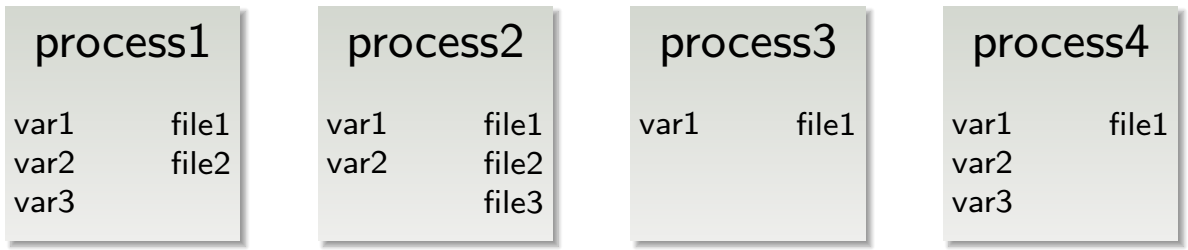
Shared-Memory Programming

Threads

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

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



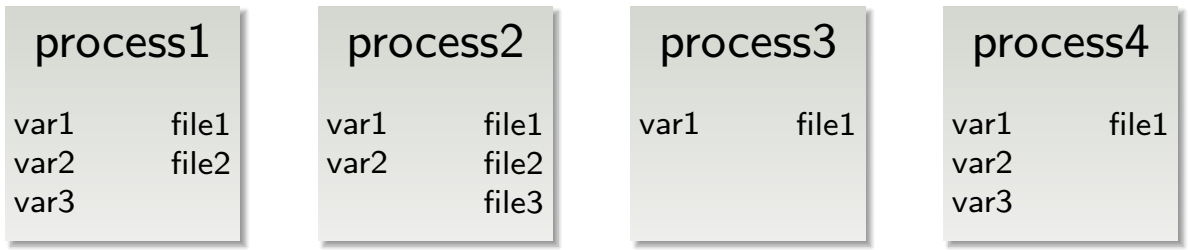
Processes may *not* directly access data in the address space of other processes.

Communication between different processes needs extra tools, e.g. via pipes, files, sockets or IPC (see Stevens 1998).

Threads

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

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

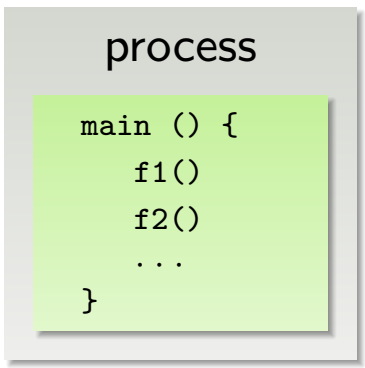


Processes may *not* directly access data in the address space of other processes.

Communication between different processes needs extra tools, e.g. via pipes, files, sockets or IPC (see Stevens 1998).

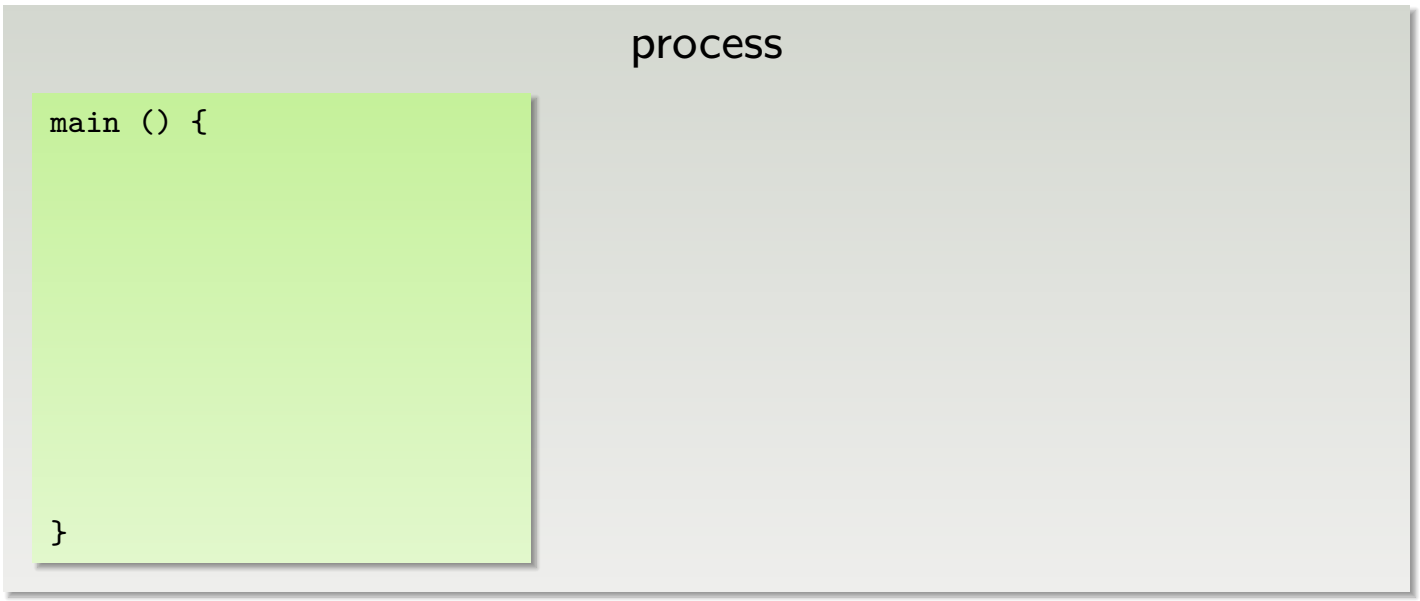
In a process execution starts with the `main` function and afterwards follows the control flow as defined by the programmer.

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



Threads

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

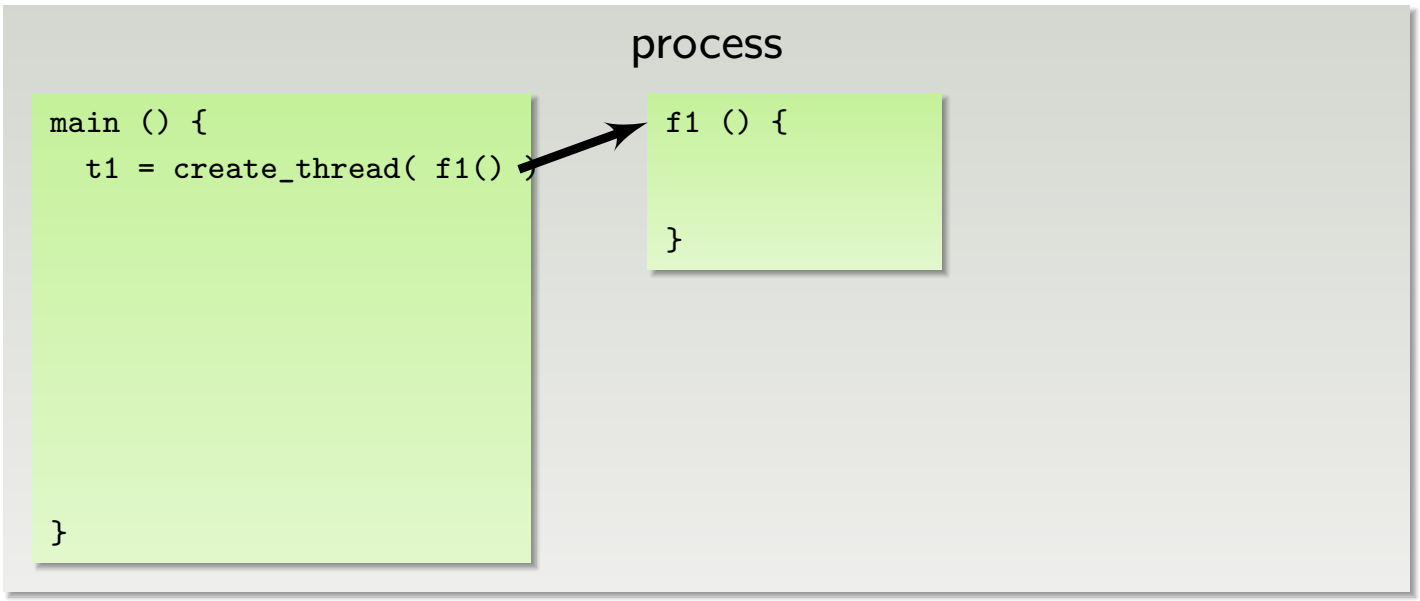


At any time during the runtime of a process, new threads may be spawned. Synchronising with the end of a thread is called *joining*.

