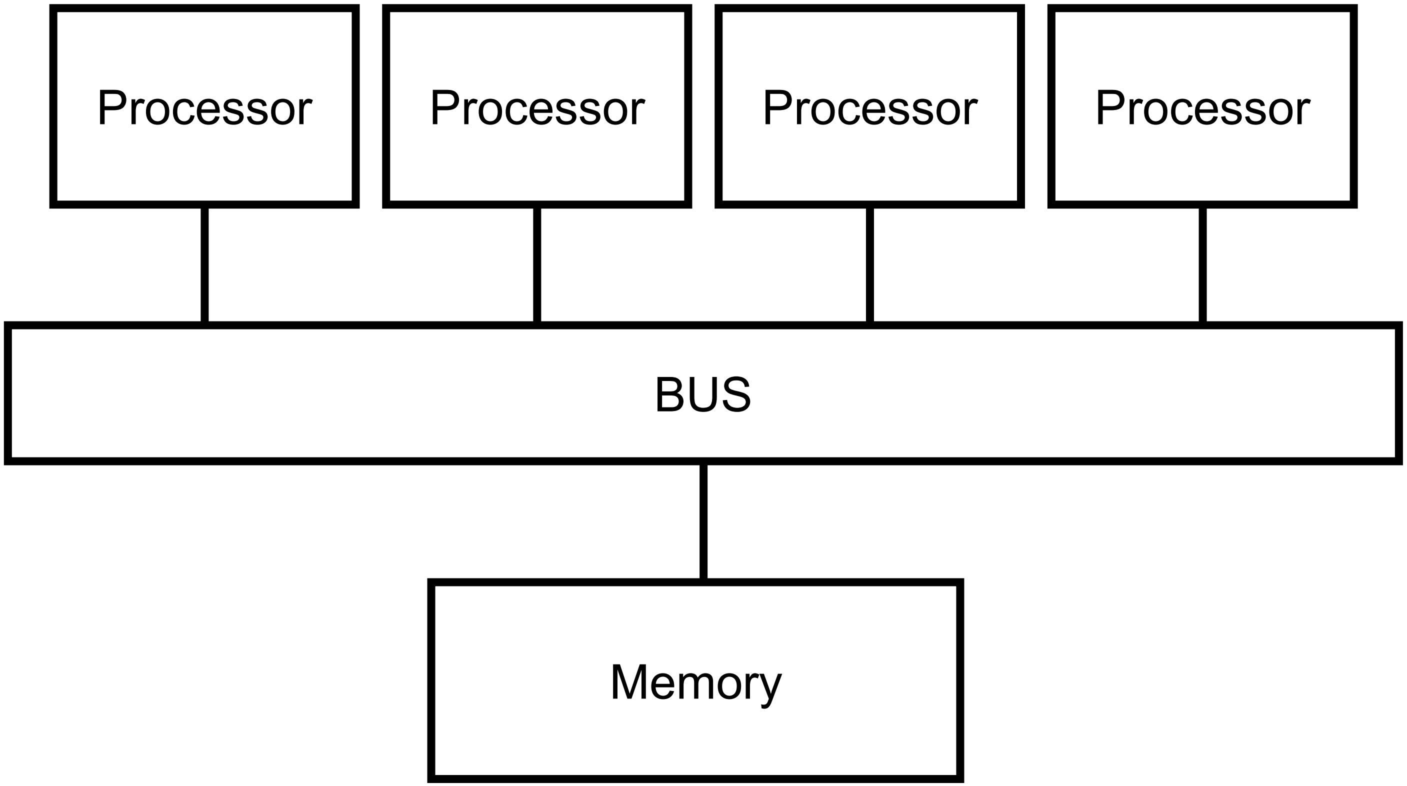
OpenMP

Shared Memory Architektur



OpenMP

- Portable programming of shared memory systems.
- It is a quasi-standard.
- OpenMP-Forum
- 1997 - ...
- API for Fortran and C/C++
  - directives
  - runtime routines
  - environment variables
- www.openmp.org

Example

Program

```
#include <omp.h>

main(){
    #pragma omp parallel
    {
     printf("Hello world");
    }
}
```

