Programming Techniques for Supercomputers: Shared-memory parallel processing with OpenMP

Getting Started

Data Scoping

Worksharing

Prof. Dr. G. Wellein[a,b] , Dr. G. Hager[a]
[a] Erlangen National High Performance Computing Center (NHR@FAU)
[b] Department für Informatik
Friedrich-Alexander-Universität Erlangen-Nürnberg, Sommersemester 2024

NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

Shared-memory parallel processing with OpenMP

Getting Started
Data Scoping
Worksharing

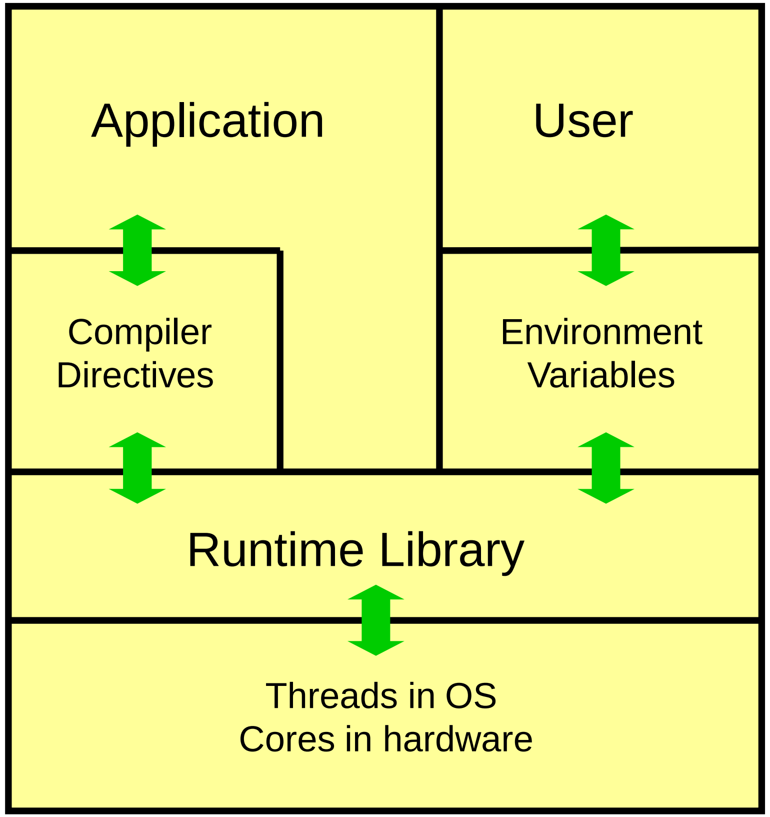
Introduction to OpenMP: Basics

- "Easy," incremental and portable parallel programming of shared-memory computers: OpenMP
- Original design goal: Data-level shared memory parallelism – many extensions: Task parallelism, Accelerator offloading, SIMD support,…

- Standardized set of compiler directives & library functions: **http://www.openmp.org/**
  - FORTRAN, C and C++ interfaces are defined
  - Supported by all current compilers
  - Free tools are available

- B. Chapman, G. Jost, R. v. d. Pas: Using OpenMP. MIT Press, 2007, ISBN 978-0262533027
- R. v. d. Pas, E. Stotzer, C. Terboven: Using OpenMP – The Next Step. MIT Press, 2017, ISBN 978-0-262-53478-9