It is also possible to start threads from within other created threads.

Threads

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

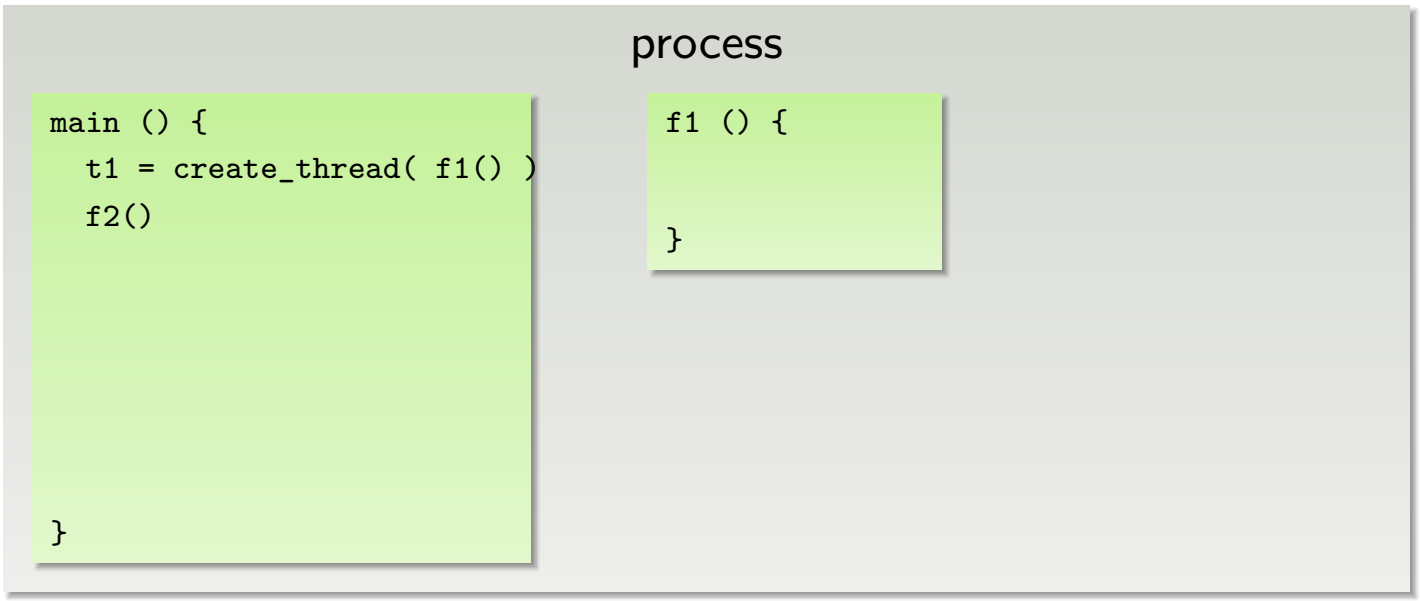


At any time during the runtime of a process, new threads may be spawned. Synchronising with the end of a thread is called *joining*.

It is also possible to start threads from within other created threads.

Threads

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

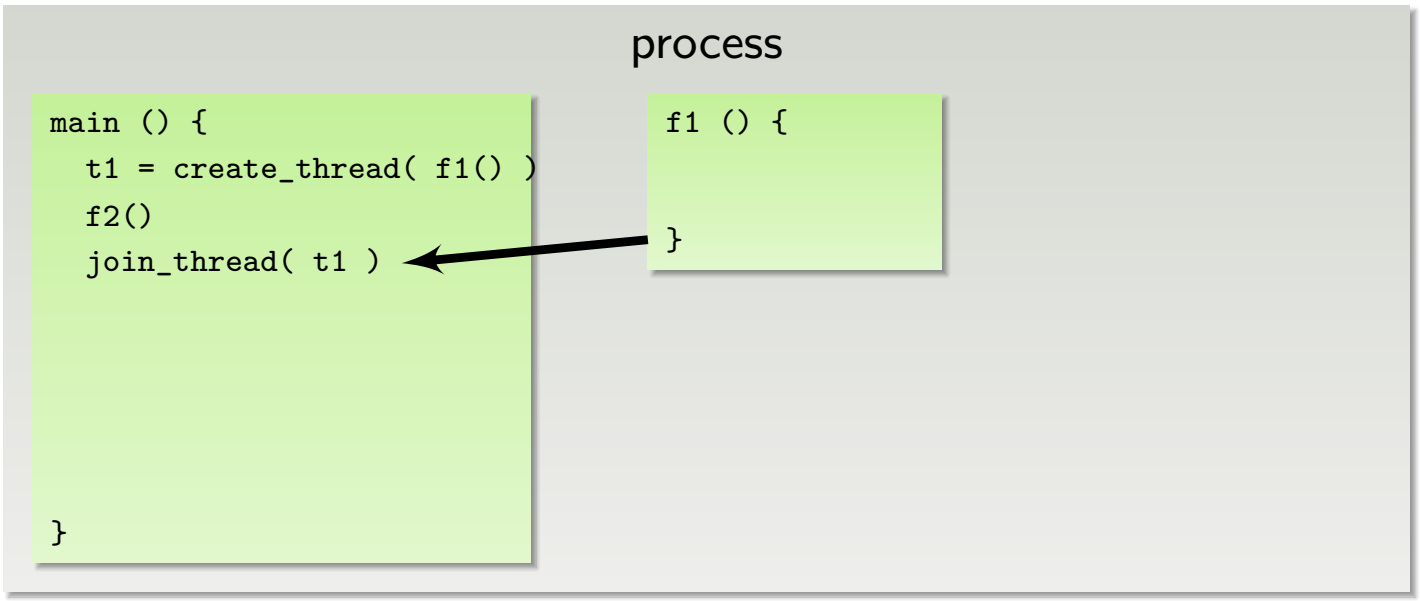


At any time during the runtime of a process, new threads may be spawned. Synchronising with the end of a thread is called *joining*.

It is also possible to start threads from within other created threads.

Threads

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

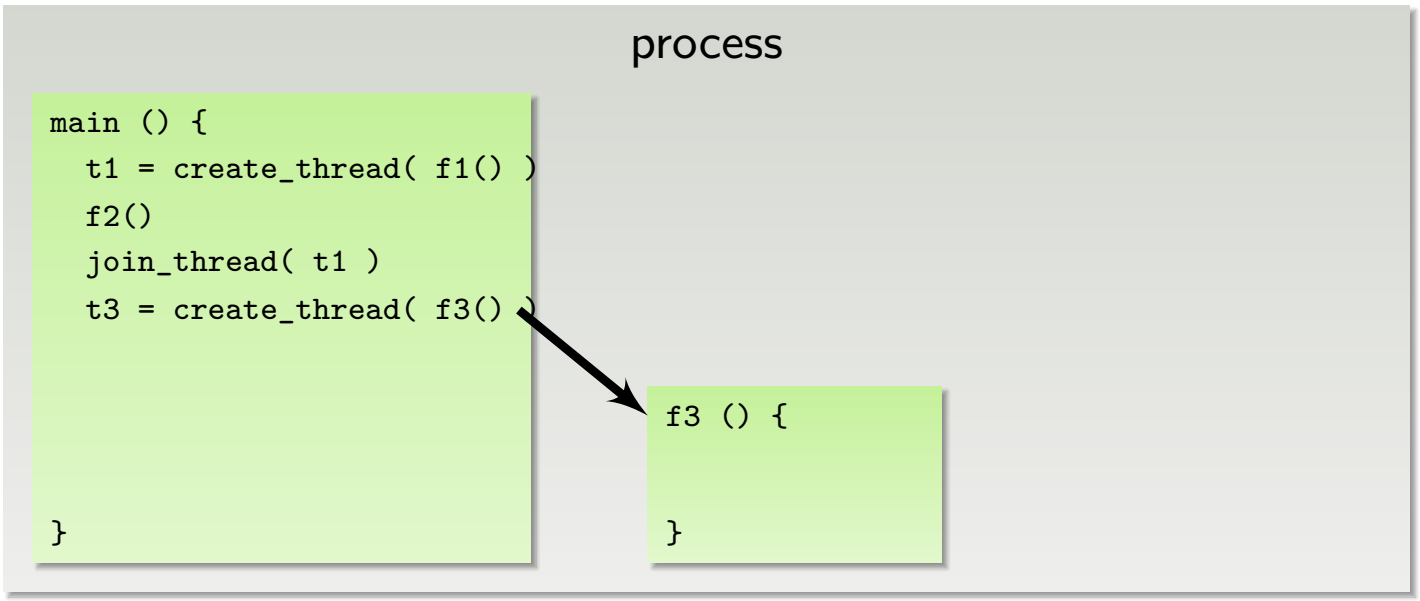


At any time during the runtime of a process, new threads may be spawned. Synchronising with the end of a thread is called *joining*.

