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

March 2, 2020

Slightly adapted from Georg Stadler's lectures.

Today

Last lecture

- Performance models work depth
- Parallel programming models
- Performance models Amdahl's law
- Threads
- Tool: git

Today

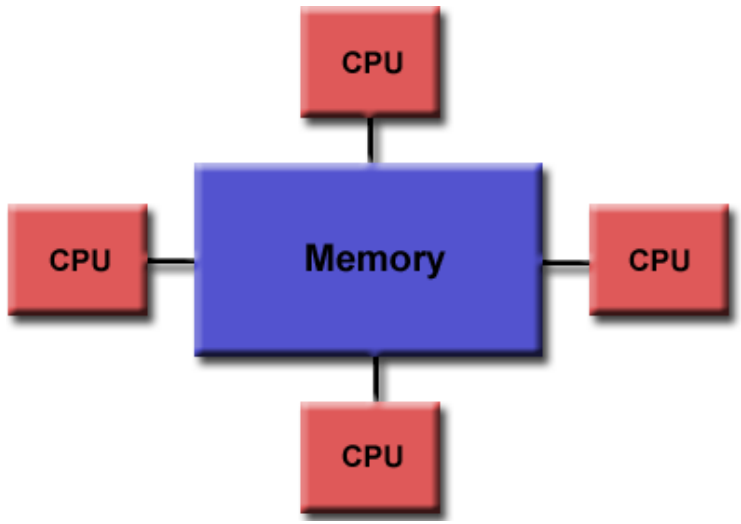
- OpenMP I

Announcements and upcoming

- Extended deadline for HW2

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`

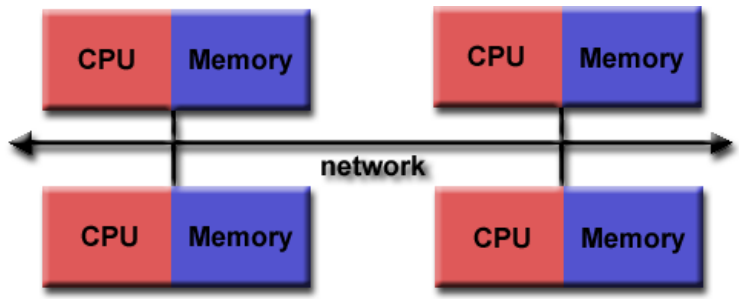
Shared memory programming model

Advantages and disadvantages:

+ Relative easy to parallelize loops etc. in serial programs
- Limited amount of parallelism possible in practice
- Memory bus becomes bottleneck if too many processors access the same memory (usually used with ≤ 50 cores)
- Cache coherency: Need to make sure that values stored in cache of each processor coincide (handled by hardware)
- Size of shared memory is limited and can get very expensive

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).

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

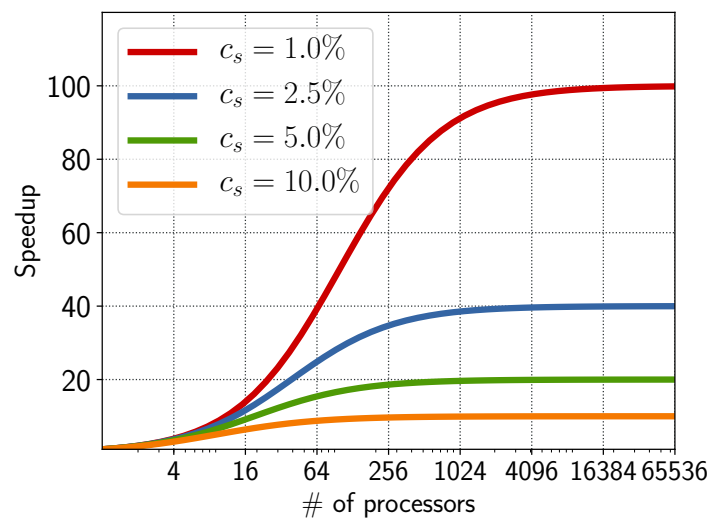
$$t(p) = c_s t(1) + \frac{(1 - c_s)}{p} t(1)$$

This leads to

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

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

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



Load (im)balance in parallel computations

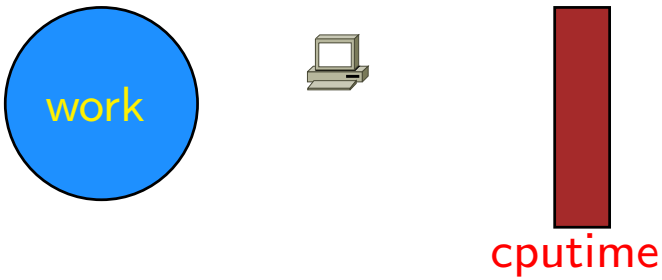
In parallel computations, the work should be distributed *evenly* across workers/processors.

- Load imbalance: Idle time due to insufficient parallelism or unequal sized tasks
- Initial/static load balancing: distribution of work at beginning of computation
- Dynamic load balancing: work load needs to be re-balanced during computation. Imbalance can occur, e.g., due to
  - adapting (mesh refinement)
  - in completely unstructured problems