Introduction to OpenMP: Software Architecture

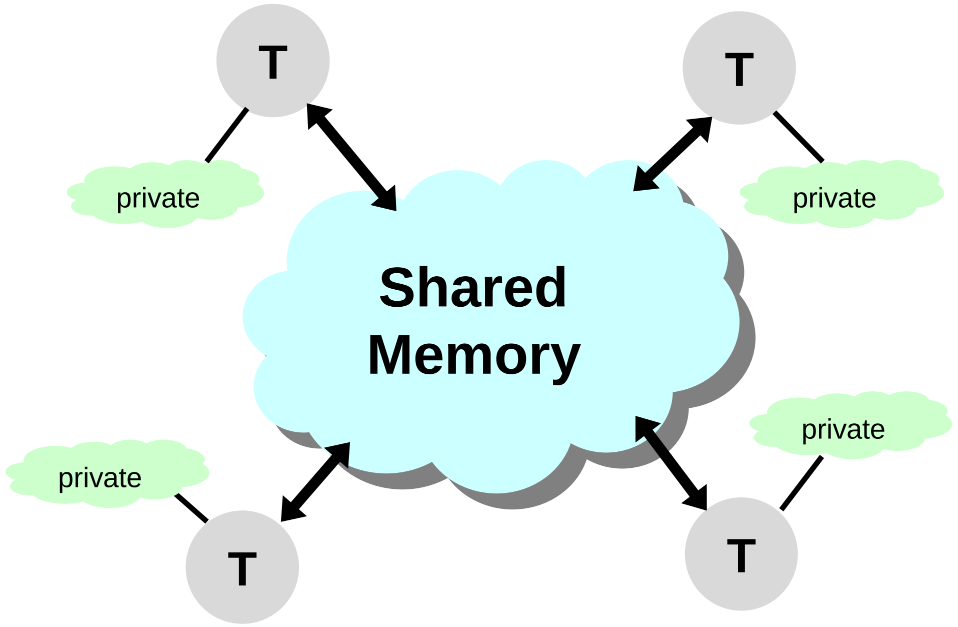


- Programmer’s view:
  - **Directives/pragmas** in application code
  - (A few) library routines
- User’s view (code execution):
  - **Environment variables** determine:
    - resource allocation
    - scheduling strategies and other (implementation-dependent) behavior
- Operating system view:
  - Parallel work done by OS **threads**

Introduction to OpenMP: shared-memory model

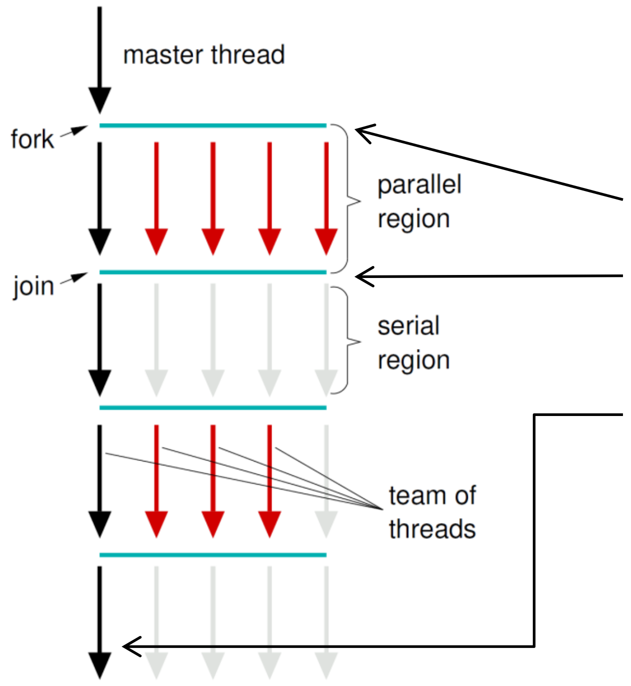
Central concept of OpenMP programming:

Threads



- Threads:
  - Spawned by a process
  - Local register set, instruction pointer, stack
  - Shared global address space
- Data scope: shared or private
  - shared data available to all threads
  - private data only available to thread that owns it
- Data transfer between threads:
  - transparent to programmer

Introduction to OpenMP: fork-join execution model



Thread # 0 1 2 3 4

Program start:
one process (master thread) running

Parallel region: team of threads is generated (“fork”)

Synchronize when leaving parallel region (“join”)

Serial region:
only master executes

Introduction to OpenMP: General syntax in C/C++

- Compiler directive:

```
#pragma omp [directive [clause ...]]
   structured block
```

  - If OpenMP is not enabled by compiler → treated like comment

- Include file for API calls: `#include <omp.h>`
- Conditional compilation: Compiler's OpenMP switch sets preprocessor macro (acts like `-D_OPENMP`)

```
#ifdef _OPENMP
   t = omp_get_thread_num();
#endif
```

Introduction to OpenMP: General syntax in Fortran

- Each directive starts with sentinel in column 1:
  - fixed source: `!$OMP` or `C$OMP` or `*$OMP`
  - free source: `!$OMP`

  followed by a directive and, optionally, clauses.