It is also possible to start threads from within other created threads.

Threads

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

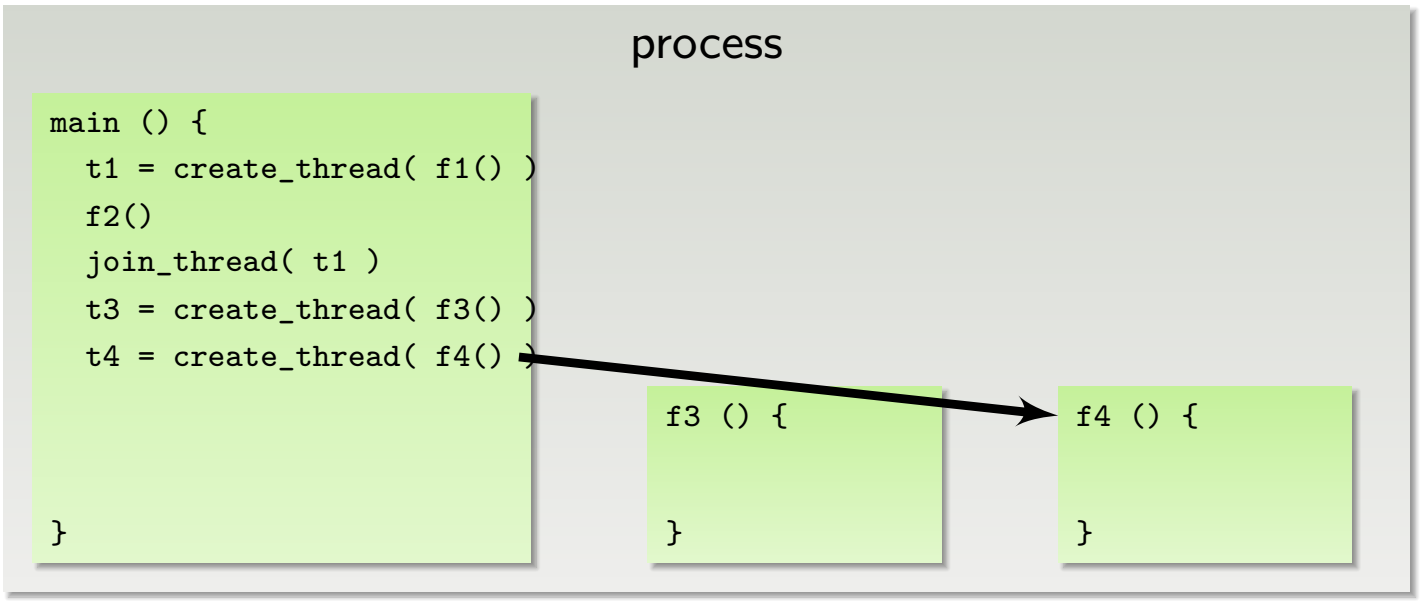


At any time during the runtime of a process, new threads may be spawned. Synchronising with the end of a thread is called *joining*.

It is also possible to start threads from within other created threads.

Threads

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

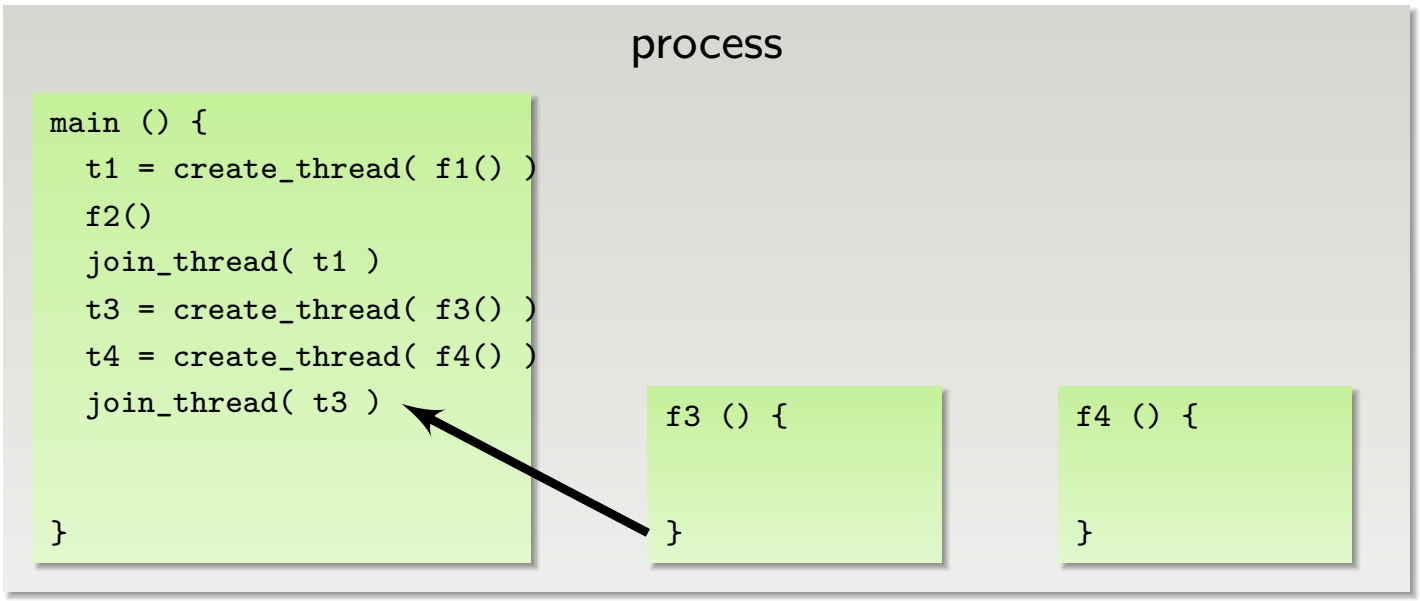


At any time during the runtime of a process, new threads may be spawned. Synchronising with the end of a thread is called *joining*.

It is also possible to start threads from within other created threads.

Threads

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

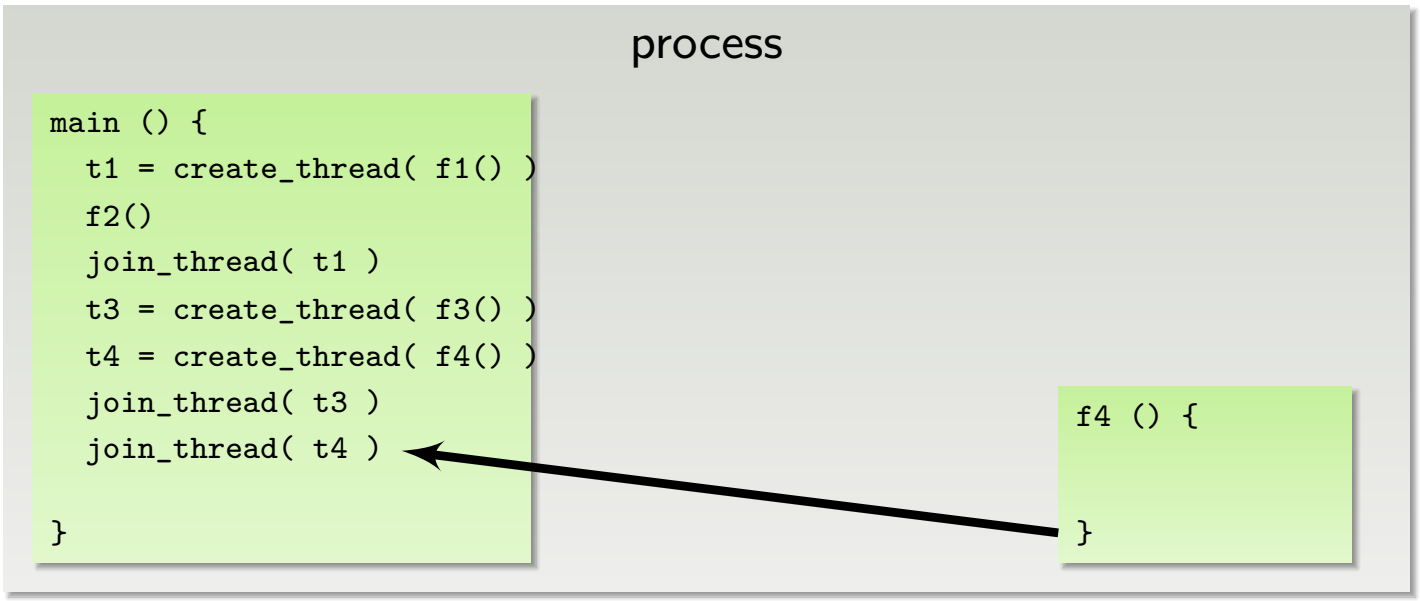


At any time during the runtime of a process, new threads may be spawned. Synchronising with the end of a thread is called *joining*.