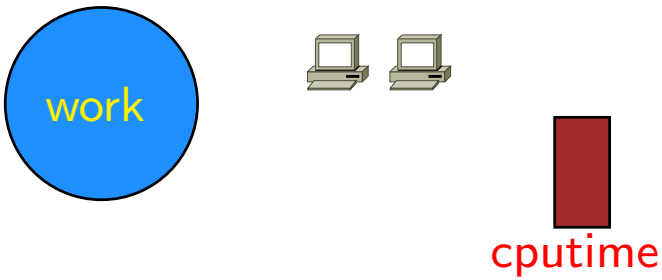
Parallel scalability

Strong and weak scaling/speedup

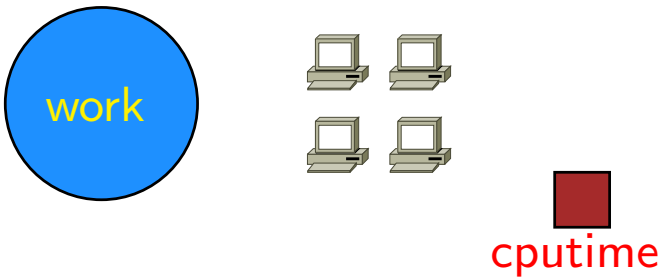
Strong scalability



Strong scalability



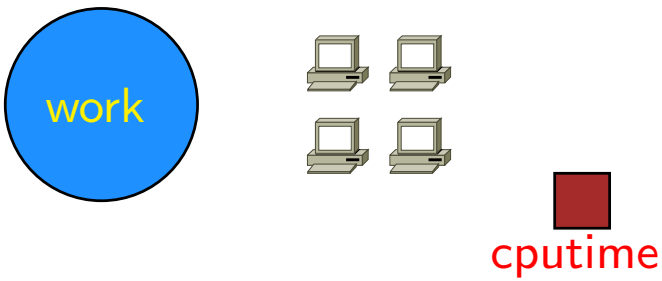
Strong scalability

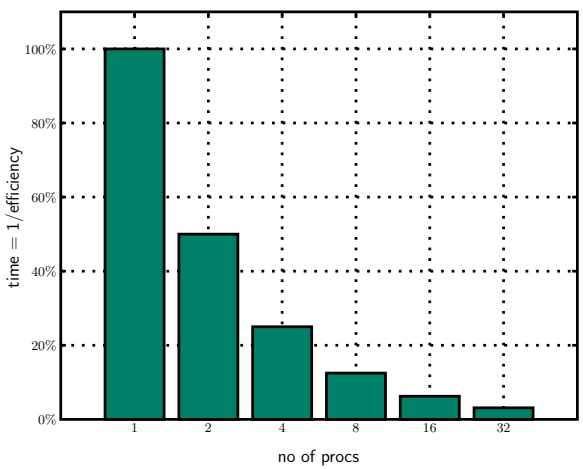


Parallel scalability

Strong and weak scaling/speedup

Strong scalability

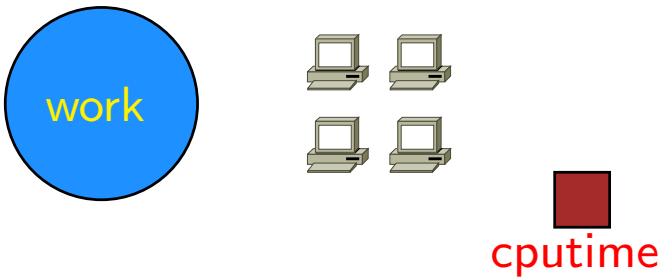


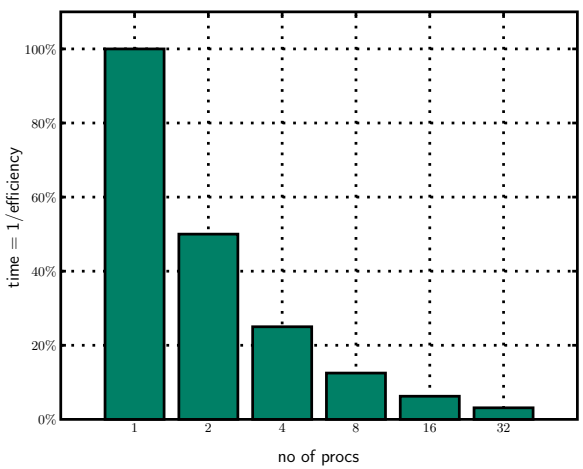


Parallel scalability

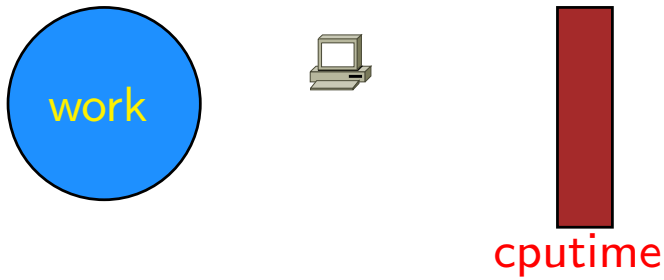
Strong and weak scaling/speedup

Strong scalability





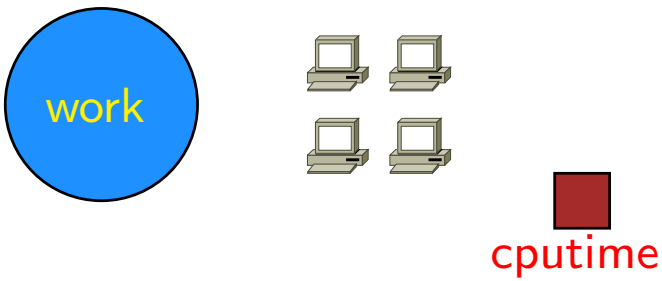
Weak scalability

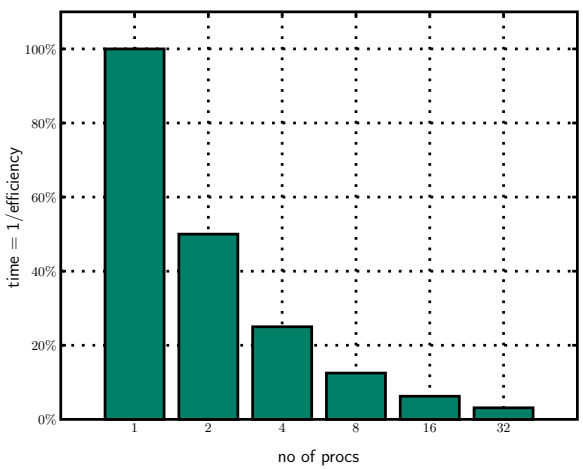


Parallel scalability

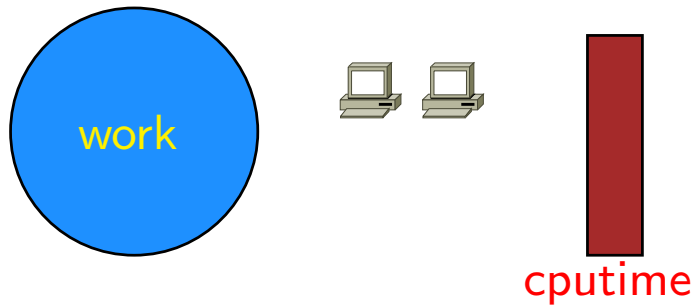
Strong and weak scaling/speedup

Strong scalability





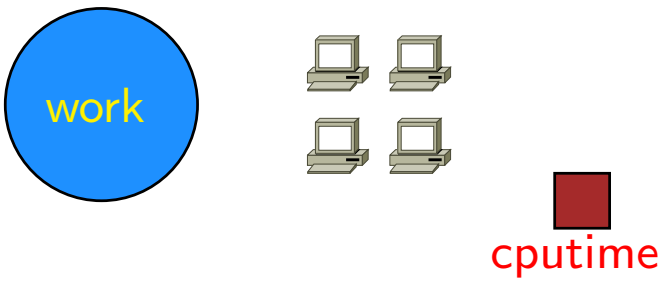
Weak scalability



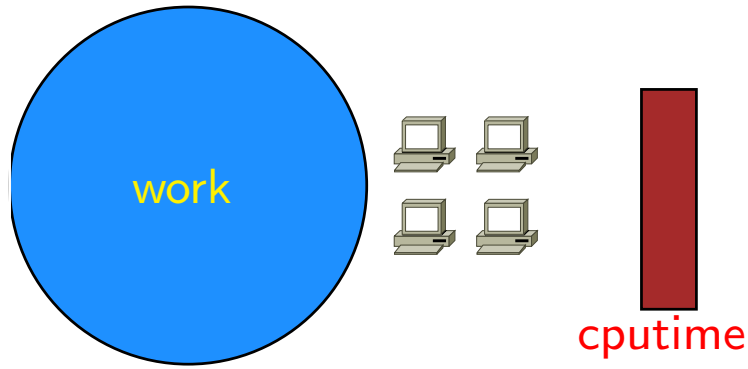
Parallel scalability

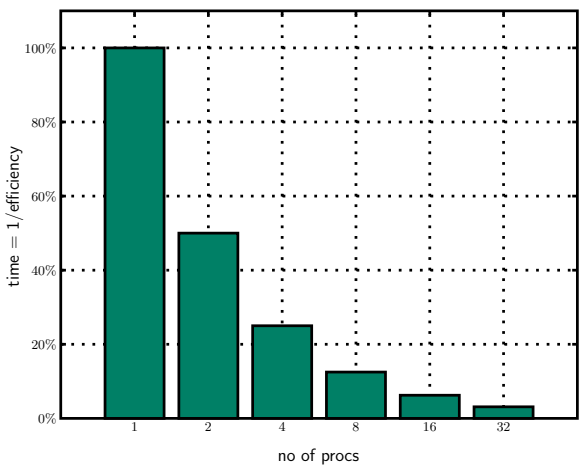
Strong and weak scaling/speedup

Strong scalability



Weak scalability

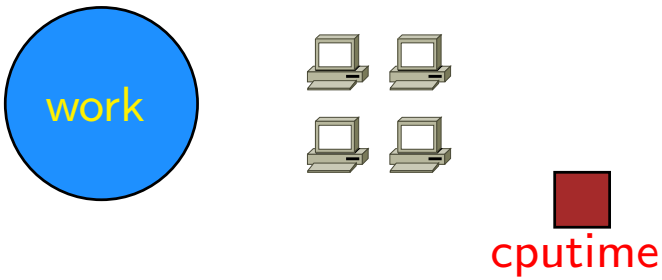


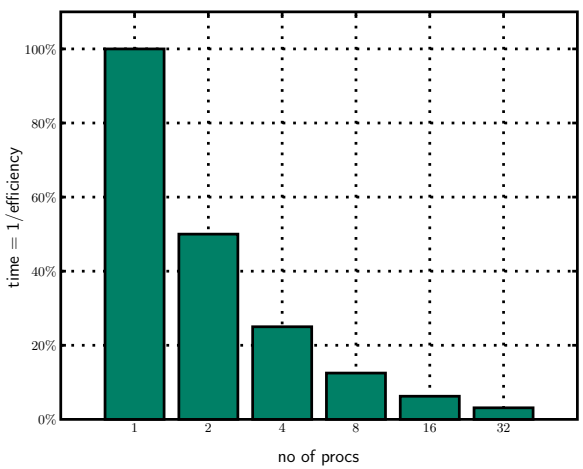


Parallel scalability

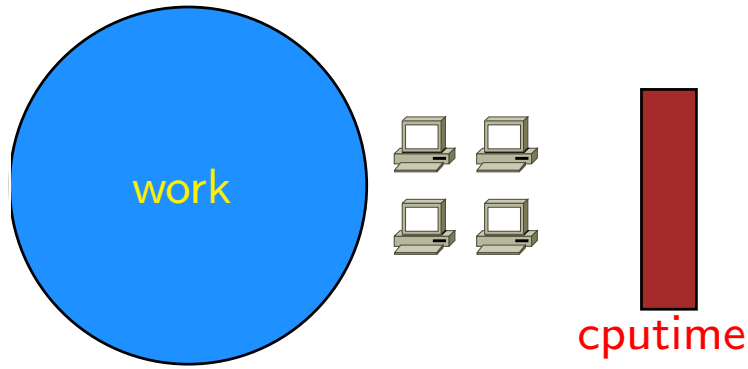
Strong and weak scaling/speedup

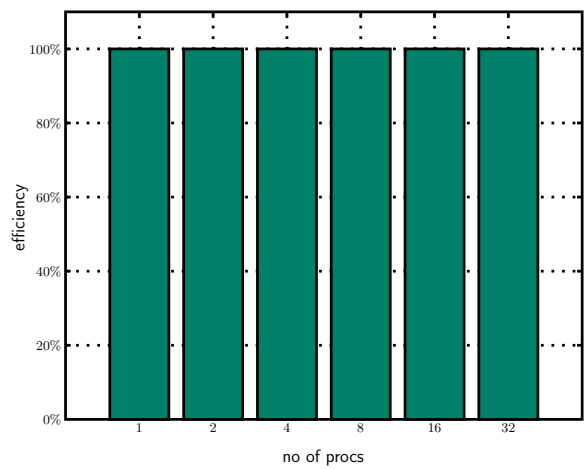
Strong scalability





Weak scalability





OpenMP

History