- API calls:
  - F77: include file `omp_lib.h`, F90+: module `omp_lib`
  - Conditional compilation of lines starting with `!$` or `C$` or `*$` to ensure compatibility with sequential execution

- Example:

```
   myid = 0
!$ myid = omp_get_thread_num()
   numthreads = 1
!$ numthreads = omp_get_num_threads()
```

Introduction to OpenMP: parallel region

- **`#pragma omp parallel`**

  `structured block`

  - Makes structured block a parallel region: All code executed between start and end of this region is executed by all threads
  - This includes subroutine calls within the region

```
#pragma omp parallel
  printf("Hello from %d of %d\n",
            omp_get_thread_num(), omp_get_num_threads());
```

API functions →

ID of calling thread 0…n-1

# of threads in region

- **`END PARALLEL`** required in Fortran

Introduction to OpenMP: compile and run

- Activate OpenMP directives
  - Intel: **`-qopenmp`**, GCC: **`-fopenmp`**
- Number of threads: Shell variable **`OMP_NUM_THREADS`**

```
$ icc -qopenmp hello.c
$ OMP_NUM_THREADS=4 ./a.out
 Hello from  0 of  4
 Hello from  3 of  4
 Hello from  1 of  4
 Hello from  2 of  4
```

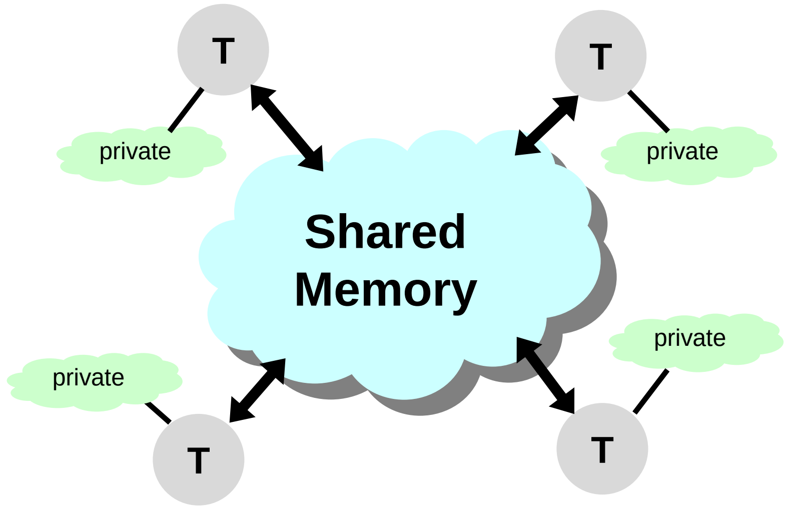
- Ordering of output is not defined
- Avoid extensive output to stdout in parallel regions!

Shared-memory parallel processing with OpenMP

Getting Started
Data Scoping
Worksharing

Data scoping: Shared vs. private data

Data in a parallel region can be:



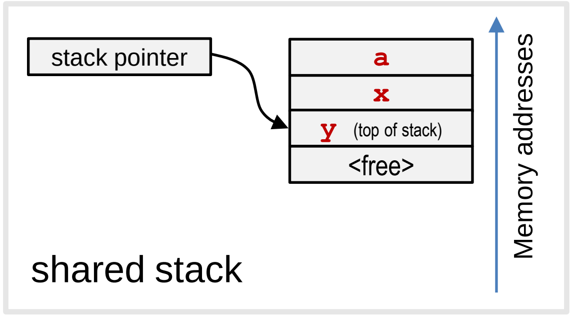
- private to each executing thread
  → each thread has its own local copy of data
- shared between threads
  → there is only one instance of data available to all threads
  → this does not mean that the instance is always visible to all threads!

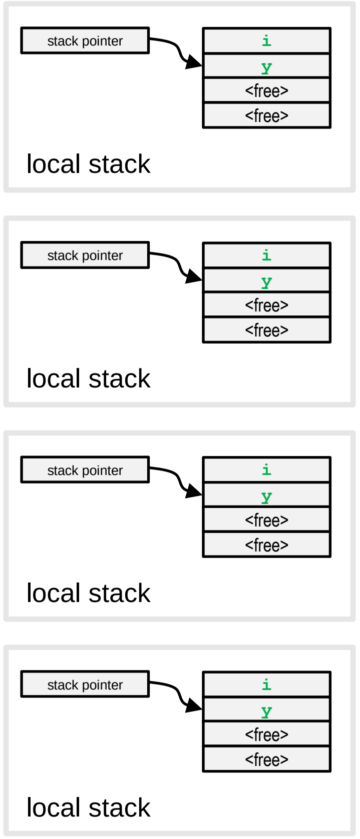
OpenMP clause specifies scope of variables:

```
#pragma omp parallel private(var1, tmp) shared(eps)
```