It is also possible to start threads from within other created threads.

Threads

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*.

It is also possible to start threads from within other created threads.

Threads

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

process

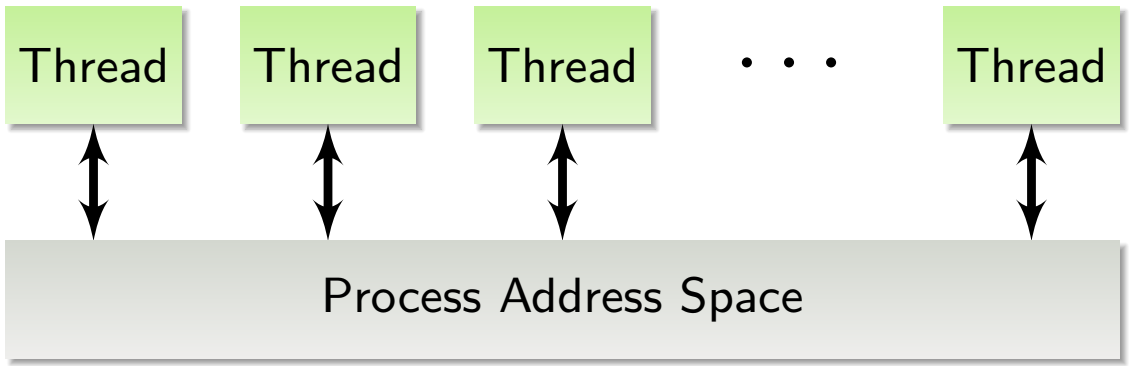
```
main () {
  t1 = create_thread( f1() )
  f2()
  join_thread( t1 )
  t3 = create_thread( f3() )
  t4 = create_thread( f4() )
  join_thread( t3 )
  join_thread( t4 )
  ...
}
```

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

It is also possible to start threads from within other created threads.

Threads

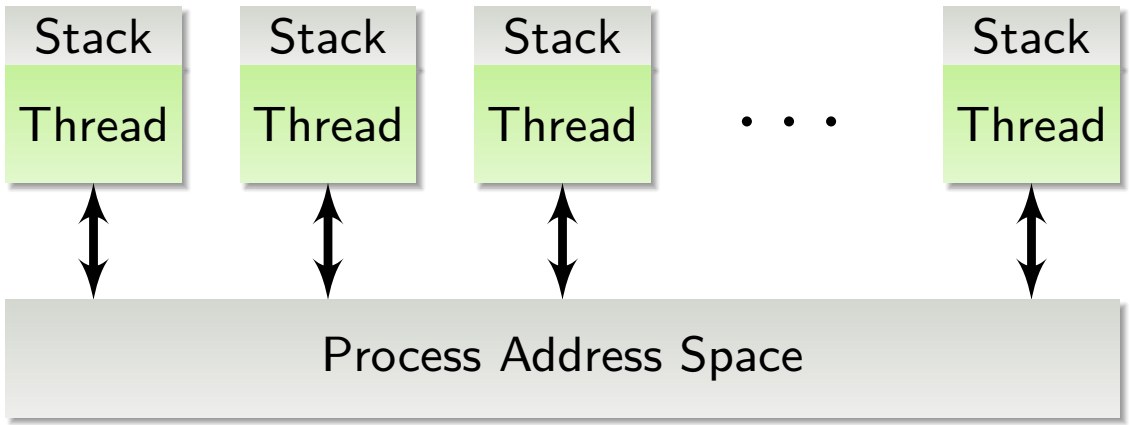
All threads have a direct access to the address space of the process.



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

Threads

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.

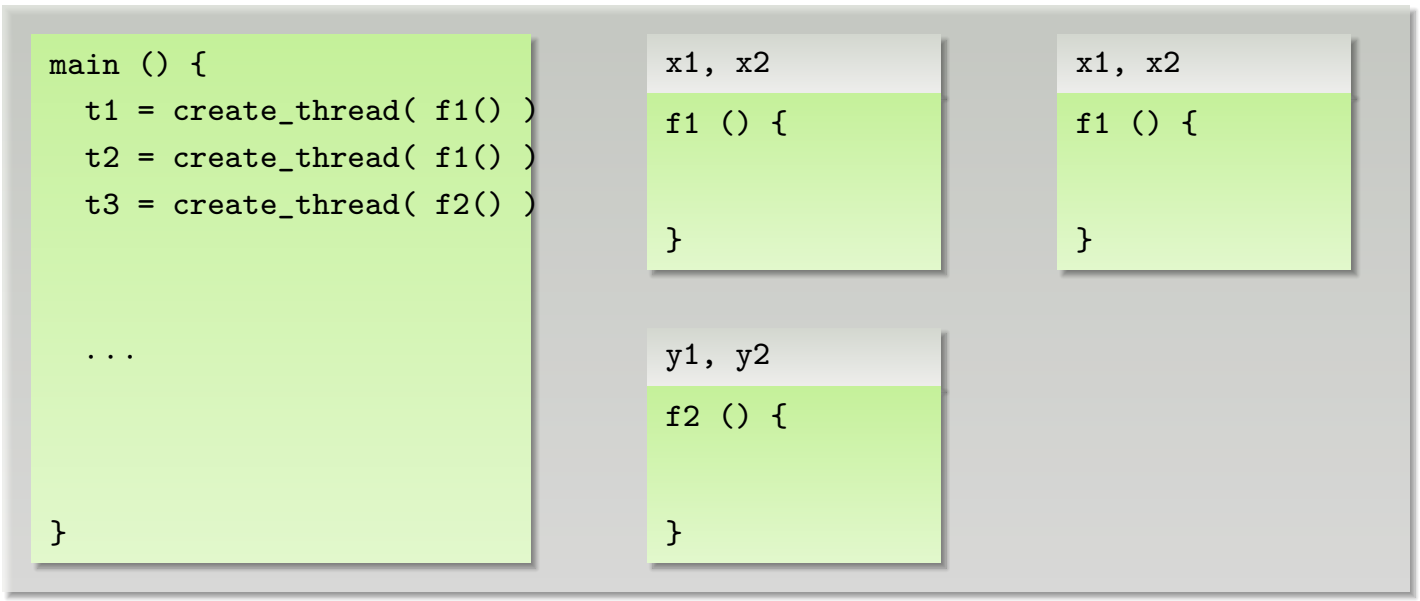
Furthermore, each thread executes a function which may have *local* variables stored on the *stack*. Such data is considered *thread-local* and must not be accessed by other threads.

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;
  ...
}
```



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

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.

Mutual Exclusion

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.

Mutual Exclusion

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.

Mutual Exclusion

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 );
}
```

Mutual Exclusion

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 );
}
```

Most mutex implementations will also provide a function `trylock()`, which either blocks an unlocked mutex or *immediately* returns with a corresponding return code if the mutex is already locked. Above all, the thread will *not* block.

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*.

Thread Scheduling

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 (see Butenhof 1997).

```
#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 );
  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.

```
#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.

Threading Building Blocks

Threading Building Blocks (TBB) is a C++ software library, which differs from the previous interfaces as it works with tasks instead of threads.

The underlying principle of the TBB framework is, that the decomposition of the computation into small but many tasks yields a better utilisation of the parallel resources than having just a few threads.