OpenMP started as a joined initiative of most of the major hardware and software vendors to provide compiler support for parallel programs, with a main focus on parallelising loops.

- 1997: OpenMP 1.0 for Fortran
- 1998: OpenMP 1.0 for C/C++
- 2000: OpenMP 2.0 for Fortran with fixes and clarifications
- 2002: OpenMP 2.0 for C/C++
- 2005: OpenMP 2.5 combining Fortran and C/C++
- 2008: OpenMP 3.0 introduces tasks
- 2011: OpenMP 3.1
- 2013: OpenMP 4.0 with support for vector units and special targets
- 2018: OpenMP 5.0 with support for accelerator cards

Beside the compiler directives (pragmas), OpenMP also contains a set of data types and functions, imported via the header file `omp.h`.

All major C/C++ and Fortran compilers support OpenMP

Hello world

The standard starting point for each programming lecture is:

```
#include <iostream>
int main () {
  #pragma omp parallel
  std::cout << "Hello, world!" << std::endl;
}
```

To enable OpenMP during compilation, a special compiler flag has to be provided:

Intel Compiler

```
> icpc -openmp -o hello -c hello.cc
```

GNU Compiler

```
> g++ -fopenmp -o hello -c hello.cc
```

Possible program output on a Quad-Core CPU may then be:

```
Hello, world!
Hello, world!
Hello, world!
Hello, world!
```

```
Hello, world!Hello, world!Hello,
     world!
Hello, world!
```

DEMO: omp01.cpp

Remark

Question: Why do we see garbled output?

