Advanced OpenMP Programming

R. Bader (LRZ)
G. Hager (RRZE)
V. Weinberg (LRZ)
M. Wittmann (NHR@FAU)

Work Sharing Schemes

Loops and loop scheduling
Collapsing loop nests
Parallel sections

The `schedule` clause

- **Default scheduling:**
  - implementation dependent
  - **typical:** largest possible chunks of as-equal-as-possible size („static scheduling“)

iteration space
(threads color coded)

- **User-defined scheduling:**

```
#pragma omp for schedule(...)
```

```
!$OMP do schedule(...)
```

```
            static
schedule( dynamic [,chunk] )
            guided
```

`chunk`: always a non-negative integer. If omitted, has a schedule dependent default value

- **Static scheduling**
  - `schedule(static,10)`

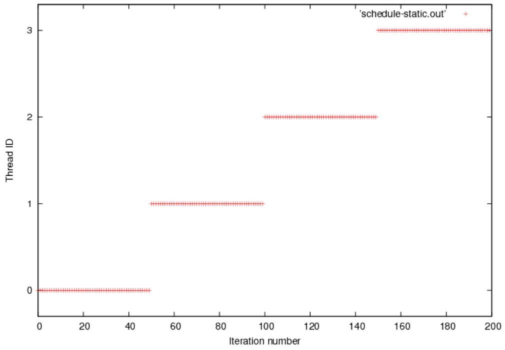
  - minimal overhead (precalculated work assignment)
  - default chunk value: see left

- **Dynamic scheduling**
  - `schedule(dynamic, 10)`

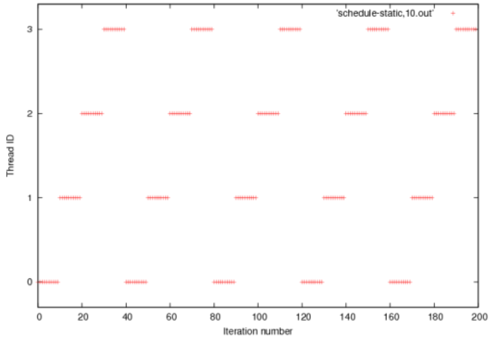
both threads take long to complete their chunk (workload imbalance)

  - after a thread has completed a chunk, it is assigned a new one, until no chunks are left
  - synchronization **overhead**
  - default chunk value is **1**

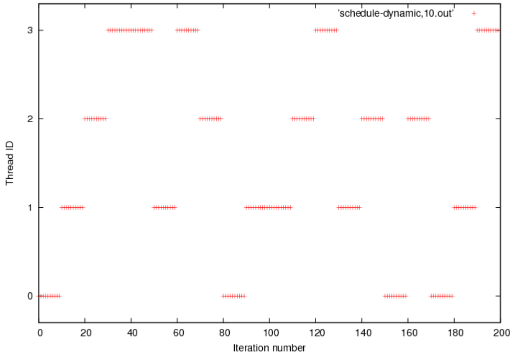
OpenMP Scheduling of simple for loops



OMP_SCHEDULE=static



OMP_SCHEDULE=static,10



OMP_SCHEDULE=dynamic,10

Guided scheduling

- **Size of chunks in dynamic schedule**
  - too small → large overhead
  - too large → load imbalance
- **Guided scheduling: dynamically vary chunk size.**
  - Size of each chunk is proportional to the number of unassigned iterations divided by the number of threads in the team, decreasing to chunk-size (default = 1).
- **Chunk size:**
  - means minimum chunk size (except perhaps final chunk)
  - default value is 1

- Both dynamic and guided scheduling useful for handling poorly balanced and unpredictable workloads.