Compilation

```
> icc –O3 –openmp openmp.c
```

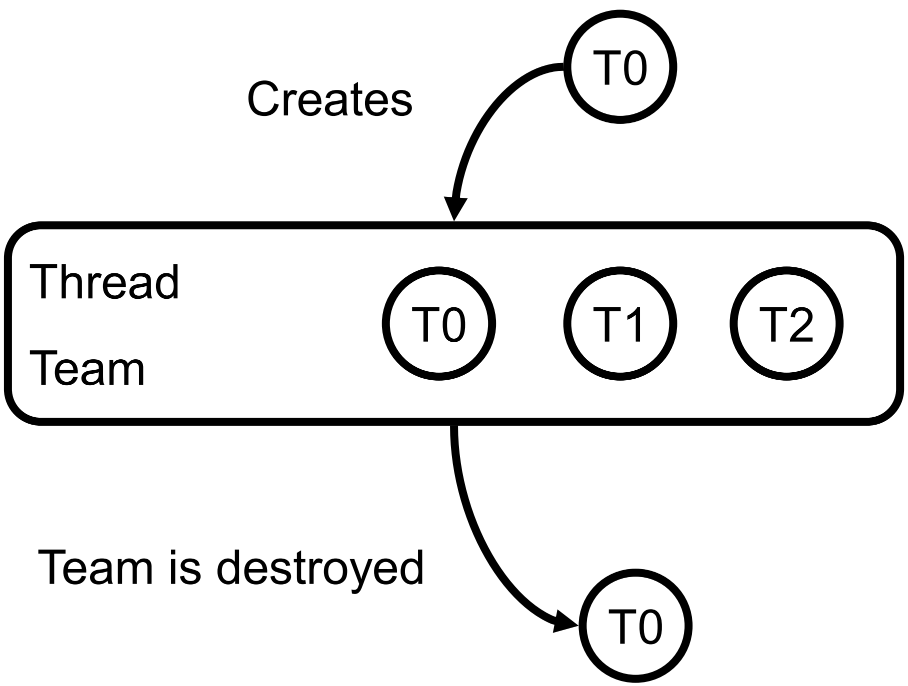
Execution

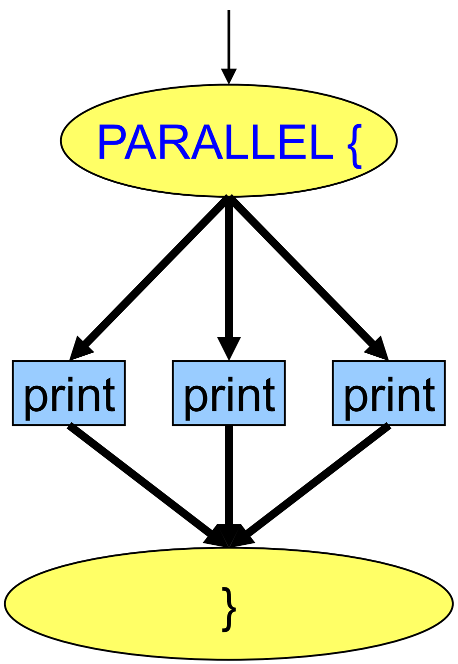
```
> export OMP_NUM_THREADS=2
> a.out
Hello world
Hello world
```

```
> export OMP_NUM_THREADS=3
> a.out
Hello World
Hello World
Hello World
```

Execution Model

```
#pragma omp parallel
{
 printf("Hello world %d\n", omp_get_thread_num());
}
```





Fork/Join Execution Model

1. An OpenMP-program starts as a single thread (*master thread*).
2. Additional threads (*Team*) are created when the master hits a parallel region.
3. When all threads finished the parallel region, the new threads are given back to the runtime or operating system.

- A team consists of a fixed set of threads executing the parallel region redundantly.
- All threads in the team are synchronized at the end of a parallel region via a barrier.
- The master continues after the parallel region.

Work Sharing in a Parallel Region

```
main (){
int a[100];
#pragma omp parallel
 {
   #pragma omp for
    for (int i= 1; i<n;i++)
        a(i) = i;
   …
 }
}
```

Shared and Private Data

- Shared data are accessible by all threads. A reference a[5] to a shared array accesses the same address in all threads.
- Private data are accessible only by a single thread. Each thread has its own copy.
- The default is shared.

Private clause for parallel loop

```
main (){
int a[100], t;

#pragma omp parallel
 {
   #pragma omp for private(t)
    for (int i= 1; i<n;i++){
        t=f(i);
        a(i)=t;
      }
 }
}
```

Private Data

- A new copy is created for each thread.
- One thread may reuse the global shared copy.
- The private copies are destroyed after the parallel region.
- The value of the shared copy is undefined.

Parallel Region

- The statements enclosed lexically within a region define the lexical extent of the region.
- The dynamic extent further includes the routines called from within the construct.

```
#pragma omp parallel [parameters]
{
  parallel region
}
```

Work-Sharing Constructs

- Work-sharing constructs distribute the specified work to all threads within the current team.

- Types
  - Parallel loop
  - Parallel section
  - Master region
  - Single region
  - General work-sharing construct (only Fortran)

Parallel Loop

```
#pragma omp for [parameters]
     for ...
```

- The iterations of the do-loop are distributed to the threads.
- The scheduling of loop iterations is determined by one of the scheduling strategies *static*, *dynamic*, *guided*, and *runtime*.
- There is no synchronization at the beginning.
- All threads of the team synchronize at an implicit barrier if the parameter *nowait* is not specified.
- The loop variable is by default private. It must not be modified in the loop body.
- The expressions in the for-statement are very restricted.