Hello world

The standard starting point for each programming lecture is:

```
#include <iostream>
int main () {
  #pragma omp parallel
  std::cout << "Hello, world!" << std::endl;
}
```

To enable OpenMP during compilation, a special compiler flag has to be provided:

Intel Compiler

```
> icpc -openmp -o hello -c hello.cc
```

GNU Compiler

```
> g++ -fopenmp -o hello -c hello.cc
```

Possible program output on a Quad-Core CPU may then be:

```
Hello, world!
Hello, world!
Hello, world!
Hello, world!
```

```
Hello, world!Hello, world!Hello,
     world!
Hello, world!
```

DEMO: omp01.cpp

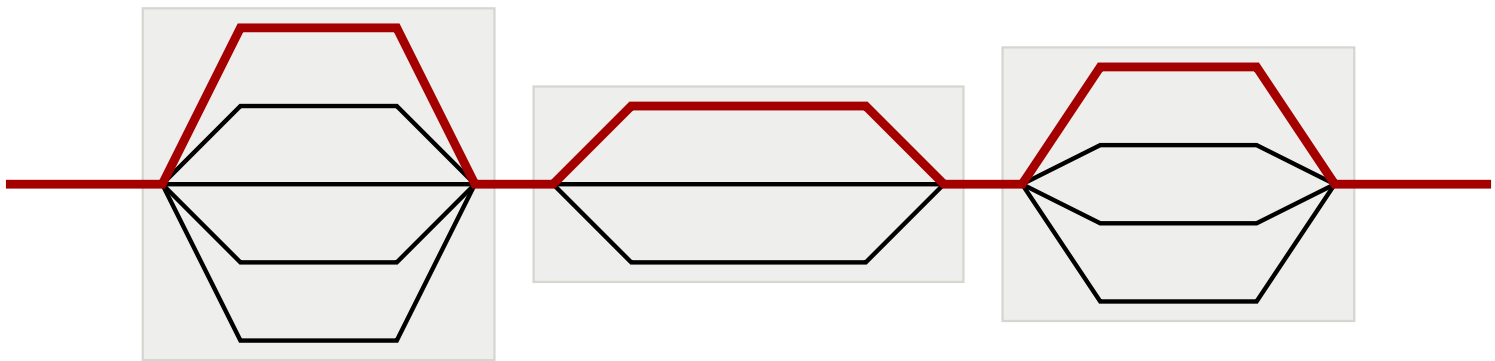
Remark

Question: Why do we see garbled output? The garbled output is due to a race condition between all threads competing for the same output channel.

Fork-join model

The underlying model of OpenMP is the *fork-join model*:

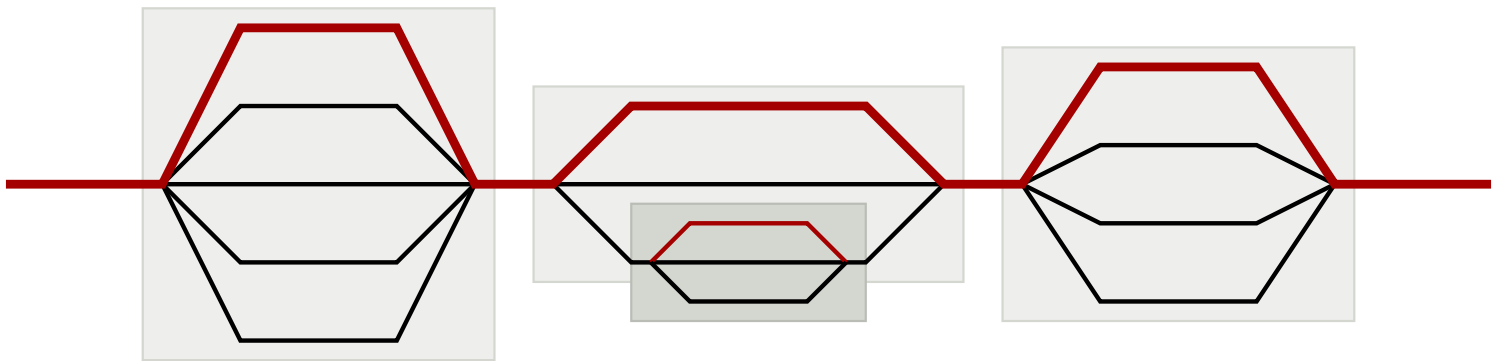
> *A master thread creates a team of worker threads which run in parallel together with the master thread until all worker threads finish.*



Fork-join model

The underlying model of OpenMP is the *fork-join model*:

> *A master thread creates a team of worker threads which run in parallel together with the master thread until all worker threads finish.*



The fork-join model is recursive: any thread in a team may create new threads, which form a new team. This is known as *nested parallelism*.

The number of threads in a team is user defined and may vary between different parallel sections of the program.

When all threads of a team are joined, a synchronisation takes place, i.e. the master thread of the team will only proceed when all other team threads have finished.

OpenMP directives

OpenMP extends the underlying programming language by compiler *directives*. Each OpenMP directive starts with the pragma

```
#pragma omp <directive> new-line
```

Remark

The new-line is mandatory!

The directive is applied to the immediately following source code block, i.e.

```
#pragma omp <directive>
{
  ...
}
```

or just

```
#pragma omp <directive>
  ...
```

in case of a single-line block.

After the source code block, the worker threads of the thread team will stop and only the master thread will proceed.

Two important concepts appear in the context of OpenMP directives.

Construct

The *construct* of an OpenMP directive includes only the block of source code directly following the directive.

Region

The (parallel) *region* of an OpenMP directive is the set of all code executed by a thread of the created team. This also includes all code executed in called functions.

```
void f () {
  ...                    // part of the region but NOT the construct
}

#pragma omp <directive>
{
  ...                    // construct of the directive
  f();
  ...
}
```

Parallel vs Sequential Mode

In case OpenMP is not supported or explicitly turned off during compilation, all OpenMP directives will be ignored and the output is a standard sequential program.

```
> g++ -Wall -o hello hello.cc
hello.cc:5:0: warning: ignoring #pragma omp parallel [-Wunknown-pragmas]
```

Most OpenMP implementations will support parallel *and* sequential mode of programs containing OpenMP directives. Albeit, it is legal to develop a program, which will only work correctly in parallel mode!

Furthermore, the output of the program may vary between sequential and parallel mode.