Deferred scheduling

- **auto: automatic scheduling**
  - Programmer gives implementation the freedom to use any possible mapping.
- **Decided at run time:**

```
!$OMP do schedule(runtime)
```

```
#pragma omp for schedule(runtime)
```

- **runtime:**
  - schedule is one of the above or the previous two slides
  - determine by either setting `OMP_SCHEDULE`, and/or calling `omp_set_schedule()` (overrides env. setting)
  - find which is active by calling `omp_get_schedule()`

- **Examples:**
  - environment setting:

```
export OMP_SCHEDULE="guided,4"
./a.out
```

  - call to API routine:

```
call omp_set_schedule(omp_sched_dynamic,4)
!$OMP parallel
!$OMP do schedule(runtime)
   do
      …
   end do
!$OMP end do
```

```
omp_set_schedule(omp_sched_dynamic, 4)
#pragma omp parallel
#pragma omp schedule(runtime)
   for (…) { }
```

runtime scheduling and `OMP_SCHEDULE` is not set: implementation chooses a schedule

Collapsing loop nests

- **Collapse nested loops into a single iteration space**

```
!$OMP do collapse(2)
  do k=1, kmax
    do j=1, jmax
      :
    end do
  end do
!$OMP end do
```

argument specifies number of loop nests to flatten

```
#pragma omp for collapse(2)
for (k=0; k<kmax; ++k)
  for (j=0; j<jmax; ++j)
    ...
```

- **Restrictions:**
  - iteration space computable at entry to loop (rectangular)
  - **CYCLE** (Fortran) or **continue** (C/C++) only in innermost loop

- **Logical iteration space**
  - example: kmax=3, jmax=3

	0	1	2	3	4	5	6	7	8
J	1	2	3	1	2	3	1	2	3
K	1	1	1	2	2	2	3	3	3

  - this is what is divided up into chunks and distributed among threads
  - Sequential execution of the iterations in all loops determines the order of iterations in the collapsed iteration space

- **Optimization effect**
  - may improve memory locality properties
  - may reduce data traffic between cores

Performance Tuning: the `nowait` clause

- **Remember:**
  - an OpenMP **`for`/`do`** performs implicit synchronization at loop completion
- **Shooting yourself in the foot**
  - modified variables must not be accessed unless explicit synchronization is performed
  - use a **`barrier`** for this
- **Example: multiple loops in parallel region**

```
!$omp parallel
!$omp do
  do k=1, kmax_1
    a(k) = a(k) + b(k)
  end do
!$omp end do nowait
  ! code not involving
  ! r/w of a, writes to b
!$omp do
  do k=1, kmax_2
    c(k) = c(k) * d(k)
  end do
!$omp end do
!$omp end parallel
```

do not synchronize

Implicit barrier

```
#pragma omp parallel
{
  #pragma omp for nowait
  for (int k = 0; k < kmax_1; ++k) {
    a[k] += b[k]
  }
  /* code not involving */
  /* r/w of a, writes to b */
  #pragma omp for
  for (int k = 0; k < kmax_2; ++k) {
    c[k] *= d[k]
  }
}
```

Explicit barrier synchronization

- barrier construct is a stand-alone directive
- barrier synchronizes all threads
- each barrier must be encountered by all threads in the team or by non at all.

```
!$omp parallel
!$omp do
  do k=1, kmax_1
    a(k) = a(k) + b(k)
  end do
!$omp end do nowait
   ! code not involving
   ! r/w of a, writes to b
!$omp barrier
!$omp do
  do k=1, kmax_1
    a(k) = a(k) + b(k)
  end do
!$omp end do
!$omp end parallel
```

do not synchronize

explicit synchronization

```
#pragma omp parallel
{
  #pragma omp for nowait
  for (int k = 0; k < kmax_1; ++k) {
    a[k] += b[k]
  }
  /* code not involving */
  /* r/w of a, writes to b */
  #pragma omp barrier
  #pragma omp for
  for (int k = 0; k < kmax_1; ++k) {
    a[k] += b[k]
  }
}
```

Parallel sections

- **Non-iterative work-sharing construct**
  - distribute a set of structured blocks

```
!$omp parallel
!$omp sections
!$omp section
    ! code block 1
!$omp section
    ! code block 2
…
!$omp end sections
!$omp end parallel
```

