notable software in this context is of course the *Operating System*, providing access to the hardware.

Side effects of hardware and software

Certain features of parallel computers have a large influence on the behaviour of parallel programs.

This behaviour is often not directly visible to the programmer, especially, since all communication between different processors are performed using shared memory and the hardware providing the shared memory.

Such problems appear either directly due to hardware, especially memory caches, e.g.

- False Sharing and $\Rightarrow$ lecture 2
- Atomic Operations

or indirectly by software, e.g.

- Thread Scheduling or
- Memory Allocation.

The most notable software in this context is of course the *Operating System*, providing access to the hardware.

Atomic

Normally, even updates such as

```
++x;
```

or

```
y = y + x;
```

of variables of an elementary data type, e.g. `int` or `double` will require several CPU instructions, e.g. load, arithmetic and store commands. These commands may be interrupted by other threads, potentially leading to a race condition.

However, a special set of CPU instructions will provide *atomic*, indivisible operations, which perform load, arithmetic and store in one step.

Remark

In C++, atomic operations are provided by the `atomic` class.

Using atomic instructions, the program

```
void f1 ( std::atomic< double > * x ) {        void f2 ( std::atomic< double > * x ) {
  *x += 1;                                       *x += 2;
}                                              }

int main () {
  std::atomic< double >   x = 0.0;
  std::thread             t1( f1, & x );
  std::thread             t2( f2, & x );

  t1.join(); t2.join();
}
```

will always yield the same output, although no mutex is used.

Remark

Mutexes itself are based on such atomic instructions.

Since atomic instructions directly change the main memory, the cost for changing such a variable is much higher than for normal operations.

Furthermore, if other processors share the atomic variable, their cache entry will be invalidated by the atomic instruction, enforcing a reload from memory.

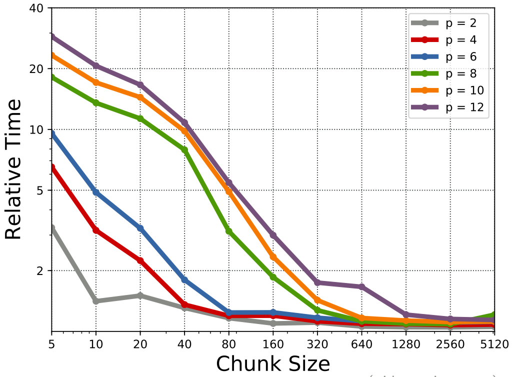
An example for the usage of atomics is a counter for the number of certain operations in a program:

```
void worker ( std::atomic< size_t > * counter ) {
  do {
    perform_work();
    (*counter)++;
  } while ( ! finished );
}

int main () {
  std::atomic< size_t >       counter( 0 );
  std::vector< std::thread >  threads( p );

  for ( int i = 0; i < p; ++i )
    threads[i] = std::thread( worker, & counter );

  //...
}
```

Here, the atomic counter may pose a critical bottleneck if the actual work in `perform_work` is small.

In such a case, the main work of the program consists of cache updates of all processors with the content of the counter variable. The behaviour of the runtime is then identical to the false sharing case, e.g. limited speedup or even *speed down*.

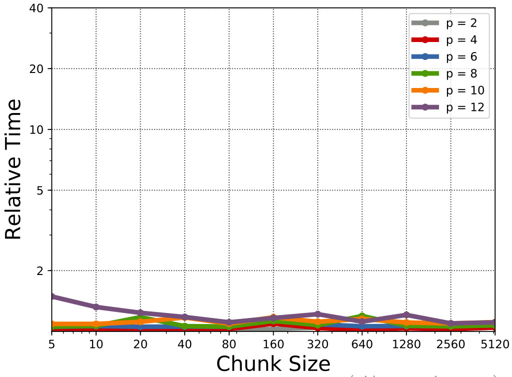
In the following example, the work per thread stays constant, only the (*chunk*) size of the work data changes. The shared atomic variable will be used to count the iterations. All times are multiples of the corresponding sequential runtime.



(with atomic counter)

The larger the chunk size, the less effect the atomic operations have. But especially for small work per thread, the computation is dominated by memory loads and has a significantly larger runtime than the sequential implementation.

In the following example, the work per thread stays constant, only the (*chunk*) size of the work data changes. The shared atomic variable will be used to count the iterations. All times are multiples of the corresponding sequential runtime.