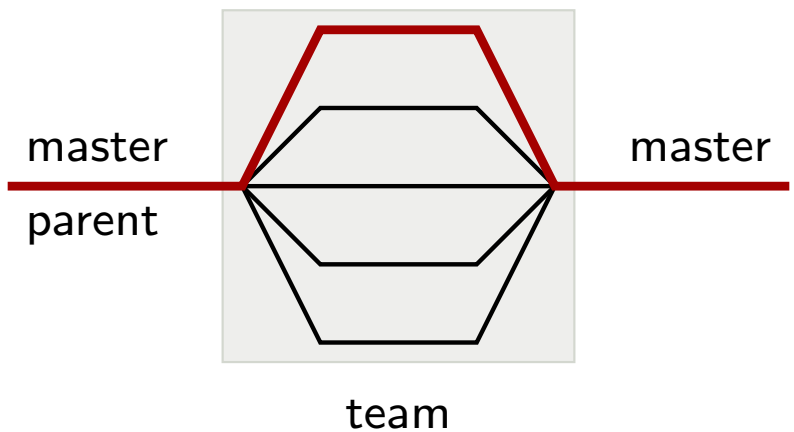
Thread creation

A new team of threads is created by the directive

```
#pragma omp parallel [clause1 [[,] clause2, ...]]
```

The thread, which initially encounters the `parallel` directive is called the *master* thread and furthermore, becomes the *parent* thread to all threads in the newly created team.

```
void f () {
  ...                          // executed by master thread
  #pragma omp parallel
  {
    ...                        // executed by team
  }
  ...                          // executed by master thread
}
```



The default number of threads in the team (including the master thread) is chosen by the OpenMP library, and usually equal to the number of processors (cores) in the system.

DEMO: omp01.cpp

The default number of threads in the team (including the master thread) is chosen by the OpenMP library, and usually equal to the number of processors (cores) in the system.

DEMO: omp01.cpp

Instead of this, the following options may be used to change the number of team threads:

- With the clause `num_threads()`:

```
#pragma omp parallel num_threads(4)
```

- With the function `omp_set_num_threads()`:

```
#include <omp.h>
int main () {
  omp_set_num_threads( 4 );
  #pragma omp parallel
  { ... }
}
```

- Using the environment variable `OMP_NUM_THREADS`:

```
> export OMP_NUM_THREADS=4   # bash
```

or

```
> setenv OMP_NUM_THREADS 4   # tcsh
```

Remark

A typical programming mistake is to forget the `parallel` directive:

```
int main () {
  #pragma omp                // no "parallel"
  {
    ...
  }
}
```

In this case, no new threads are created and the construct will be executed by the master thread alone, i.e. *sequential* execution.

Remark

A typical programming mistake is to forget the `parallel` directive:

```
int main () {
  #pragma omp                 // no "parallel"
  {
    ...
  }
}
```

In this case, no new threads are created and the construct will be executed by the master thread alone, i.e. *sequential* execution.

Thread ID

Each thread of a team has an id $< p$ which is unique for the corresponding parallel region. The master thread will always have id 0.

This id may be obtained using the function `omp_get_thread_num()`:

```
#include <omp.h>
int main () {
  #pragma omp parallel
  {
    const int  thread_id = omp_get_thread_num();
    ...
  }
}
```

Nested parallelism

The `parallel` directive may also be *nested*:

```
void f () {
  #pragma omp parallel
  {
    ...
    #pragma omp parallel
    {
      ...
    }
    ...
  }
}
```

Usually, this nested mode has to be enabled *explicitly* using either the environment variable `OMP_NESTED`:

```
> export OMP_NESTED=TRUE   # bash
> setenv OMP_NESTED TRUE   # tcsh
```

or with the OpenMP function `omp_set_nested()`:

```
omp_set_nested( true );
```

If nested parallelism is not enabled, the nested directives will be handled by only one thread, i.e. sequentially.

Since, by default, the number of threads for a parallel region equals the number of processors, the second parallel region will *not* spawn new threads.

Hence, nested parallel regions should be used together with the `num_threads` clause:

```
void f () {
  #pragma omp parallel num_threads(4)
  {
    ...
    #pragma omp parallel num_threads(2)
    {
      ...
    }
    ...
  }
}
```

Here, each of the 4 worker threads will create a new team with 2 threads each, i.e. the inner construct is executed by 8 threads.

Nested parallelism may be used within the whole region of a `parallel` directive:

```
void g () {
  #pragma omp parallel      // inside parallel region of f()
  {
    ...
  }
}

void f () {
  #pragma omp parallel
  {
    ...
    g();
    ...
  }
}
```

This also allows recursive computations to be handled by nested parallel regions:

```
void f () {
  #pragma omp parallel num_threads(2)
  {
    f();
  }
}
```

Here, each call of `f()` will spawn 2 new threads until all processors are used.

Shared vs. private data

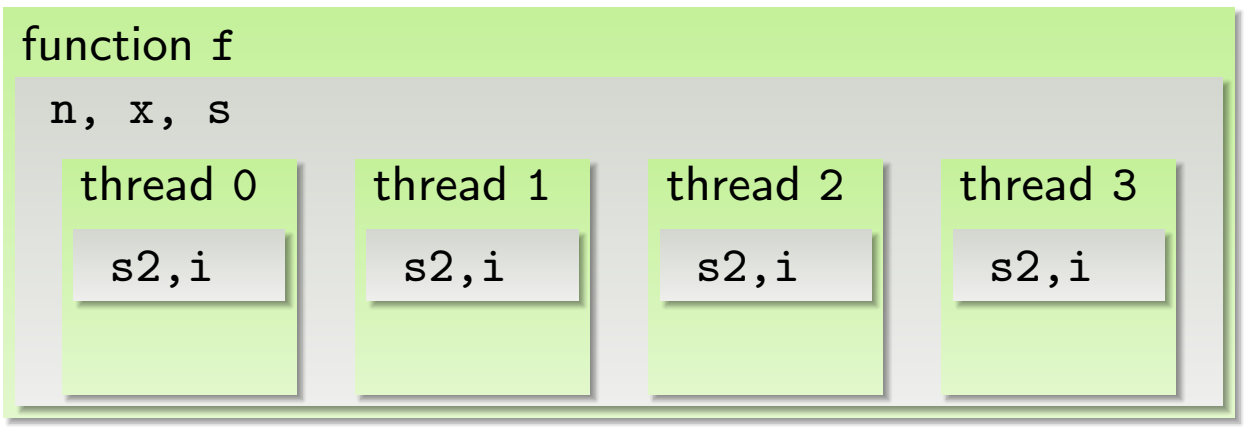
All data defined in the surrounding block of a parallel construct is *shared* by all threads.

Data defined within a parallel construct is *private* to a specific thread.

At line 4, the variables `n`, `x` and `s` are defined. In the parallel construct 5-12, these variables are shared by all threads.

The variables `s2` and `i` are defined within the parallel construct and are therefore, private to each thread.

```
1  void f ( int n, double * x ) {
2    double  s = 0.0;
3
4    #pragma omp parallel
5    {
6      double s2 = 0;
7
8      for ( int i = 0; i < n; ++i )
9        s2 += x[i];
10
11     s = s+s2;
12   }
13 }
```



Do you see a problem in this program?