thread 0

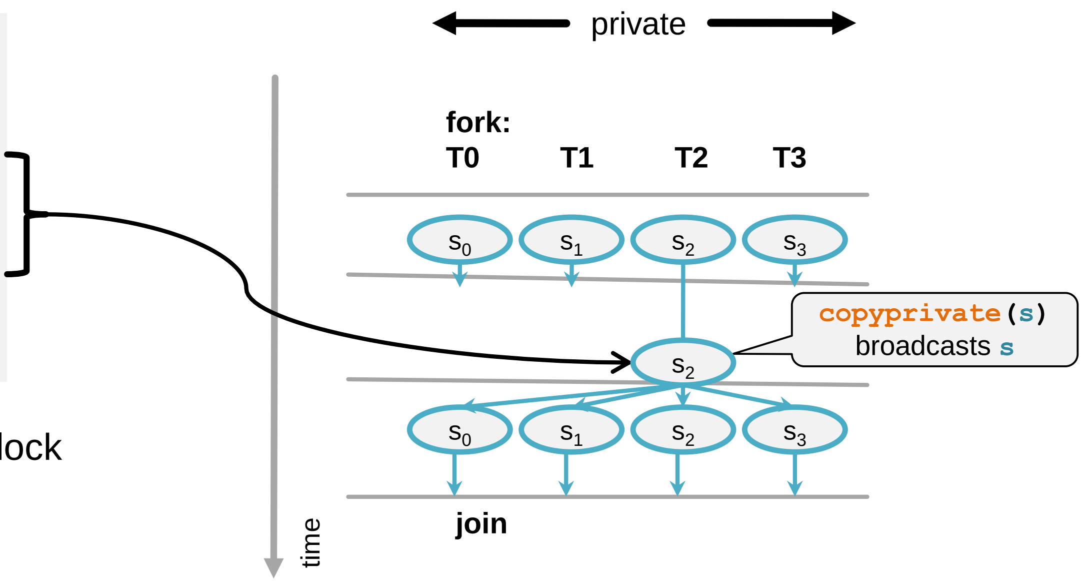
thread 1

synchronization

  - each block executed exactly once by one of the threads in team
- **Allowed clauses on sections:**
  - `private`, `firstprivate`, `lastprivate`, `reduction`, `nowait`
- **Restrictions:**
  - `section` directive must be within lexical scope of sections directive
  - `sections` directive binds to innermost parallel region
  - → only the threads executing the binding parallel region participate in the execution of the section blocks and the implicit barrier (if not eliminated with `nowait`)
- **Scheduling to threads**
  - implementation-dependent
  - if there are more threads than code blocks: excess threads wait at synchronization point

The single directive


```
#pragma omp parallel
{
  double s = …;
  #pragma omp single copyprivate(s)
  {
    s = …
  }
  … = … + s
}
```



- one thread only executes enclosed code block
- all other threads wait until block completes execution
- allowed clauses: **`private`**, **`firstprivate`**, **`copyprivate`**, **`nowait`**
- use for updates of shared entities, but …
  - single – really a worksharing directive?

- **`copyprivate`** and **`nowait`** clauses: appear on **`end single`** in Fortran, on **`single`** in C/C++

```
!$omp single
  s = …
!$omp end single copyprivate(s)
```

Combining regions and work sharing

- **Example:**

```
!$OMP parallel do
...
!$OMP end parallel do
```

```
#pragma omp parallel for
...
```

- **is equivalent to**

```
!$omp parallel
!$omp do
...
!$omp end do
!$omp end parallel
```

```
#pragma omp parallel
#pragma omp for
...
```