```
#include <tbb/task.h>

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

struct sum1_t : public tbb::task {
  double  t = 0;
  tbb::task * execute() {
    t = a[0] + a[1];
    return nullptr;
  }
};
struct sum2_t : public tbb::task { ... }

int main () {
  sum1_t &  t1 = *new( tbb::task::allocate_root() ) sum1_t();
  sum2_t &  t2 = *new( tbb::task::allocate_root() ) sum2_t();

  tbb::task::spawn_root_and_wait( t1 );
  tbb::task::spawn_root_and_wait( t2 );
  b = t1.t + t2.t;
}
```

TBB also implements parallel containers, e.g. vectors, various forms of mutexes and a special memory allocator.

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
- Atomic Operations

or indirectly by software, e.g.

- Thread Scheduling or
- Memory Allocation.

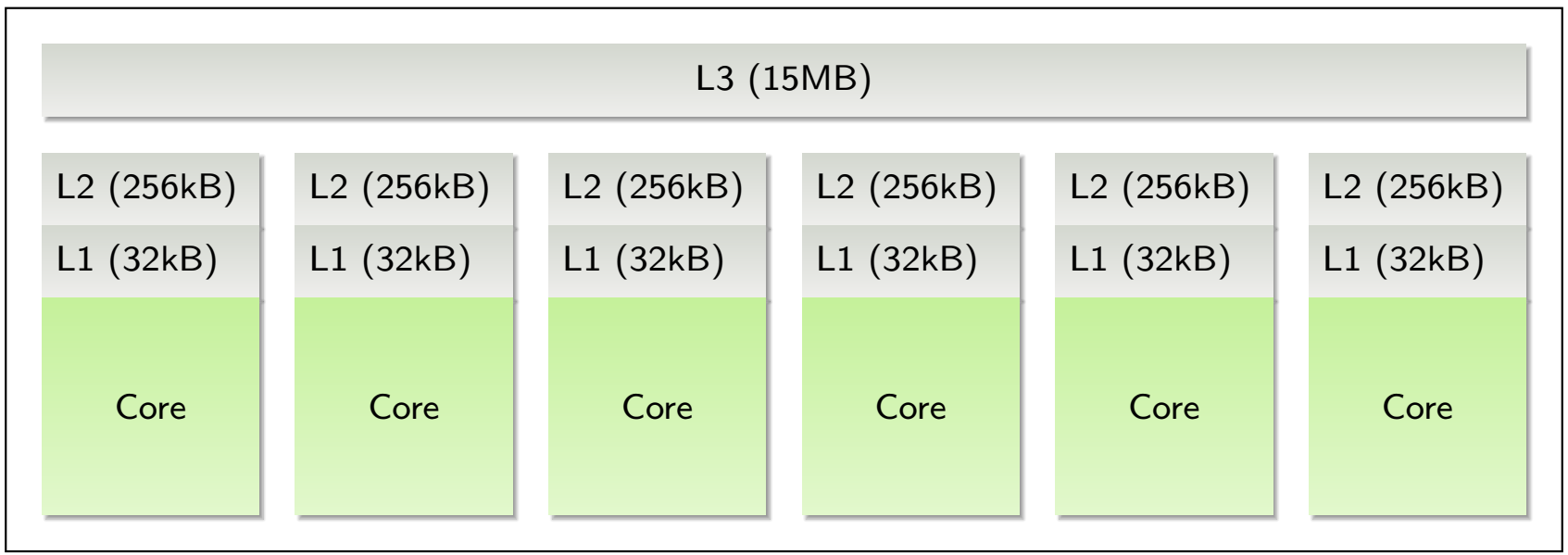
The most notable software in this context is of course the *Operating System*, providing access to the hardware.

Cache Coherence

A typical parallel computer today consists of one or more processors with several cores, each having

- first level cache (instruction and data) in the order of 16kB to 128kB,
- second level cache of size 256kB up to some MB,
- optional third level cache up to several MB capacity

The 2nd and 3rd level caches may be shared between different cores.



Cache configuration of an Intel Xeon E5-2640

Cache Coherence

Each cache may have its own copy of data from the main memory. If the data is changed in one place, all copies of that data have to be updated.

Cache coherence exists, if all caches contain the most recent version of some shared data from the main memory.

Various algorithm are used to maintain cache coherence in modern processors.

A typical cache coherence protocol will

- listen for memory accesses of all other cores,
- if a write to a locally cached memory position is detected, *invalidates* the local cache entry,
- which enforces a reload of the data from memory.

False Sharing

Processors will *not* map individual bytes into local caches, but handle memory in segments of size 64–128 bytes, so called *cache lines*.

If a single byte has changed in memory, not only the entry of this single data item is invalidated in the cache but the whole cache line, leading to a memory read of size 64–128 bytes.

Consider the following example where eight threads will compute a separate coefficient of an array.

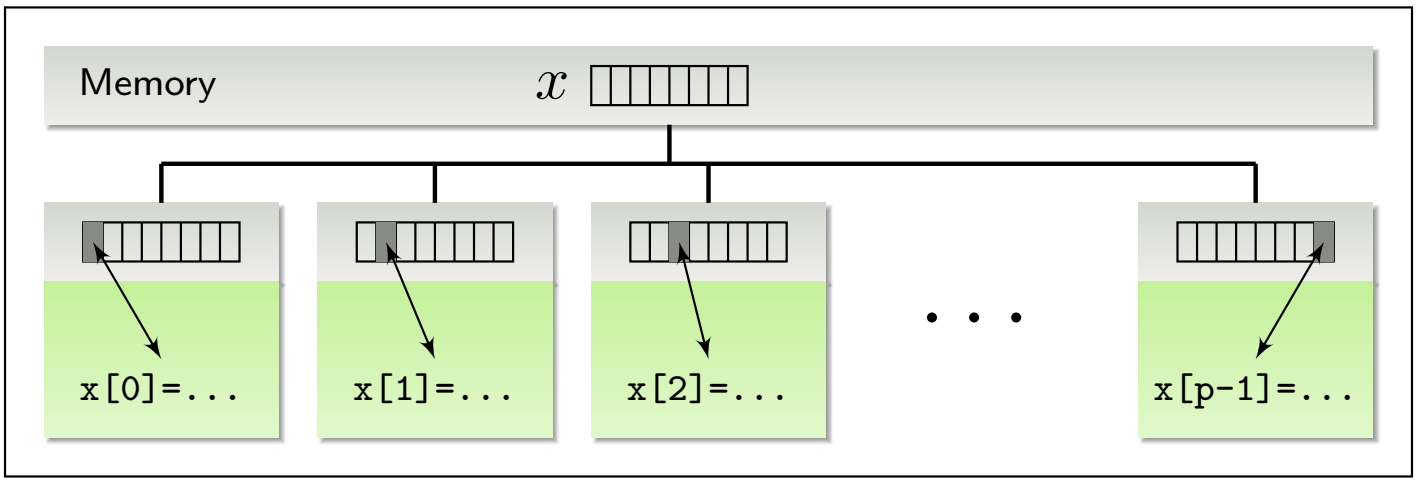
```
int main () {
  double                        x[8];
  std::vector< std::thread >    threads( 8 );

  for ( int i = 0; i < 8; ++i )
    threads[i] = std::thread( f, i, x );