Shared vs. private data

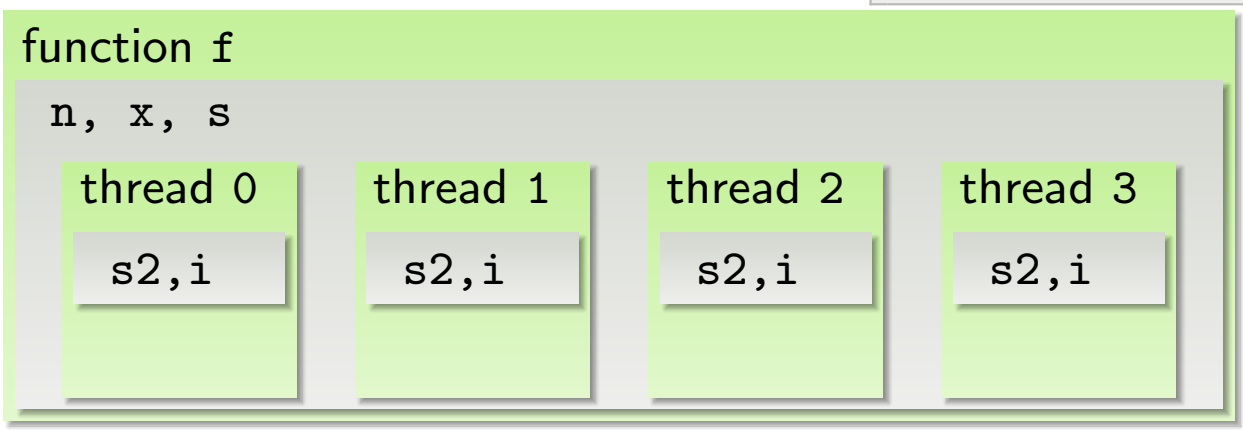
All data defined in the surrounding block of a parallel construct is *shared* by all threads.

Data defined within a parallel construct is *private* to a specific thread.

At line 4, the variables `n`, `x` and `s` are defined. In the parallel construct 5-12, these variables are shared by all threads.

The variables `s2` and `i` are defined within the parallel construct and are therefore, private to each thread.

```
1  void f ( int n, double * x ) {
2    double  s = 0.0;
3
4    #pragma omp parallel
5    {
6      double s2 = 0;
7
8      for ( int i = 0; i < n; ++i )
9        s2 += x[i];
10
11     s = s+s2;
12   }
13 }
```



Do you see a problem in this program? Shared variable s is updated, yielding a race condition

Any change to a shared variable will affect all threads of the thread team, and hence, defines a critical section.

```
1  void f ( int n, double * x ) {
2    double  s = 0.0;
3
4    #pragma omp parallel
5    {
6      double s2 = 0;
7
8      for ( int i = 0; i < n; ++i )
9        s2 += x[i];
10
11     s = s+s2;
12   }
13 }
```

In the example, at line 11 the shared variable `s` is updated, yielding a race condition.

In the `for` loop at lines 8 and 9, the shared variables `n` and `x` are accessed read-only. As long as the content of both variables is not changed by a thread, such accesses are uncritical.

Shared and default clause

With the clause

```
#pragma omp parallel shared(x1,x2,...)
```

a list of variables is defined, which exist in the scope of the code block *before* entering the parallel construct and which should be shared within the parallel construct.

Since by default, all such variables are shared, the statement would normally have no effect.

This standard behaviour can be changed by the clause

```
#pragma omp parallel default(shared|none)
```

with

`default(shared)`: representing the default behaviour, i.e. variables are shared,

`default(none)`: disable automatic variable sharing.

In the latter case, the status of any non-private variable referenced within the parallel construct has to be defined explicitly, e.g. by the `shared` clause.

DEMO: omp02.cpp

The combination of both clauses allows to limit the risk of unwanted side effects, e.g. if shared variables are erroneously accessed:

```
void f ( int n, double * x ) {
  double  x1[10], x2[100];
  double  s = 0.0;

  #pragma omp parallel default(none) shared(n,x,s)
  {
    double s2 = 0;
    for ( int i = 0; i < n; ++i )
      s2 += x1[i];   // error reported by compiler
    s = s+s2;
  }
}
```

Without `default` and `shared`, the compiler would not report any error.

Private clause

Outside variables may also be declared *private* for parallel constructs:

```
#pragma omp parallel private(var1,var2,...)
```

For each variable listed in the `private` clause, a new thread-private variable is created but *not initialised*. In particular, the thread-private copy will not have the same value as the variable in the surrounding scope.

This clause is therefore useful for automatic creation of thread-private variables, which are mainly used for local work and not for data sharing.

DEMO: omp03.cpp

Private clause

Outside variables may also be declared *private* for parallel constructs:

```
#pragma omp parallel private(var1,var2,...)
```

For each variable listed in the `private` clause, a new thread-private variable is created but *not initialised*. In particular, the thread-private copy will not have the same value as the variable in the surrounding scope.

This clause is therefore useful for automatic creation of thread-private variables, which are mainly used for local work and not for data sharing.

DEMO: omp03.cpp

Some limitations apply to the `private` clause:

- The private variable must not appear inside a compound

```
struct { double  real, imag; } c;
#pragma omp parallel private(c.real) // illegal: struct member
```

or array data structure

```
double y[10];
#pragma omp parallel private(y[5])   // illegal: array member
```

- The variable must not be declared `const`.

```
const double  x = 1.0;
#pragma omp parallel private(x)   // illegal: const variable
```

- If the variable is a class, an accessible default constructor must exist.

```
struct A {
  double  x, y;

  A ( double x, double y );

private:
  A ();
};

A  a( 1, 2 );

#pragma omp parallel private(a)   // illegal: A() is not accessible
```

- Reference types are not allowed.

```
void f ( double & y ) {
  #pragma omp parallel private(y)   // illegal: reference type
  ...
}
```

Construct vs. Region

The `private` clause applies to an OpenMP construct. It is not defined how the access to a variable is handled in the remaining parallel region.

```
int  n;

void f ( int i ) {
  n = i;                              // undefined, may refer to private or global variable
}
int main () {
  #pragma omp parallel private(n)
  {
    n = 1;                            // refers to private variable
    f( 2 );
  }
}
```

Construct vs. Region

The `private` clause applies to an OpenMP construct. It is not defined how the access to a variable is handled in the remaining parallel region.

```
int  n;

void f ( int i ) {
  n = i;                              // undefined, may refer to private or global variable
}
int main () {
  #pragma omp parallel private(n)
  {
    n = 1;                            // refers to private variable
    f( 2 );
  }
}
```

Access to the Original Variable

The original variable of a private copy may still be accessed in the parallel region, e.g. via pointers:

```
void f ( int i ) {
  int *  i_ptr = & i;

  #pragma omp parallel private(i)
  {
    i      = 1;                       // refers to the private variable
    *i_ptr = 2;                       // refers to the original variable
  }
}
```

Firstprivate clause

To initialise private variables with the value the variable has outside the parallel construct the clause `firstprivate` is provided:

```
#pragma omp parallel firstprivate(var1,var2,...)
```

The initialisation of each variable is performed before the parallel construct is executed.