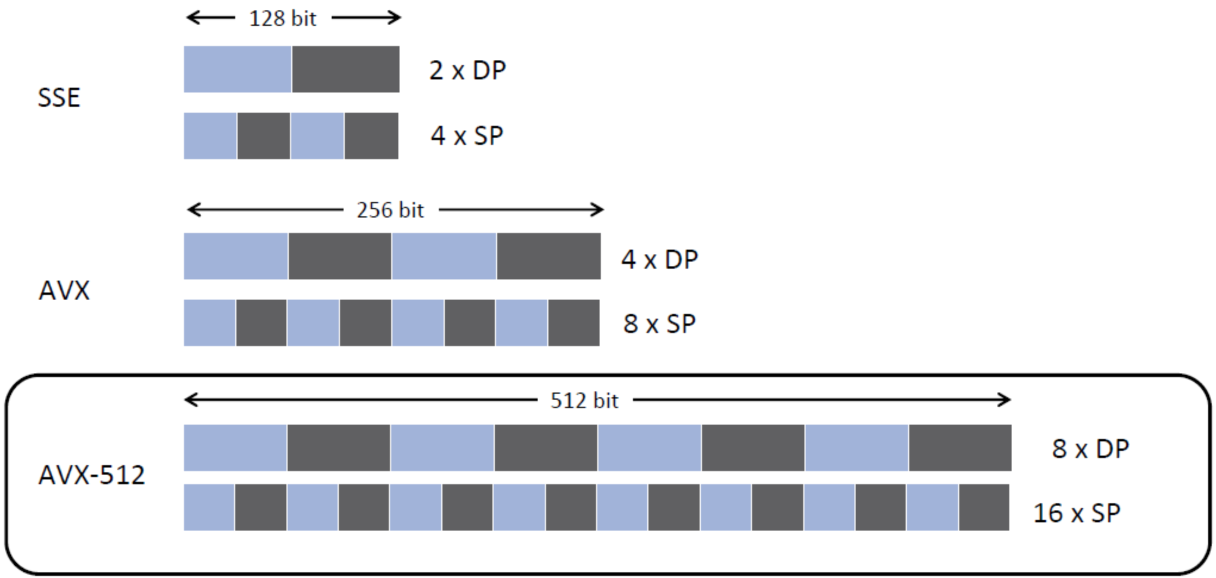
- **Applies to most work-sharing constructs**
  - `do`/`for`
  - `workshare`
  - `sections`
- **Notes:**
  - clauses for work-sharing constructs can appear on combined construct
  - the reverse is not true
    `shared` can only appear in a parallel region
  - clauses on a work-sharing construct only apply for the specific construct block

Vectorization with OpenMP SIMD

Acknowledgements: M. Klemm (OpenMP ARB), C. Terboven (RWTH Aachen)

SIMD on Intel Architecture

- Width of SIMD (Single Instruction, Multiple Data) registers has been growing in the past:



Before OpenMP 4.0

- **Support required vendor-specific extensions**
  - Programming models (e.g. Intel Cilk Plus)
  - Compiler pragmas (e.g. **`#pragma vector`**)
  - Low-level constructs (e.g. **`_mm_add_pd()`**)

```
#pragma omp parallel for
#pragma vector always
#pragma ivdep
for (int i = 0; i < n; i++) {
  a[i] = b[i] + …;
}
```

You need to trust your compiler to do the right thing

SIMD Loop Construct

- **Vectorize a loop nest**
  - Cut loop into chunks that fit a SIMD vector register
  - No parallelization of the loop body

C/C++

```
#pragma omp simd [clause, …]
for-loops
```

Fortran

```
!$omp simd [clause, …]
do-loops
!$omp end simd
```

- **`simd` construct can be applied to a loop to indicate that the loop can be transformed into a SIMD loop**
  - multiple iterations of the loop can be executed concurrently using SIMD instructions

- **`simd` specifies that there are no dependencies among loop iterations**
  - see `safelen` clause

SIMD Loop Construct: Clauses

- **`private (var-list)`**
  uninitialized vectors for variables in `var-list`
- **`reduction (op:var-list)`**
  create private variables for `var-list` and apply reduction operator `op` at the end of the construct
- **`simdlen (length)`**
  length is treated as a hint that specifies the preferred number of iterations to be executed concurrently
- **`safelen (length)`**
  maximum number of iterations that can run concurrently without breaking a dependence
- **`linear (list[:linear-step])`**
  the variable's value is in relationship with the iteration number $x_i = x_{orig} + i * \text{linear-step}$
- **`aligned (list[:alignment])`**
  specifies that the list items have a given alignment
- **`collapse (n)`**
  collapse n nested loops into a single iteration space

Examples of the SIMD construct

```
#pragma omp simd
for (i=0; i<n); i++)
  a[i] = b[i] + c[i];
```

```
#pragma omp simd private(t1, t2)
for (i=0; i<n; i++) {
 t1 = funca(b[i], c[i]);
 t2 = funcb(b[i], c[i]);
 a[i] = t1 + t2;
}
```

```
#pragma omp simd reduction(+:t1) collapse(2)
for (i=0; i<n; i++)
  for (j=0; j<m; j++)
     t1 += func1(b[i], c[j]);
```

SIMD Worksharing Construct

- **Parallelize and vectorize a loop next**
  - Distribute a loop’s iteration space across a thread team
  - Subdivide loop chunks to fit a SIMD vector register