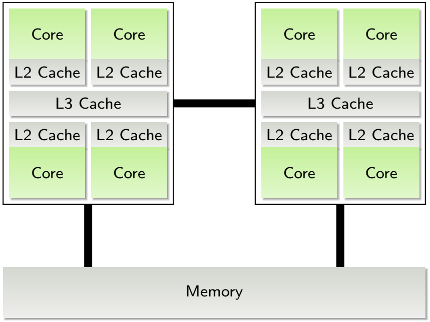


(without atomic counter)

The larger the chunk size, the less effect the atomic operations have. But especially for small work per thread, the computation is dominated by memory loads and has a significantly larger runtime than the sequential implementation.

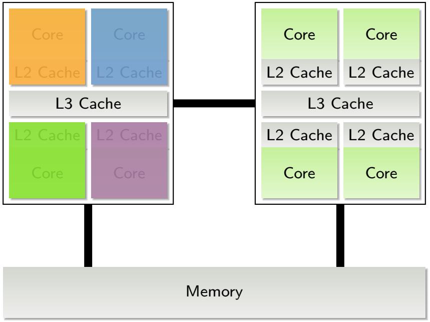
Thread scheduling

A typical hardware configuration of a compute server consists of two processors, each having several cores:



Thread scheduling

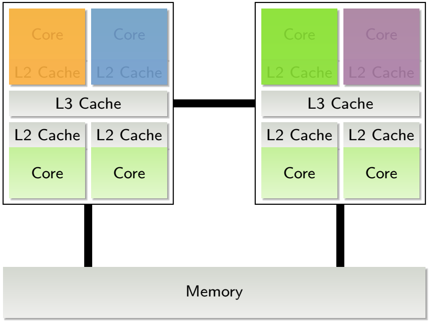
A typical hardware configuration of a compute server consists of two processors, each having several cores:



For a multi-threaded program it may have a severe impact on the runtime if all threads will be mapped to cores of the same processor with a joined L3 cache (*high communication speed*) or if the threads are mapped to different processors (*larger cache per thread*).

Thread scheduling

A typical hardware configuration of a compute server consists of two processors, each having several cores:



For a multi-threaded program it may have a severe impact on the runtime if all threads will be mapped to cores of the same processor with a joined L3 cache (*high communication speed*) or if the threads are mapped to different processors (*larger cache per thread*).

Although the performance difference will be based on properties of the hardware, the actual mapping of the threads will be defined by software.

By default, thread scheduling is handled by the *scheduler* of the operating system. A scheduler is responsible for

- mapping threads to processors and
- assigning slices of processor runtime to threads.

In the standard case, the operating system is allowed to schedule a thread to *any* processor.

Control of the scheduling by the programmer is possible using processor affinity functions of the OS.

Processor Affinity

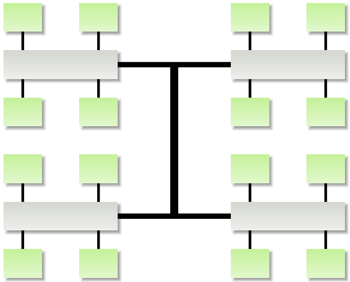
Each thread has a mask defining at which processor it may be be executed, the *CPU affinity mask*.

With it, the set of processors for a specific thread may be limited to cores of a single physical processor or to separate processors.

However, for many thread-parallel programs, the OS scheduler will provide a reasonable scheduling, yielding close to optimal performance.

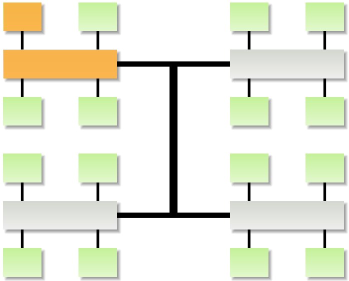
Memory allocation

On Non-Uniform Memory Access (NUMA) systems the actual position of the allocated memory of a program in the global memory has a direct influence on the performance of the program, although local caches will often hide different memory speed.



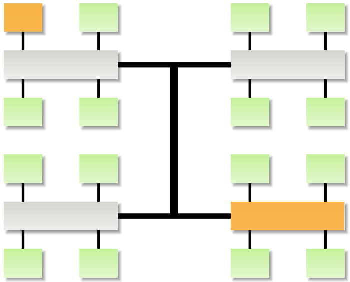
Memory allocation

On Non-Uniform Memory Access (NUMA) systems the actual position of the allocated memory of a program in the global memory has a direct influence on the performance of the program, although local caches will often hide different memory speed.