Variables of elementary data types are initialised with standard copy assignments, whereas arrays are initialised element wise. For classes the copy constructor is used.

The same restrictions as for the `private` clause apply to the `firstprivate` clause.

Thread overhead

Spawning and finishing threads creates overhead, e.g. the operating system has to copy internal data for managing threads etc..

As an example, the following program which uses C++11 threads will take 33 seconds to run on a 2-CPU Intel Xeon E5-2640 (Hexa-Core):

```
for ( size_t i = 0; i < 100000; ++i ) {
    for ( int p = 0; p < 12; ++p ) threads[p] = std::thread( f );
    for ( int p = 0; p < 12; ++p ) threads[p].join();
}
```

In contrast to this, the equivalent OpenMP version:

```
for ( size_t  i = 0; i < 100000; ++i ) {
    #pragma omp parallel
    {
        f();
    }
}
```

only takes *0.6* seconds.

The reason for this difference is, that OpenMP will *not* spawn new threads for each parallel directive, but instead keep a *pool* of threads running all the time and only assign the corresponding tasks to these threads. Nevertheless, the actual task should have some minimal runtime for efficient OpenMP parallelisation.

Parallelizing loops

Initially, OpenMP was focused on parallelising loops.

The corresponding directive is

```
#pragma omp for [clause1 [[,] clause2, ...]]
```

and applies to the immediately following `for` loop:

```
#pragma omp for
for ( size_t i = 0; i < n; ++i ) {
  ...
}
```

Furthermore, the `for` directive must be inside the region of a `parallel` construct:

```
#pragma omp parallel
{
  #pragma omp for
  for ( size_t i = 0; i < n; ++i ) {
    ...
  }
}
```

The loop will then be split automatically into individual chunks, which are mapped to the threads of the current team, e.g. each worker thread will handle n/p indices of the `for` loop.

Remark

Common mistakes for the `for` directive are

- Forgotton `parallel` clause, which results in *sequential* execution:

```
// no "parallel" directive
{
  #pragma omp for
   for ( size_t  i = 0; i < n; ++i ) {
     ...
   }
}
```

- Forgotton `for` keyword, in which case all threads will execute the *whole* loop:

```
#pragma omp parallel
{
  #pragma omp                                  // no "for" directive
   for ( size_t  i = 0; i < n; ++i ) {
     ...
   }
}
```

Unfortunately, the compiler will not warn about such errors.

Combined Parallel Loop Construct

In many cases, the parallel loop applies to a single parallel construct. In these cases, both directives may be combined:

```
#pragma omp parallel for [clause1 [[,] clause2, ...]]
```

DEMO: omp04.cpp
DEMO: omp05.cpp

Only constant index changes are supported, e.g. `i++`, `--i` or `i += 5`.

```
#pragma omp for
for ( size_t i = 0; i < n; i = 2*i ) {              // error
  ...
}
```

Changing the index variable inside the loop body is not allowed.

```
#pragma omp for
for ( size_t i = 0; i < n; ++i ) {
  ...
  i += 2;                                           // error
  ...
}
```

The loop test must be simple, e.g. `i < n` or `i >= n`.

```
#pragma omp for
for ( size_t i = 0; i < n && i > n/2; ++i ) {      // error
  ...
}
```

Also, C++11 range-based loops are *not* supported:

```
#pragma omp for
for ( auto & v : vec ) {                          // error
  ...
}
```

Due to the implicit barrier at the end of the construct, *all* team threads must encounter the `for` directive during their execution or none at all:

```
#pragma omp parallel
{
  if ( do_loop() ) {  // conditional loop execution
    #pragma omp for
    for ( ... ) {
      ...
    }
  }
}
```

Here, `do_loop` has to return the same values for *all* threads. Otherwise, the program may block because of the barrier.

Even if the barrier is removed using `nowait`, the resulting program is non-conforming and may not work correctly.

Loop scheduling

When encountering a `for` loop:

```
#pragma omp for
for ( size_t i = 0; i < n; ++i ) {
  ...
}
```

the range of the loop index is split into *chunks* of (almost) equal size, each forming a task. These tasks are then assigned to the threads of the team.

The typical chunk size is n/p, e.g. the loop range is split as:

Loop scheduling

When encountering a `for` loop:

```
#pragma omp for
for ( size_t i = 0; i < n; ++i ) {
  ...
}
```

the range of the loop index is split into *chunks* of (almost) equal size, each forming a task. These tasks are then assigned to the threads of the team.

The typical chunk size is n/p, e.g. the loop range is split as:

This *static* scheduling can be changed by the `schedule` clause:

```
#pragma omp for schedule(static|dynamic|guided|auto|runtime)
```

For `static`, `dynamic` and `guided` an optional chunk size may be specified:

```
#pragma omp for schedule(static|dynamic|guided [, chunksize])
```

The definition of the different scheduling algorithms is:

static: Divide the loop range into chunks of size `chunksize` and assign the resulting task to the team threads in a round-robin fashion:

If no `chunksize` is specified, the default size is $\approx n/p$.

dynamic: Divide the loop range into chunks of size `chunksize`. Tasks are assigned as each thread requests them one after the other:

If no `chunksize` is specified, the default size is 1.

guided: Start by building chunks of a size $\geq$ `chunksize` proportional to the unassigned indices. Tasks are assigned as each thread requests them.

If no `chunksize` is specified, the default size is 1.

DEMO: omp05b.cpp

Further scheduling types are:

auto: Let the compiler or runtime system decide about best scheduling.

runtime: The runtime system decides about best scheduling.

In case of runtime scheduling, the user may change the default behaviour using the `OMP_SCHEDULE` environment variable, which should contain one of the above scheduling types:

```
> export OMP_SCHEDULE="static,16"   # bash
```

or

```
> setenv OMP_SCHEDULE "dynamic"     # tcsh
```

The environment variable `OMP_SCHEDULE` applies to *all* `for` directives in the program using `auto` or `runtime` scheduling, e.g. no control of single loops is possible.

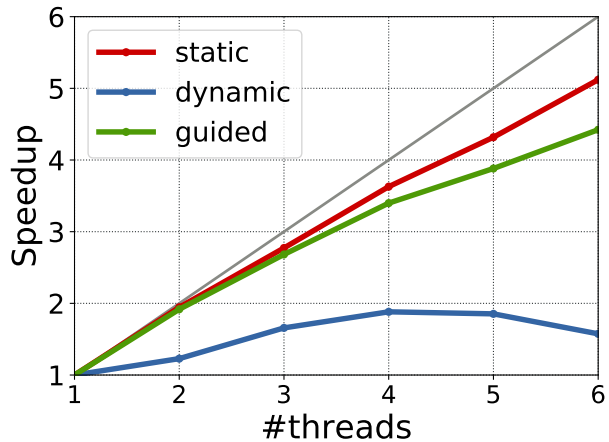
DEMO: omp06.cpp
DEMO: omp07.cpp

If the work per loop index is equal, static scheduling yields an optimal load balance between all team threads.