C/C++

```
#pragma omp for simd [clause, …]
for-loops
```

Fortran

```
!$omp do simd [clause, …]
do-loops
!$omp end
```

- **OpenMP 4.5. simplifies SIMD chunks:**
  - `schedule(simd: static, n)`
  - Chooses chunk sizes that are multiples of the SIMD length

© 2010-2023

SIMD Function Vectorization

- **Declare one or more functions to be compiled for calls from a SIMD loop**
  - You can think of this as a special kind of SIMD function prototype
  - Compiler may generate multiple versions of a SIMD function and select the appropriate version to invoke at a specific call-site in a `simd` construct

C/C++

```
#pragma omp declare simd
            [clause, …]
function declaration/definitions
```

Fortran

```
!$omp declare simd
             (proc-name-list)
function/subroutine &
  declaration/definitions
```

SIMD Function Vectorization Clauses

- **`simdlen (length)`**
  generate function to support a given vector length (not a hint as for `simd`!)
- **`uniform (argument-list)`**
  argument has a constant value between the iterations of a given loop
- **`inbranch`**
  function is always called from inside an if statement
- **`notinbranch`**
  function is never called from inside an if statement
- **`linear (list[:linear-step])`**
  indicates that an argument passed to a function parameter has a linear relationship across the concurrent invocations of a function (not a data-sharing clause as for `simd`!)
- **`aligned (list[:alignment])`**
  declares that the value of the pointer variable argument has the specified byte alignment, the SIMD version of the function may then use aligned vector memory accesses for the the pointer variable

Example of SIMD Function Vectorization


```
#pragma omp declare simd
double my_func (double b, double c)
{
  double r;
  r = b + c;
  return r;
}
```

Instructs the compiler to generate at least one additional SIMD version of the function `my_func`.

```
void simd_loop_function(double *a, double *b,
                        double *c, int n)
{
  #pragma omp simd
  for (int i=0; i<n; i+=2) {
    a[i] = my_func(b[i], c[i]);
  }
}
```

Call to `my_func` will be to a SIMD variant of the function.

Synchronization and its issues

Memory model

Additional directives

Performance issues

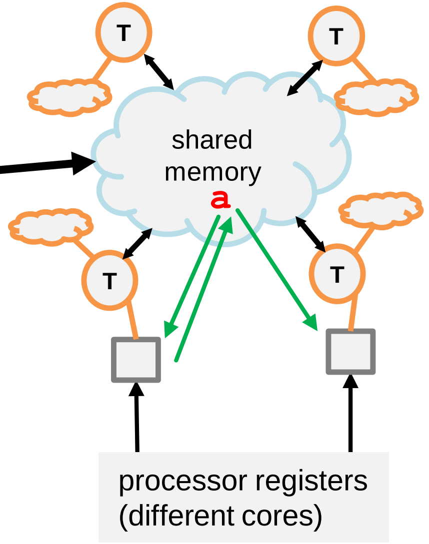
User-defined synchronization

Why do we need synchronization?

- **OpenMP Memory Model**

- **private (thread-local):**
  - no access by other threads

- **shared: two views**
  - **temporary view**: thread has modified data in its registers (or other intermediate device)
  - content becomes inconsistent with that in cache/memory
  - **other threads**: cannot know that their copy of data is **invalid**

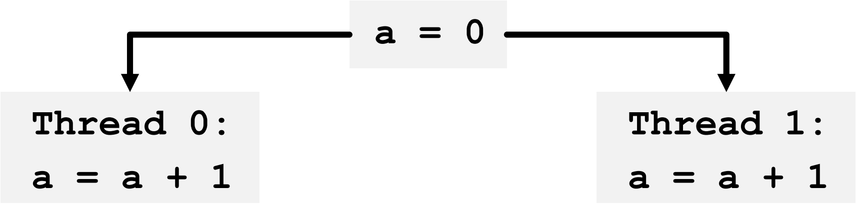


two threads execute

`a = a + 1`

in same parallel region

→ **race condition**



- **Following results could be obtained on each thread**
  - a after completion of statement:

Thread 0	Thread 1
1	1
1	2
2	1

  - may be different from run to run, depending on which thread is the last one
  - after completion of parallel region, may obtain 1 or 2.