  ...
}
```

```
void f ( int tid, double * x ) {
  do {
    x[tid] = compute_update();
  } while ( ... );
}
```

Since each thread will update a private entry of the array `x`, no shared data and no critical region exist.

False Sharing

Unfortunately, `x` is 64 bytes long ($8 \cdot$`sizeof(double)` $= 8 \cdot 8$), and hence, may occupy an entire cache line. Therefore, all processors will hold the complete content of x in their local caches:



If thread i updates the content of the coefficient x_i, this invalidates the cache line of x_i and therefore x in all other processors.

Hence, for each update of a coefficient of x, all processors need to load the memory corresponding to the whole array x into the local caches.

The behavious is known as *false sharing* and is an example of a hidden task interaction in multi-threaded programs.

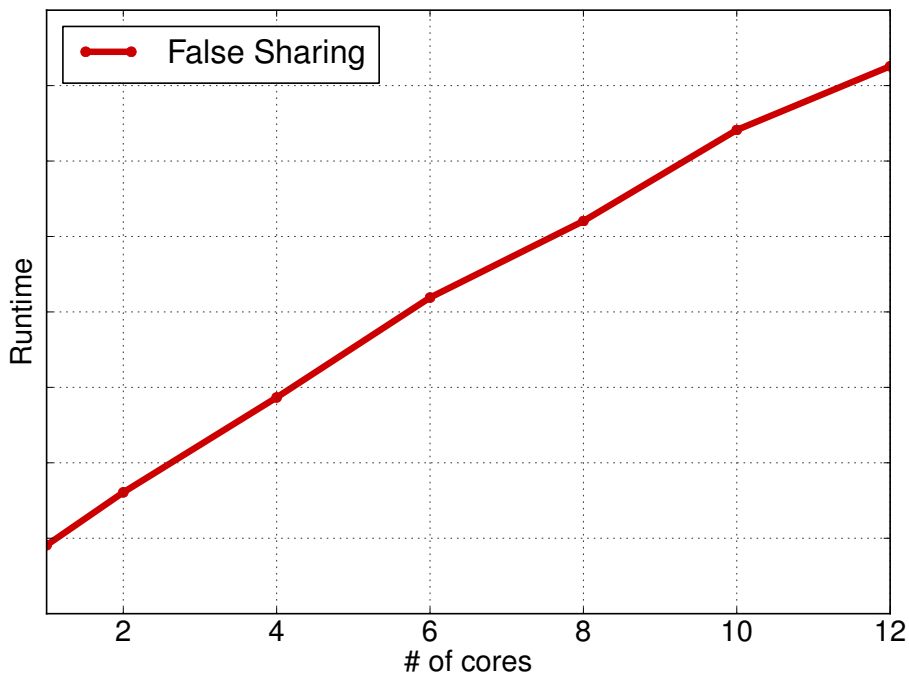
False Sharing

If the updates to x are fast enough, the memory loads will dominate the computation and severly limit the parallel efficiency. In extreme cases, the parallel runtime may be higher than the sequential runtime!

In the following example, each thread will updated its local coefficient of x via

```
void f ( int tid, double * x ) {
  for ( size_t i = 0; i < 10000000; ++i )
    x[tid] += std::sin( double(i) );
}
```

The resulting runtime grows linear with p!

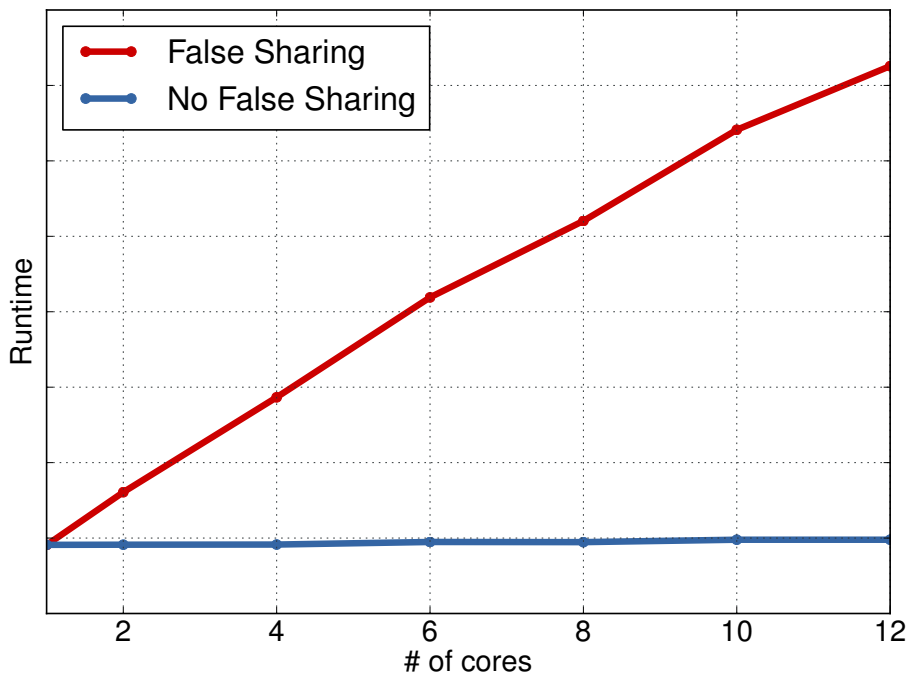


False Sharing

If instead the update is applied to a thread-private variable:

```
void f ( int tid, double * x ) {
  double  t = 0.0;

  for ( size_t i = 0; i < 10000000; ++i )
    t += std::sin( double(i) );

  x[tid] += t;
}
```

the runtime stays constant.



False Sharing

Even for a larger data set, false sharing may be an issue at the per-task data boundaries:

If the computation is *concentrated* at these boundaries, memory loads will be induced in the corresponding processor handling the neighboured task.

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.

Atomic

Using atomic instructions, the program

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

```
void f2 ( std::atomic< double > * x ) {
  *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.

Atomic

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

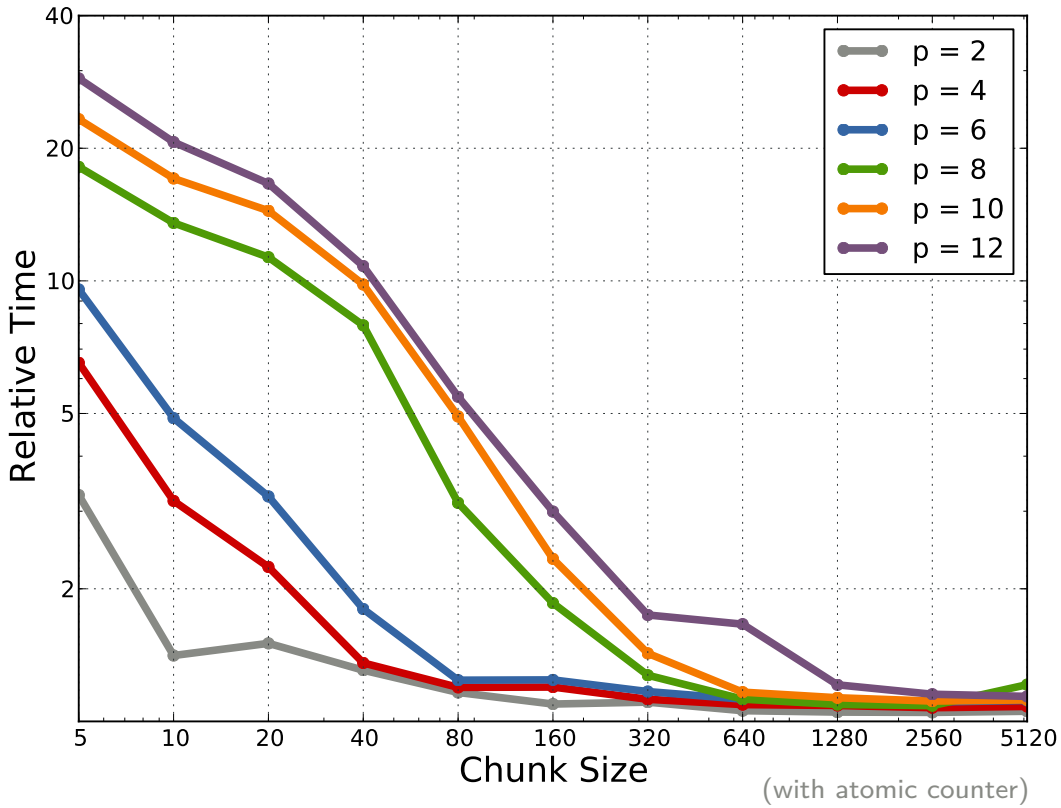
```
void worker ( std::atomic< size_t > * counter ) {
  do {
    perform_work();
    (*counter)++;
  } while ( ! finished );
}

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 );

  for ( auto &  t : threads )
    t.join();
}
```

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*.

Atomic

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.