Matrix-Vector Multiplication

Let $A \in \mathbb{R}^{n \times n}$ and $x, y \in \mathbb{R}^n$. Compute $y = A \cdot x$:

```
void mat_vec ( const size_t n, const Matrix & A,
               const Vector & x, Vector & y ) {
  #pragma omp parallel
  #pragma omp for schedule(● ● ●)
  for ( size_t i = 0; i < n; ++i ) {
    double y_i = 0;
    for ( size_t k = 0; k < n; ++k )
      y_i += A(i,k) * x(k);
    y(i) = y_i;
  }
}
```



The performance of dynamic and guided scheduling is due to the additional overhead induced by assigning tasks to threads during runtime.

DEMO: omp06.cpp

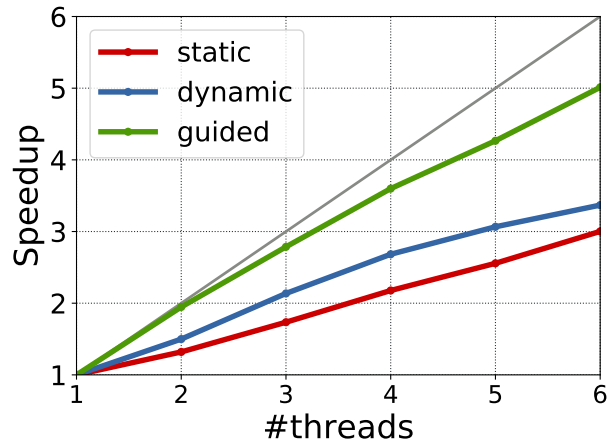
If the work per task is increased, this overhead can often be neglected, e.g. increasing the above dimension n by a factor of 4 yields the same speedup for all scheduling schemes.

The situation changes, if the work is unevenly distributed with respect to the loop index.

Triangular Matrix-Vector Multiplication

Let $A \in \mathbb{R}^{n \times n}$ be a lower triangular matrix and $x, y \in \mathbb{R}^n$. Compute $y = A \cdot x$:

```
void mat_vec ( const size_t n, const Matrix & A,
               const Vector & x, Vector & y ) {
  #pragma omp parallel
  #pragma omp for schedule(●●●)
  for ( size_t  i = 0; i < n; ++i ) {
    double  y_i = 0;
    for ( size_t  k = 0; k <= i; ++k )
      y_i += A(i,k) * x(k);
    y(i) = y_i;
  }
}
```



A static mapping leads to an unbalanced load between all threads since the load increases with the loop index.

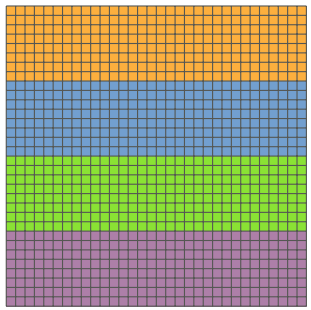
DEMO: omp07.cpp

In contrast to this, with dynamic or guided scheduling more tasks are constructed such that idling threads may request more work, yielding a more even load balance.

Remark

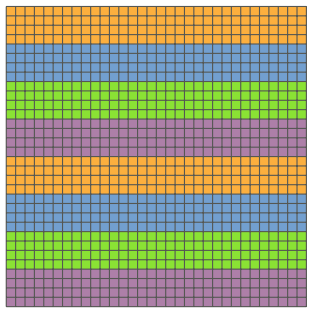
For the matrix-vector multiplication, the default static mapping corresponds to a row-wise block decomposition:

```
#pragma omp for
for ( size_t  i = 0; i < n; ++i ) {
  for ( size_t  k = 0; k < n; ++k )
    y(i) += A(i,k) * x(k);
}
```



By specifying a chunk size, this can be changed to a *cyclic* row-wise block decomposition, which would also result in a better load balance in the triangular matrix case:

```
#pragma omp for schedule(static,4)
for ( size_t  i = 0; i < n; ++i ) {
  for ( size_t  k = 0; k < n; ++k )
    y(i) += A(i,k) * x(k);
}
```

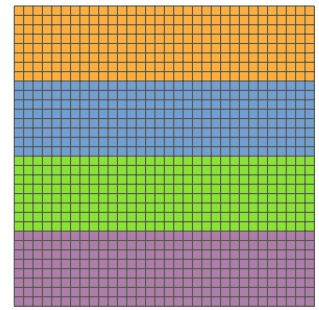


Dynamic or guided scheduling corresponds to a randomised block decomposition and is an example of *Online Scheduling*.

Remark

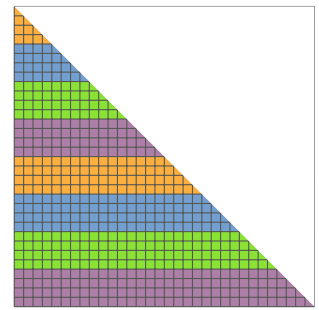
For the matrix-vector multiplication, the default static mapping corresponds to a row-wise block decomposition:

```
#pragma omp for
for ( size_t  i = 0; i < n; ++i ) {
  for ( size_t  k = 0; k < n; ++k )
    y(i) += A(i,k) * x(k);
}
```



By specifying a chunk size, this can be changed to a *cyclic* row-wise block decomposition, which would also result in a better load balance in the triangular matrix case:

```
#pragma omp for schedule(static,4)
for ( size_t  i = 0; i < n; ++i ) {
  for ( size_t  k = 0; k < n; ++k )
    y(i) += A(i,k) * x(k);
}
```



Dynamic or guided scheduling corresponds to a randomised block decomposition and is an example of *Online Scheduling*.

False sharing

If the chunk size in the `schedule` clause is too small, *false sharing* may happen:

```
std::vector< double >  x( n );

#pragma omp parallel
#pragma omp for schedule(static,1)
for ( size_t  i = 0; i < n; ++i )
  x[i] = f(i);
```

Here, each index position is cyclically scheduled to a different thread:

cache line

Hence the update of the corresponding array entries will affect the cache lines of several other threads. Depending on the runtime for each call of `f`, this may severely limit the parallel speedup:

	chunk size			
	default	1	16	128
`static`	10.09	7.89	10.01	10.33
`dynamic`	0.51	0.51	6.82	10.17
`guided`	10.23	10.23	10.28	10.36

Speedup on 2-CPU Xeon E5-2640