Consequences and (theoretical) remedies

- **For threaded code without synchronization this means**
  - multiple threads write to same memory location
    - resulting value is unspecified
  - some threads read and another writes
    - result on reading threads unspecified
- **Flush Operation**
  - is performed on a set of (shared) variables or on the whole thread-visible data state of a program

    → flush-set
  - discards temporary view:

    → modified values forced to cache/memory

    → next read access must be from cache/memory
  - further memory operations only allowed after all involved threads complete flush:
    - restrictions on memory instruction reordering (by compiler)
- **Ensure consistent view of memory:**
  - assumption: want to write a data item with first thread, read it with second
  - order of execution required:
    1. thread 1 writes to shared variable
    2. thread 1 flushes variable
    3. thread 2 flushes same variable
    4. thread 2 reads variable

OpenMP flush syntax

- **OpenMP directive for explicit flushing**

```
!$omp flush [(var1[,var2,…])]
```

- **Stand-alone directive**
- **applicable to all variables with shared scope**
  - including: **SAVE**, **COMMON**/module globals, shared dummy arguments, shared pointer dereferences
- **If no variables specified, the flush-set**
  - encompasses all shared variables
    which are accessible in the scope of the FLUSH directive
  - potentially slower
- **Implicit flush operations (with no list) occur at:**
  - All explicit and implicit barriers
  - Entry to and exit from critical regions
  - Entry to and exit from lock routines

Barrier synchronization

- **Explicit via directive:**
  - the execution flow of each thread blocks upon reaching the barrier until all threads have reached the barrier
  - flush synchronization of all accessible shared variables happens before all threads continue
    - after the barrier, all shared variables have consistent value visible to all threads
  - barrier may not appear within work-sharing code block
    - e.g. `!$omp do` block, since this would imply deadlock
- **Implicit for some directives:**
  - at the beginning and end of parallel regions
  - at the end of `do`, `single`, `sections`, workshare blocks unless a `nowait` clause is specified (where allowed)
  - all threads in the executing team are synchronized
  - this is what makes these directives “easy-and-safe-to-use”

Relaxing synchronization requirements

- **Use a `nowait` clause**
  - on **`end do`** / **`end sections`** / **`end single`** / **`end workshare`** (Fortran)
  - on **`for`** / **`sections`** / **`single`** (C/C++)
  - removes the synchronization at end of block
  - potential performance improvement
    - especially if load imbalance occurs within construct)
  - programmer's responsibility to prevent races

- **The `critical` and `atomic` directives:**
  - each thread arriving at the code block executes it (in contrast to **`single`**)
  - mutual exclusion: only one at a time within code block
  - **`atomic`**: code block must be a single line update of a scalar entity of intrinsic type with an intrinsic operation

Fortran

```
!$omp critical
  block
!$omp end critical
```

```
!$omp atomic
x = x <op> y
```

unary operator also allowed

C/C++

```
# pragma omp critical
{ block }
```

```
# pragma omp atomic
x = x <op> y ;
```

Synchronizing effect of critical regions

- **Mutual exclusion is only assured for the statements inside the block**
  - i.e., subsequent threads executing the block are synchronized against each other
- **If other statements access the shared variable, may be in trouble:**

C/C++

```
#pragma omp parallel
{
  :
  #pragma omp atomic
  x = x + y
  :
  a = f(x, …)
}
```

Fortran

```
!$omp parallel
  :
!$omp atomic
  x = x + y
  :
  a = f(x, …)
!$omp end parallel
```

- Race on read to x.
- A barrier is required before this statement to assure that all threads have executed their atomic updates

Named critical

- **Consider multiple updates**
  - same shared variable

thread 0

```
subroutine foo()
!$omp critical
  x = x + y
!$omp end critical
```

thread 1

```
subroutine bar()
!$omp critical
  x = x + z
!$omp end critical
```

    - critical region is global: OK
  - different shared variables

```
subroutine foo()
!$omp critical
  x = x + y
!$omp end critical
```

```
subroutine bar()
!$omp critical
  w = w + z
!$omp end critical
```

    - mutual exclusion not required
    - unnecessary loss of performance

- **Solution:**
  - use named criticals

```
subroutine foo()
!$omp critical (foo_x)
  x = x + y
!$omp end critical (foo_x)
```

```
subroutine bar()
!$omp critical (foo_w)
  w = w + z
!$omp end critical (foo_w)
```