How is private data different from shared data?

```
void f() {
  int a;
  float x,y;
  ...
  #pragma omp parallel
  {
    int i;
    float y; // masking shared y
    ...
  }
}
```





- Local variables are kept on a stack (last-in first-out memory)
- Every thread has a private stack area
  - i.e., there is one global stack, plus one local stack for each thread
  - Private data goes to private stacks
  - Stack size is limited!

Data scoping: Shared vs. private data

- Default: All data in a parallel region is shared
  This includes global data (global/static variables, C++ class variables)
- Exceptions:
  1. Loop variables of parallel (“sliced”) loops are private (cf. workshare constructs)
  2. Local (stack) variables within parallel region
  3. Local data within enclosed function calls are private unless declared **`static`**

- Stack size limits → may be necessary to make large arrays static
  - If not possible → use heap [i.e., **`malloc()`**, **`new[]`**, **`allocate()`**]
  - **`OMP_STACKSIZE`** shell variable allows to set per-thread stack size

```
$ export OMP_STACKSIZE=100M
```

Data scoping: private data example

C:

```
include <omp.h>
...
int myid = 0, numthreads = 1;
#pragma omp parallel \
       private(myid, numthreads)
{
#ifdef _OPENMP
    myid       = omp_get_thread_num();
    numthreads = omp_get_num_threads();
#endif
   printf(“I am %d of %d\n”,
           myid, numthreads);
}
```

Fortran 90+:

```
    use omp_lib
    integer myid, numthreads
    ...
    myid = 0
    numthreads = 1
!$omp parallel private(myid,numthreads)
!$ myid = omp_get_thread_num()
!$ numthreads = omp_get_num_threads()
   print *,“I am ”,myid, &
           “ of ”,numthreads
!$omp end parallel
```

Data scoping: alternative in C

```
include <omp.h>
...
#pragma omp parallel
{
   int myid = 0, numthreads = 1;
#ifdef _OPENMP
   myid = omp_get_thread_num();
   numthreads = omp_get_num_threads();
#endif
   printf("I am %d of %d\n",
            myid, numthreads);
}
```

Local variables in structured block are automatically private! → less need for private clauses in C

Caveat: local variables are destroyed (go out of scope) at end of block!

Data scoping: important side effects

- What happens if a variable is unintentionally shared?
  - Nothing if it is just read
  - Possibly hazardous if at least one thread writes to it

```
float x = 0.0;
#pragma omp parallel
{
  x += some_work(...);
}
```

"Race condition"

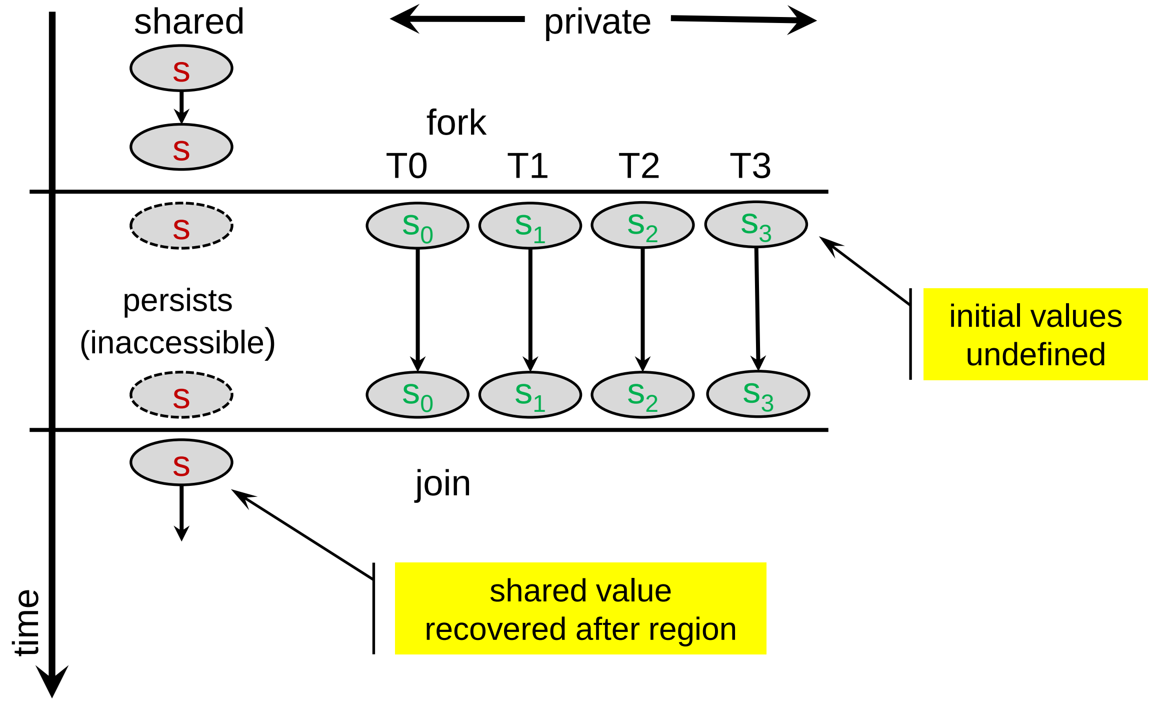
- Clause for specifying default scope: `default(shared|private|none)`
- Recommendation: Use
  `#pragma omp parallel default(none)`
  - to not overlook anything
  - compiler complains about every variable that has no explicit scoping attribute