Scheduling Strategies

- Schedule clause

  schedule (type [,size])

- Scheduling types:
  - static: Chunks of the specified size are assigned in a round-robin fashion to the threads.
  - dynamic: The iterations are broken into chunks of the specified size. When a thread finishes the execution of a chunk, the next chunk is assigned to that thread.
  - guided: Similar to dynamic, but the size of the chunks is exponentially decreasing. The size parameter specifies the smallest chunk. The initial chunk is implementation dependent.
  - runtime: The scheduling type and the chunk size is determined via environment variables.

Example: Dynamic Scheduling

```
main(){
int i, a[1000];

#pragma omp parallel
 {
  #pragma omp for schedule(dynamic, 4)
  for (int i=0; i<1000;i++)
    a[i] = omp_get_thread_num();

  #pragma omp for schedule(guided)
  for (int i=0; i<1000;i++)
          a[i] = omp_get_thread_num();

 }
}
```

Reductions

```
reduction(operator: list)
```

- This clause performs a reduction on the variables that appear in *list*, with the operator *operator*.
- Variables must be shared scalars
- *operator* is one of the following:
  - +, *, -, &, ^, |, &&, ||
- Reduction variable might only appear in statements with the following form:
  - *x = x operator expr*
  - *x binop= expr*
  - *x++, ++x, x--, --x*

Example: Reduction

```
#pragma omp parallel for reduction(+: a)
for (i=0; i<n; i++) {
  a = a + b[i];

}
```

Classification of Variables

- private(var-list)
  - Variables in var-list are private.
- shared(var-list)
  - Variables in var-list are shared.

Parallel Section

```
#pragma omp sections [parameters]
{
 [#pragma omp section]
      block
 [#pragma omp section
      block ]
}
```

- Each section of a parallel section is executed once by one thread of the team.
- Threads that finished their section wait at the implicit barrier at the end of the section construct.

Example: Parallel Section

```
main(){
int i, a[1000], b[1000]

#pragma omp parallel private(i)
 {
 #pragma omp sections
  {
  #pragma omp section
  for (int i=0; i<1000; i++)
    a[i] = 100;
  #pragma omp section
  for (int i=0; i<1000; i++)
    b[i] = 200;
  }

 }
}
```

Master / Single Region

```
#pragma omp master
        block

#pragma omp single [parameters]
  block
```

- A master or single region enforces that only a single thread executes the enclosed code within a parallel region.
- Common
  - No synchronization at the beginning of region.
- Different
  - Master region is executed by master thread while the single region can be executed by any thread.
  - Master region is skipped by other threads while all threads are synchronized at the end of a single region.

Combined Work-Sharing and Parallel Constructs

- #pragma omp parallel for
- #pragma omp parallel sections
- !$OMP PARALLEL WORKSHARE

Barrier

```
#pragma omp barrier
```

- The barrier synchronizes all the threads in a team.
- When encountered, each thread waits until all of the other threads in that team have reached this point.

Critical Section

```
#pragma omp critical [(Name)]
{ ... }
```

- Mutual exclusion
  - A critical section is a block of code that can be executed by only one thread at a time.
- Critical section name
  - A thread waits at the beginning of a critical section until no other thread is executing a critical section with the same name.
  - All unnamed critical directives map to the same name.
  - Critical section names are global entities of the program. If a name conflicts with any other entity, the behavior of the program is unspecified.

Simple Locks

- Locks can be hold by only one thread at a time.
- A lock is represented by a lock variable of type omp_lock_t.
- The thread that obtained a simple lock cannot set it again.

- Operations
  - omp_init_lock(&lockvar): initialize a lock
  - omp_destroy_lock(&lockvar): destroy a lock
  - omp_set_lock(&lockvar): set lock
  - omp_unset_lock(&lockvar): free lock
  - logicalvar = omp_test_lock(&lockvar): check lock and possibly set lock, returns true if lock was set by the executing thread.

Explicit Tasking

- Explicit creation of tasks

```
#pragma omp parallel
{
  #pragma omp single {
  for ( elem = l->first; elem; elem = elem->next)
     #pragma omp task
         process(elem)
  }
// all tasks are complete by this point
}
```

- Task scheduling
  - Tasks can be executed by any thread in the team
- Barrier
  - All tasks created in the parallel region have to be finished.

Summary

- OpenMP is quasi-standard for shared memory programming
- Based on Fork-Join Model
- Parallel region and work sharing constructs
  - Declaration of private or shared variables
  - Reduction variables
  - Scheduling strategies
- Synchronization via Barrier, Critical section, Atomic, locks, nestable locks