  - mutual exclusion only if same name is used for `critical`
- **`atomic` is bound to updated variable**
  - problem does not occur

The master directive (depricated)

Fortran

```
!$omp master
  block
!$omp end master
```

C/C++

```
#pragma omp master
{ block }
```

- **Only thread zero (from the current team) executes the enclosed code block**
  - There is no implied barrier either on entry to, or exit from, the master construct. Other threads continue without synchronization
- **Not all threads must reach the construct**
  - if the master thread does not reach it, it will not be executed at all
- **Equivalent to:**

```
if (omp_get_thread_num() == 0) { … }
```

The masked directive

≥ v5.1

Fortran

```
!$omp masked [filter(scalar-integer-expression)]
 block
!$omp end masked
```

C/C++

```
# pragma omp masked [filter(integer-expression)]
{ block }
```

- **only threads selected by the filter clause execute the structured block**
- **other threads in the team do not execute the associated structured block.**

- **If a filter clause is present on the construct and the parameter specifies the thread number of the current thread in the current team then the current thread executes the associated structured block.**

- **No implied barrier on entry to, or exit from, the `masked` construct.**

The `ordered` clause and directive

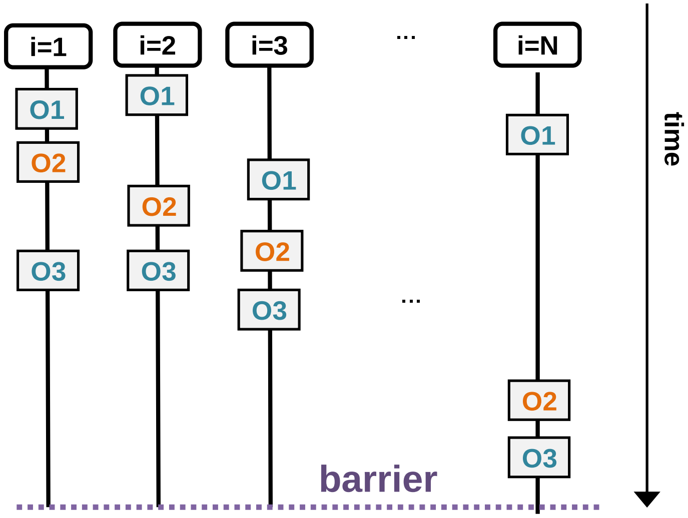
- **Statements must be within body of a loop**
  - directive acts similar to **`single`**
    - threads do work ordered as in sequential execution
    - execution in the order of the loop iterations
  - requires **`ordered`** clause on enclosing **`do`**/**`for`** construct
  - only effective if code is executed in parallel
  - only one ordered region per loop

C/C++

```
#pragma omp for ordered
for (i=0; i<N; ++i) {
  O1
  #pragma omp ordered
  { O2 }
  O3
}
```

Fortran

```
!$OMP do ordered
  do I=1,N
    O1
    !$OMP ordered
      O2
    !$OMP end ordered
    O3
  end do
!$OMP end do
```



Two applications of ordered

- **Loop contains recursion**
  - dependency requires serialization
  - only small part of loop (otherwise performance issue)

Fortran

```
!$OMP do ordered
do I=2,N
   ... ! large block
   !$OMP ordered
     a(I) = a(I-1) + ...
   !$OMP end ordered
end do
!$OMP end do
```

C/C++

```
#pragma omp for ordered
for (i=1; i<N; ++i) {
   ... /* large block */
   #pragma omp ordered
   a[i] = a[i-1] + ...
}
```

- **Loop contains I/O**
  - it is desired that output (file) be consistent with serial execution

Fortran

```
!$OMP do ordered
do I=1,N
   ... ! calculate a(I)
   !$OMP ordered
     write(unit,...) a(I)
   !$OMP end ordered
end do
!$OMP end do
```

C/C++

```
#pragma omp for ordered
for (i=0; i<N; ++i) {
   ... /* calculate a[i] */
   #pragma omp ordered
     printf("%e ", a[i]);
   }
}
```

Mutual exclusion with locks

- **A shared lock variable can be used to implement specifically designed synchronization mechanisms**
  - In the following, var is of type
    - `Fortran: integer(omp_lock_kind)`
    - `C/C++:    omp_lock_t`
  - OpenMP lock variables must be only accessed by the lock routines