Data scoping: private variables and masking

```
double s;

s = ...;
#pragma omp parallel private(s)
{
  s = ...;
  ... = ... + s;
}
... = ... + s;
```

Masking privatized variables defined in scope outside the parallel region



But what happens if the initial value is required within the parallel region?

The firstprivate clause

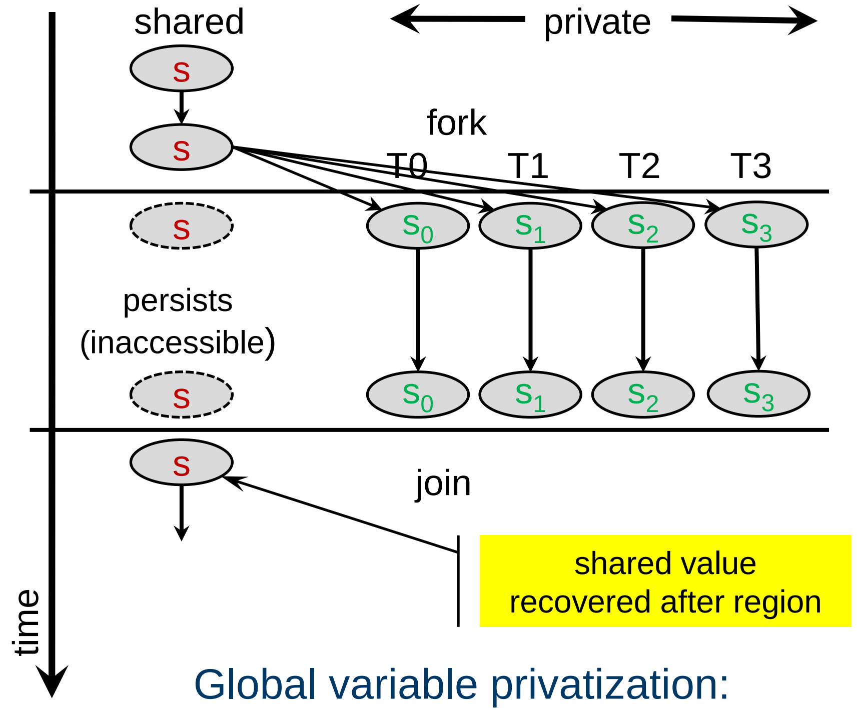
```
double s;

s = ...;
#pragma omp parallel firstprivate(s)
{
  s += ...;
  ... = ... + s;
}
... = ... + s;
```

Extension of private:

value of master copy is transferred to private variables

Restrictions: not a pointer, not assumed shape, not a subobject, master copy not itself private etc.