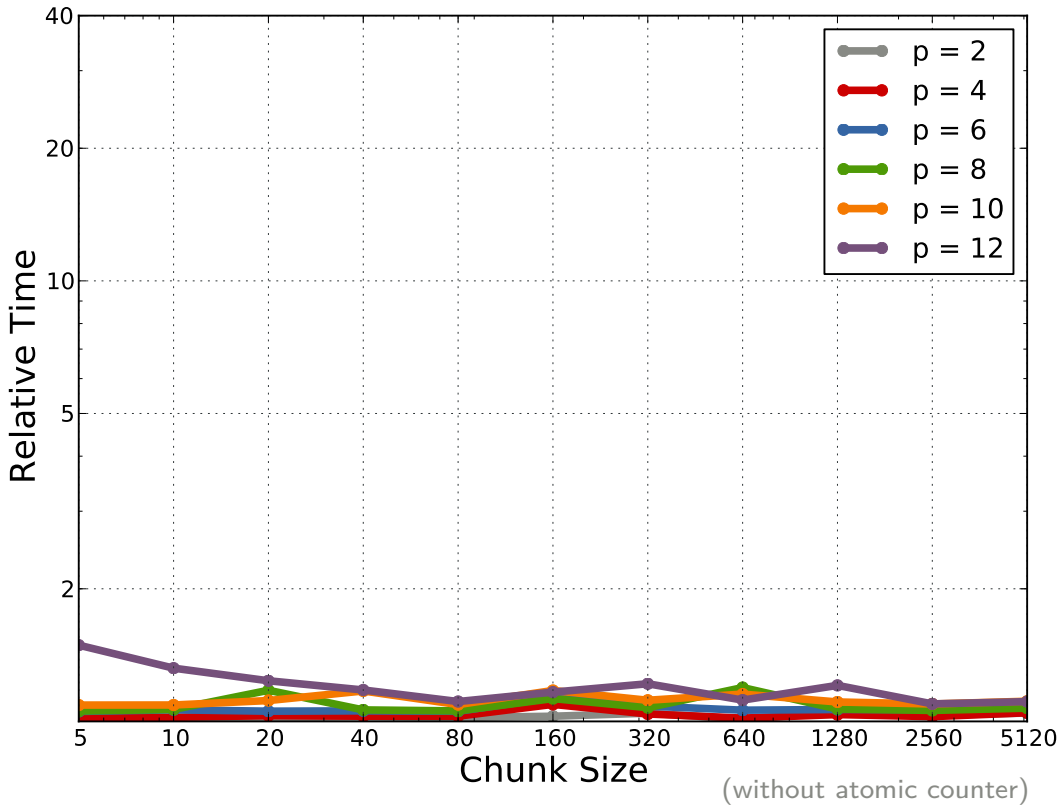


(with atomic counter)

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.

Atomic

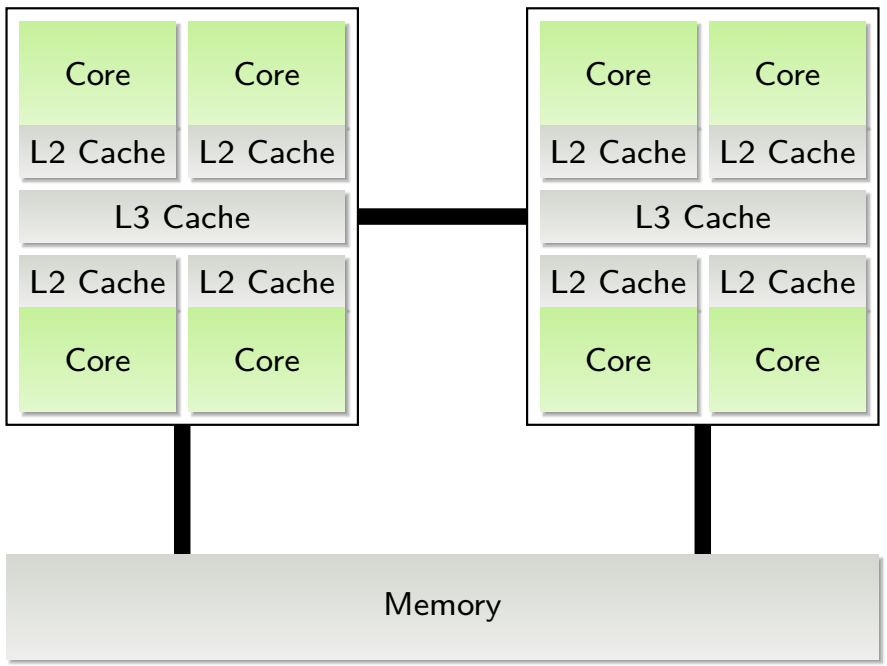
In the following example, the work per thread stays constant, only the (*chunk*) size of the work data changes. The shared atomic variable will be used to count the iterations. All times are multiples of the corresponding sequential runtime.



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.

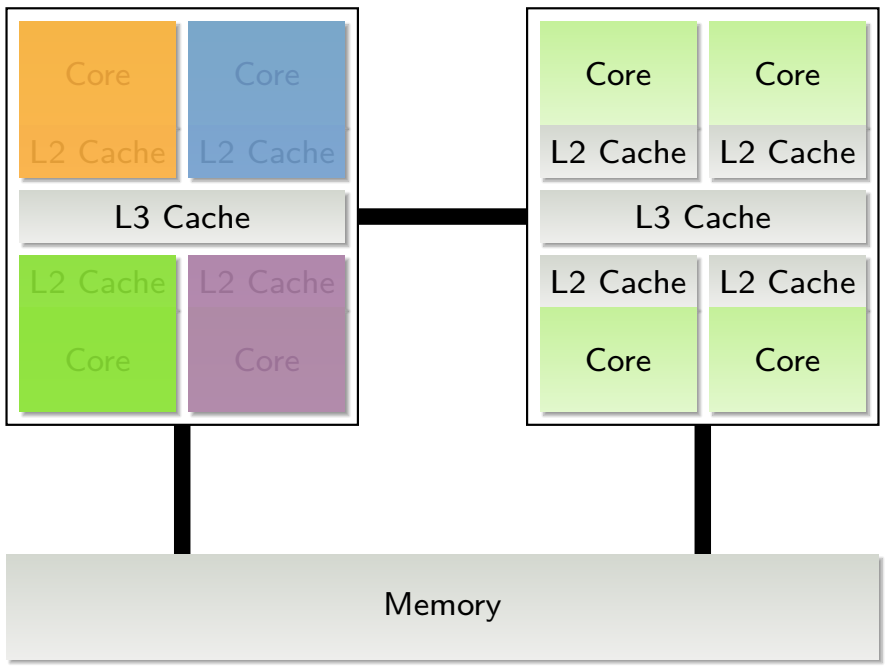
Thread Scheduling

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



Thread Scheduling

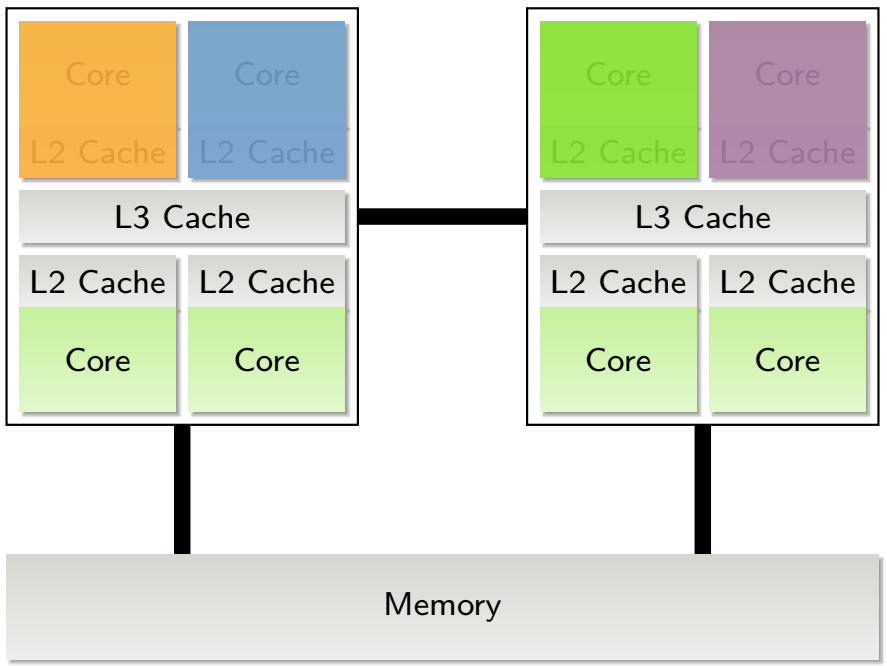
A typical hardware configuration of a compute server consists of two processors, each having several cores:



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*).

Thread Scheduling

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 per processor runtime to threads.

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

Although, knowledge of the underlying hardware structure will influence scheduling to some degree, a thread may be mapped to any processor and this mapping may vary during the runtime of the thread.

This is especially true for multiple threads of a single process.

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

Thread Scheduling

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.