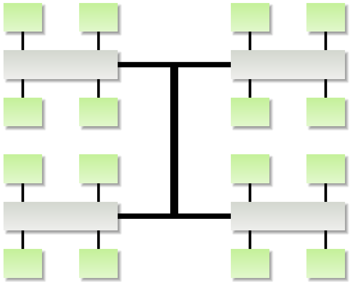
Memory allocation

On Non-Uniform Memory Access (NUMA) systems the actual position of the allocated memory of a program in the global memory has a direct influence on the performance of the program, although local caches will often hide different memory speed.



Memory allocation

On Non-Uniform Memory Access (NUMA) systems the actual position of the allocated memory of a program in the global memory has a direct influence on the performance of the program, although local caches will often hide different memory speed.



There are different reasons for threads accessing non-local memory, e.g.

- remapping of a thread to a different processor,
- moving data handling in the program from one thread to another,
- the memory allocation routine, e.g. `malloc`.

Often data is allocated in one thread and used for computations in another thread:

```
void f1 ( std::vector< double > * x ) {
  x->resize( n );    // allocate memory
  initialize( x );
}
```

```
void f2 ( std::vector< double > * x ) {
  compute_with( x );
}
```

```
int main () {
  std::vector< double >    x;
  std::thread              t1, t2;

  t1 = std::thread( f1, & x );
  t1.join();

  t2 = std::thread( f2, & x );
  t2.join();
}
```

Depending on the processor mapping of both threads, the second thread may work with remote memory and hence, has sub-optimal memory access.

Rules for thread-parallel programming

Critical rules, that should always be followed in thread-parallel programs are:

1. Avoid False Sharing by separating shared data or use thread-private data.
2. Sparsely use atomic variables.

The following rules will often only have a minor effect on the parallel performance but may be critical for special algorithms:

3. Bind threads to processors depending on the data exchange pattern.
4. Keep allocated memory local to threads.

Why Use Version Control?
Slides adapted from Andreas Skielboe

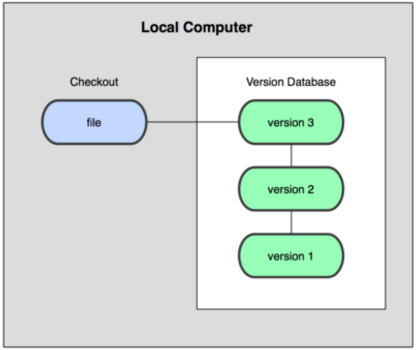
A Version Control System (VCS) is an integrated fool-proof framework for

- Backup and Restore
- Short and long-term undo
- Tracking changes
- Synchronization
- Collaborating
- Sandboxing

... with minimal overhead.

Local Version Control Systems

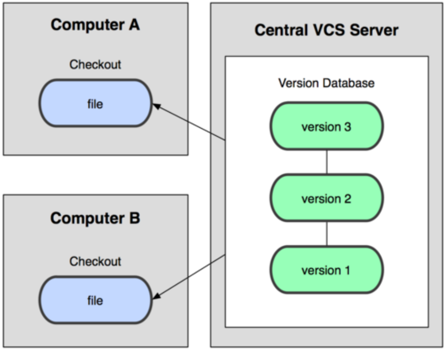
Conventional version control systems provides some of these features by making a local database with all changes made to files.



Any file can be recreated by getting changes from the database and patch them up.

Centralized Version Control Systems

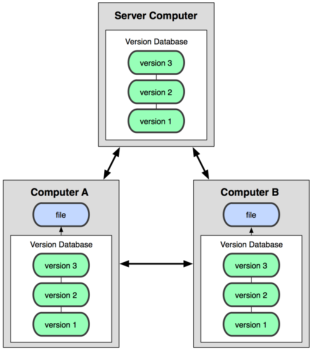
To enable synchronization and collaborative features the database is stored on a central VCS server, where everyone works in the same database.



Introduces problems: single point of failure, inability to work offline.

Distributed Version Control Systems

To overcome problems related to centralization, distributed VCSs (DVCSs) were invented. Keeping a complete copy of database in every working directory.



Actually the most **simple** and most **powerful** implementation of any VCS.

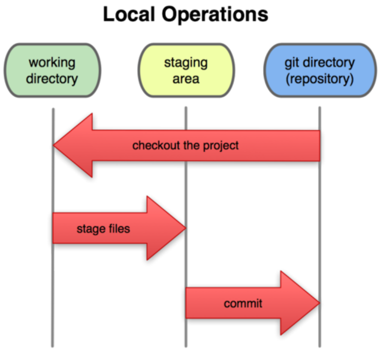
Git Basics - The Git Workflow

The simplest use of Git:

- **Modify** files in your *working directory*.
- **Stage** the files, adding snapshots of them to your *staging area*.
- **Commit**, takes files in the staging area and stores that snapshot permanently to your *Git directory*.