Global variable privatization:

threadprivate, **copyprivate** clauses

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Shared-memory parallel processing with OpenMP

Getting Started
Data Scoping
Worksharing

Worksharing: manual loop scheduling

- Work distribution by thread ID
- Only works so easily for canonical loops
- Load balancing very hard
- Complex code

→ don't do it.

```
#include
   int tid, numth, bstart, bend, blen, N;
   double a[N], b[N], c[N], d[N];
   ...
#pragma omp parallel private(tid, numth, bstart, bend, blen)
{
   tid=0; numth=1;
#ifdef _OPENMP
   tid   = omp_get_thread_num();
   numth = omp_get_num_threads();
#endif
   blen = N/numth;
   if(tid < N % numth) {
      ++blen; bstart = blen * tid;
   } else
      bstart = blen * tid + N % numth;
   bend=bstart+blen-1;
   for(int i=bstart; i<=bend; ++i)
      a[i] = b[i] + c[i] * d[i];
}
```

One consecutive chunk of iterations per thread

Actual work

Worksharing: parallel loop

- `#pragma omp for [clauses]` declares that the following loop iterations are to be distributed among threads
  - Active only if encountered within a parallel region

```
int i, N;
double a[N], b[N], c[N], d[N];
...
#pragma omp parallel       // parallel threads
{
#pragma omp for            // parallelize loop
  for(i=0; i<N; ++i)
      a[i] = b[i] + c[i] * d[i];
}
```

barriers here!

  - Loop counter of parallel loop is declared private implicitly
  - Implicit thread synchronization (barrier) at end of **`parallel`** and at end of **`for`**
- Fortran: `!$omp do [clauses]`

Worksharing: combined construct

- `#pragma omp parallel for`
  `structured block`

```
int i, N;
double a[N], b[N], c[N], d[N];
...
#pragma omp parallel for
for(i=0; i<N; ++i)
     a[i] = b[i] + c[i] * d[i];
```

- Just easier to type…

- Fortran: `!$omp parallel do  /   $!omp end parallel do`

Worksharing constructs

`#pragma omp for`

- Only the loop immediately following the directive is workshared
- Restrictions on parallel loops
  - trip count must be computable (no `do ... while`)
  - loop body with single entry and single exit point (no breaking out of loop)
- C++ random access iterator loops are supported:

```
#pragma omp for
for(auto i=v.begin(); i!=v.end(); ++i) {
  (*i) *= 2.0;
}
```

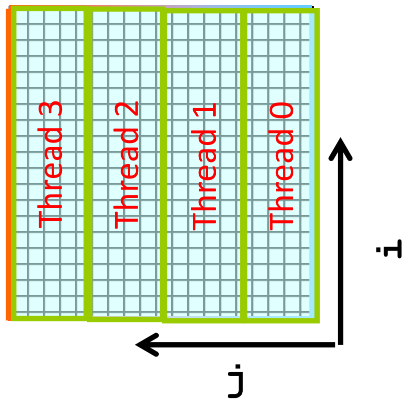
Worksharing constructs in general

- Distribute the execution of the enclosed code region among the members of the team
  - Must be enclosed dynamically within a parallel region
  - No implied barrier on entry
  - Implicit barrier at end of worksharing (unless **`nowait`** clause is specified)
- Directives
  - **`for`** directive (C/C++), **`do`** directive (Fortran)
  - **`section(s)`** directives (ignored here)
  - **`workshare`** directive (Fortran 90 only – ignored here)

  - Tasking (advanced)

Worksharing constructs example

Example: matrix processing with nested loop structure



Only these loops are parallel!

synchronization

```
double a[ndim][ndim], b[ndim][ndim];
...
#pragma omp parallel
{
#pragma omp for
  for(int j=1; j<ndim-1; ++j) {
    for(int i=1; i<ndim-1; ++i)
      a[j][i] = ( b[j][i+1]+b[j][i-1]
                 +b[j+1][i]+b[j-1][i] )*0.25;
  }
#pragma omp for
  for(int j=1; j<ndim-1; ++j) {
    for(int i=1; i<ndim-1; ++i)
      b[j][i] = ( a[j][i+1]+a[j][i-1]
                 +a[j+1][i]+a[j-1][i] )*0.25;
  }
}
```

Some workshare construct clauses

- Examples for workshare construct clauses:
  - `private, firstprivate, lastprivate`
  - `nowait`
  - `collapse(`*n*`)`
  - `schedule` (*type* [, *chunk*]) **[see next slide]**
  - `reduction` (*operator*`:`*list*) **[see later]**
  - There are some more…
- Implicit barrier at the end of loop unless `nowait` is specified (barrier may be costly!)
- `collapse`: Fuse nested loops to a single (larger one) and parallelize it
- `schedule` clause specifies how iterations of the loop are distributed among the threads of the team.