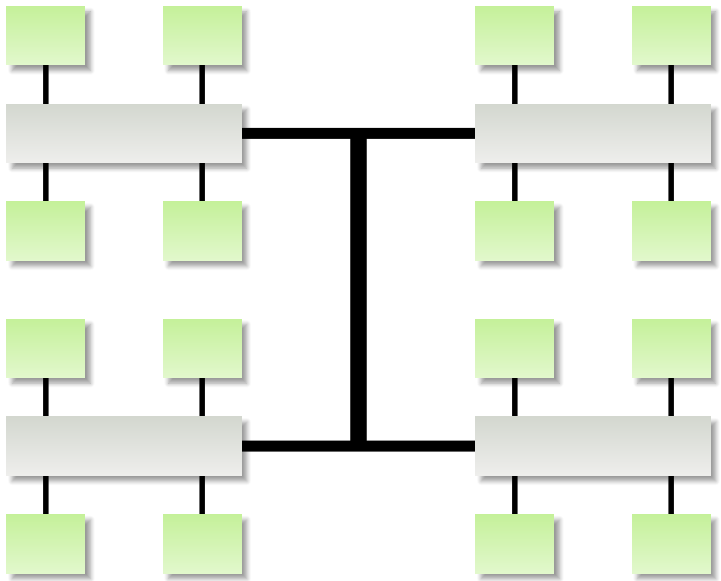
Example: Binding Thread to CPU

Under Linux, the CPU affinity mask may be controlled using the `sched_setaffinity` function. The following function will bind the calling thread to the current CPU executing the thread:

```
void bind_to_current_cpu () {
  int        cpu = sched_getcpu();  // get current CPU
  cpu_set_t  cset;

  CPU_ZERO( & cset );               // reset CPU set
  CPU_SET( cpu, & cset );           // set current CPU only
  sched_setaffinity( 0, sizeof(cset), & cset );
}
```

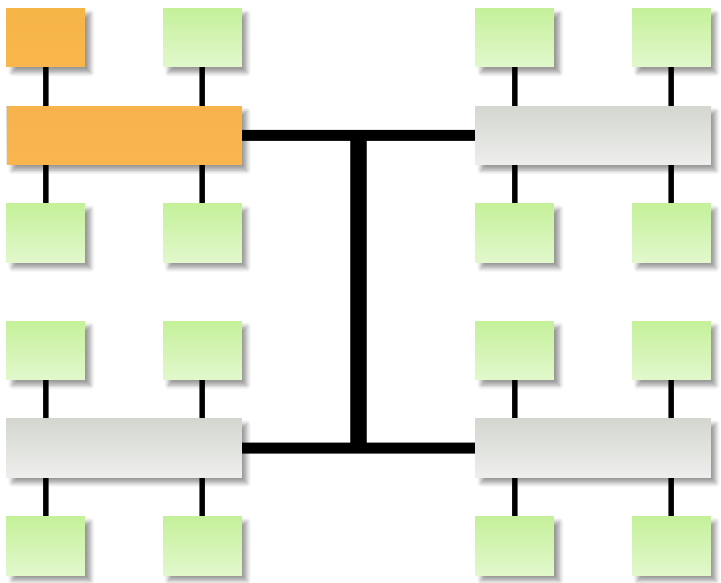
Memory Allocation

Especially on 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.



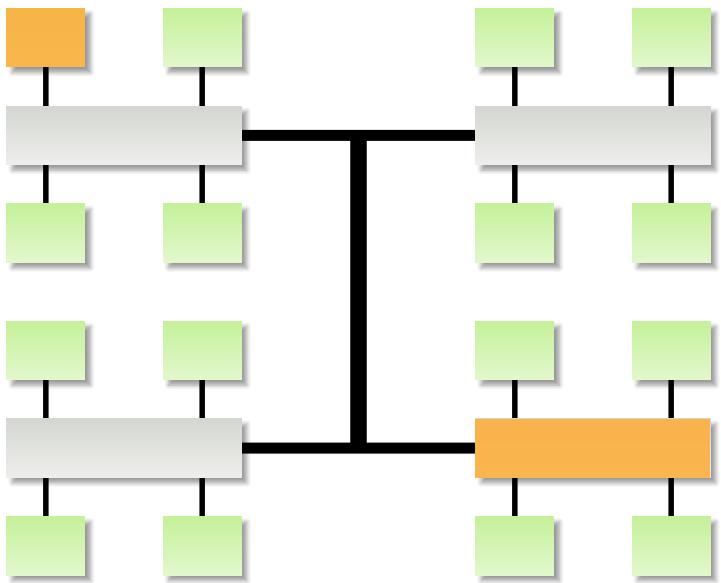
Memory Allocation

Especially on 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.



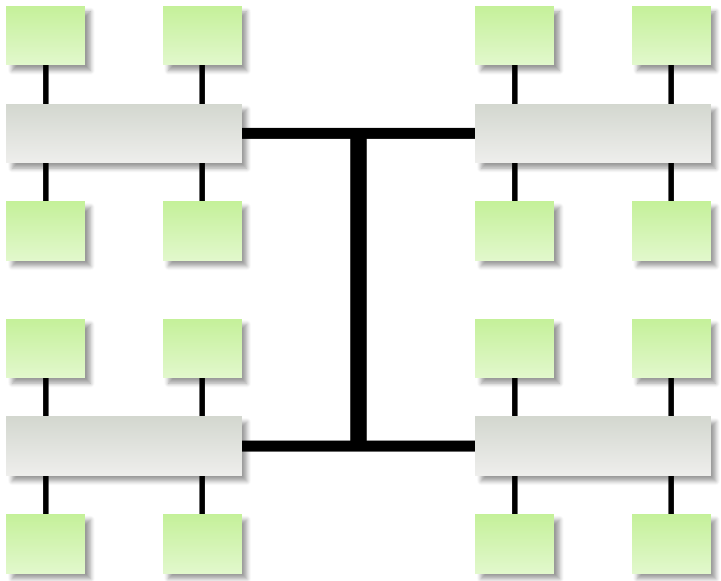
Memory Allocation

Especially on 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

Especially on 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`.

Memory Allocation

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.

Memory Allocation

Furthermore, memory allocation itself has several side-effects in multi-threaded programs, e.g.

Performance: Can the memory allocator handle several memory requests simultaneously?

Memory Consumption: How does the memory consumption grow with the number of threads?

Example: Solaris

The default `malloc` in Solaris used to have a global mutex to guard the memory allocation routine, blocking all but one thread trying to allocate memory.

However, the default `malloc` for multi-threaded application in Solaris led to a massive inrease of the memory consumption of parallel programs.

The standard memory allocator in Linux (see *PTmalloc*) handles simultaneous memory requests in parallel and has a modest increase in memory consumption when using multiple threads. Nevertheless, it may lead to non-local memory accesses.

Memory Allocation

The Linux `malloc` separates memory request from different threads by using several *heaps*, each providing their own memory management. The general algorithm is:

```
void *  malloc ( size_t n ) {
  heap_t *  work_heap = NULL;

  if ( private_heap( thread ) != NULL ) {    // try to use thread-private heap
    if ( ! is_locked( private_heap( thread ) ) ) {
      lock( private_heap( thread ) );
      work_heap = private_heap( thread );
    }
  }

  work_heap = get_and_lock_unused_heap();    // try to use unused heap of another thread

  if ( work_heap == NULL ) {
    work_heap = allocate_heap();             // allocate new heap
    lock( work_heap );
  }

  void * p = allocate_from( work_heap, n );

  set_private_heap( thread, work_heap );

  return p;
}
```

This algorithm creates $p' \leq p$ heaps, where p' is the maximal number of concurrent memory requests and p the number of threads, respectively.

Memory Allocation

By design, several threads may share the same heap and hence, memory requests may not always be served from local memory.

Furthermore, heaps may be moved between threads, e.g. if the private heap is in use by another thread, a currently unused heap from another thread is used.

Memory Allocation

By design, several threads may share the same heap and hence, memory requests may not always be served from local memory.

Furthermore, heaps may be moved between threads, e.g. if the private heap is in use by another thread, a currently unused heap from another thread is used.

Alternative: TBB Malloc

An alternative memory allocator is provided by the Threading Building Blocks. There, heaps and threads are tightly coupled without any sharing. Furthermore, the TBB malloc has also several optimisations for fast multi-threaded memory allocation.

To use TBB malloc, just link with the corresponding library:

```
> icpc -tbb -O2 main.cc -ltbbmalloc
```

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.

Literature

Butenhof, D.R. (1997). *Programming with POSIX Threads*. Addison Wesley.

Gloger, W. "*PTmalloc*". http://www.malloc.de/.

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