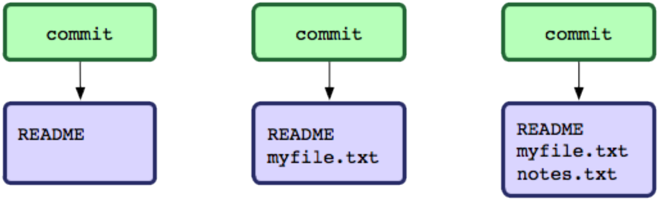
Git Basics - The Three States

The three basic states of files in your Git repository:



Git Basics - Commits

Each commit in the git directory holds a snapshot of the files that were staged and thus went into that commit, along with author information.



Each and every commit can always be looked at and retrieved.

Git Basics - Working with remotes

In Git **all remotes are equal**.

A *remote* in Git is nothing more than a link to another git directory.

Git Basics - Working with remotes

The easiest commands to get started working with a remote are

- *clone*: Cloning a remote will make a complete local copy.
- *pull*: Getting changes from a remote.
- *push*: Sending changes to a remote.

Hands-on - First-Time Git Setup

Before using Git for the first time:

Pick your identity

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

Check your settings

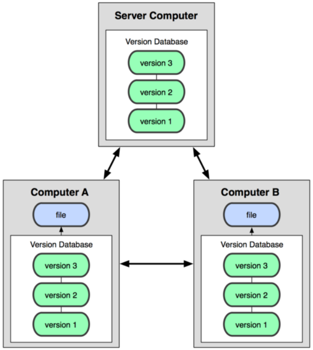
```
$ git config --list
```

Get help

```
$ git help <verb>
```

Hands-on - Getting started with a bare remote server

Using a Git server (ie. no working directory / *bare* repository) is the analogue to a regular centralized VCS in Git.



Hands-on - Getting started with remote server

When the remote server is set up with an initialized Git directory you can simply *clone* the repository:

Cloning a remote repository

```
$ git clone <repository>
```

You will then get a complete local copy of that repository, which you can edit.

Hands-on - Getting started with remote server

With your local working copy you can make any changes to the files in your working directory as you like. When satisfied with your changes you add any modified or new files to the staging area using *add*:

Adding files to the staging area

```
$ git add <filepattern>
```

Hands-on - Getting started with remote server

Finally to commit the files in the staging area you run *commit* supplying a *commit message*.

Committing staging area to the repository

```
$ git commit -m <msg>
```

Note that so far **everything is happening locally** in your working directory.

Hands-on - Getting started with remote server

To **share your commits** with the remote you invoke the *push* command:

Pushing local commits to the remote

```
$ git push
```

To recieve changes that other people have pushed to the remote server you can use the *pull* command:

Pulling remote commits to the local working directory

```
$ git pull
```

And **thats it**.

Hands-on - Summary

Summary of a minimal Git workflow:

- `clone` remote repository
- `add` you changes to the staging area
- `commit` those changes to the git directory
- `push` your changes to the remote repository

- `pull` remote changes to your local working directory

References

Some good Git sources for information:

- Git Community Book - http://book.git-scm.com/
- Pro Git - http://progit.org/
- Git Reference - http://gitref.org/
- GitHub - http://github.com/
- Git from the bottom up - http://ftp.newartisans.com/pub/git.from.bottom.up.pdf
- Understanding Git Conceptually - http://www.eecs.harvard.edu/~cduan/technical/git/
- Git Immersion - http://gitimmersion.com/

Applications

GUIs for Git:

- GitX (MacOS) - http://gitx.frim.nl/
- Giggle (Linux) - http://live.gnome.org/giggle

What should (not) be added to a repository?

Git tracks diff-files to keep its memory requirements small. Main rule: mostly add *source files that compile*.

- .c, .cpp, .f files YES!
- .tex files YES!
- .aux, .out, .dvi. . . files NO!
- compiled files, object files NO! (large, no diffs possible, conflicts)
- .pdf files YES/NO!
- large data files NO. . . sometimes maybe
- photos, movies etc. NO! (unless unavoidable)

My rule of thumb: Files in the repository are permanent, only the best should make it in there (it's not your trash can!) They should compile (code/Latex), be (more or less) cleaned up, unless it's avoidable only source/text files.