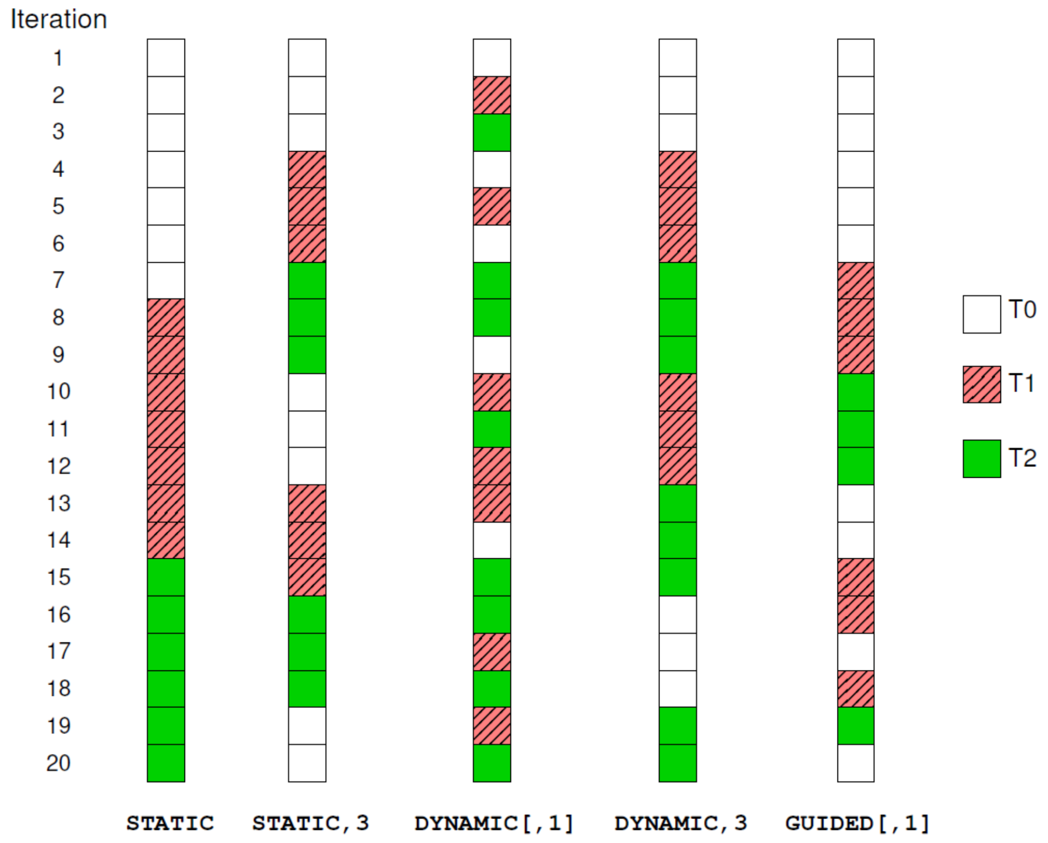
Loop worksharing: the **`schedule`** clause

Within **`schedule`** (*type* [, *chunk*]) , *type* can be one of the following:

- **`static`**: Iterations are divided into pieces of a size specified by *chunk*. The pieces are statically assigned to threads in the team in a round-robin fashion in the order of the thread number.
  Default chunk size: one contiguous piece for each thread.
- **`dynamic`**: Iterations are broken into pieces of a size specified by *chunk*. As each thread finishes a piece of the iteration space, it dynamically obtains the next set of iterations. *Default chunk size: 1.*
- **`guided`**: The chunk size is reduced in an exponentially decreasing manner with each dispatched piece of the iteration space.
  chunk specifies the smallest piece (except possibly the last).
  Default chunk size: 1. Initial chunk size is implementation dependent.
- **`runtime`**: The decision regarding scheduling is deferred until run time. The schedule type and chunk size can be chosen at run time by setting the `OMP_SCHEDULE` environment variable.
- **`auto`**: Compiler/runtime decides

- Default **`schedule`**: implementation dependent

Loop worksharing: the `schedule` clause


Iteration
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
STATIC
STATIC,3
DYNAMIC[,1]
DYNAMIC,3
GUIDED[,1]
T0
T1
T2