- **Mutual exclusion bound to objects**
  - more flexible than critical regions

OpenMP locks

- **An OpenMP lock can be in one of the following 3 stages:**
  - uninitialized
  - unlocked
  - locked
- **The task that sets the lock is then said to own the lock.**

- **Only a task that sets the lock, can unset the lock, returning it to the unlocked stage.**

- **2 types of locks are supported:**
  - simple locks
    - Can only be locked if unlocked.
    - A thread may not attempt to re-lock a lock it already has acquired.
  - nestable locks
    - Owning thread can lock multiple times
    - Owning thread must unlock the same number of times it locked it

Lock routines (1)

- **Fortran:** `omp_init_lock(var)`
  C/C++ `omp_init_lock(omp_lock_t *var)`
  - initialize a lock
  - initial state is unlocked
  - what resources are protected by lock: defined by developer
  - **`var`** not associated with a lock before this routine is called

- **Fortran:** `omp_destroy_lock(var)`
  C/C++: `omp_destroy_lock(omp_lock_t *var)`
  - disassociate **`var`** from lock
  - precondition:
    - **`var`** must have been initialized
    - **`var`** must be in unlocked state

Lock routines (2)

- **Assuming: lock variable `var` has been initialized**

- **Fortran:** `omp_set_lock(var)`
  C/C++: `void omp_set_lock(omp_lock_t *var)`
  - blocks if lock not available
  - set ownership and continue execution if lock available
- **Fortran:** `omp_unset_lock(var)`
  C/C++: `void omp_unset_lock(omp_lock_t *var)`
  - release ownership of lock
  - ownership must have been established before
- **Fortran:** `logical function omp_test_lock(var)`
  C/C++: `int omp_test_lock(omp_lock_t *var)`
  - does not block, tries to set ownership
  - returns true if lock was set, false if not
  - allows to do something else while lock is hold by another thread

Example for using locks

```
use omp_lib
integer(omp_lock_kind) :: lock

call omp_init_lock(lock)

!$omp parallel
  ...
  call omp_set_lock(lock)
  ! use resource protected by lock
  call omp_unset_lock(lock)
  ...
!$omp end parallel

call omp_destroy_lock(lock)
```

acts like a `critical` region

```
use omp_lib
integer(omp_lock_kind) :: lock

call omp_init_lock(lock)

!$omp parallel
  ...
  do while (.not. omp_test_lock(lock))
    ! work unrelated to lock protected
    ! resource
  end do
  ! use lock protected resource
  call omp_unset_lock(lock)
  ...
!$omp end parallel

call omp_destroy_lock(lock)
```

loop until lock calling thread hold lock

Example for using locks

```
#include <omp.h>
omp_lock_t lock;

omp_init_lock(&lock);

#pragma omp parallel
{
   ...
   omp_set_lock(&lock)
   /* use resource protected
      by lock */
   omp_unset_lock(&lock)
   ...
}

omp_destroy_lock(&lock)
```

acts like a **critical** region

```
#inclue <omp.h>
omp_lock_t lock;

omp_init_lock(&lock)

#pragma omp parallel
{
   ...
   while (!omp_test_lock(&lock)) {
     /* work unrelated to lock
        protected resource */
   }
   /* use lock protected
      resource */
   omp_unset_lock(&lock)
   ...
}

omp_destroy_lock(&lock)
```

loop until lock calling thread hold lock

Nestable Locks

- **replace `omp_*_lock` by `omp_*_nest_lock`**

- **task owning a nestable lock may re-lock it multiple times**
  - a nestable lock is available if it is either unlocked

  or
  - it is already owned by the task executing `omp_set_nest_lock()` or `omp_test_nest_lock()`

- **re-locking increments nest count**
- **releasing the lock decrements nest count**
- **lock is unlocked once nest count is zero**

Tasking

Work sharing for irregular problems, recursive problems and information structures

Acknowledgements: M. Klemm (AMD) / L. Meadows / T. Mattson (Intel)

Task Execution Model

- **Supports unstructured parallelism**
  - unbounded loops

```
while (<expr>) {
   ...
}
```

```
do while (<expr>)
   ...
end do
```

  - recursive functions

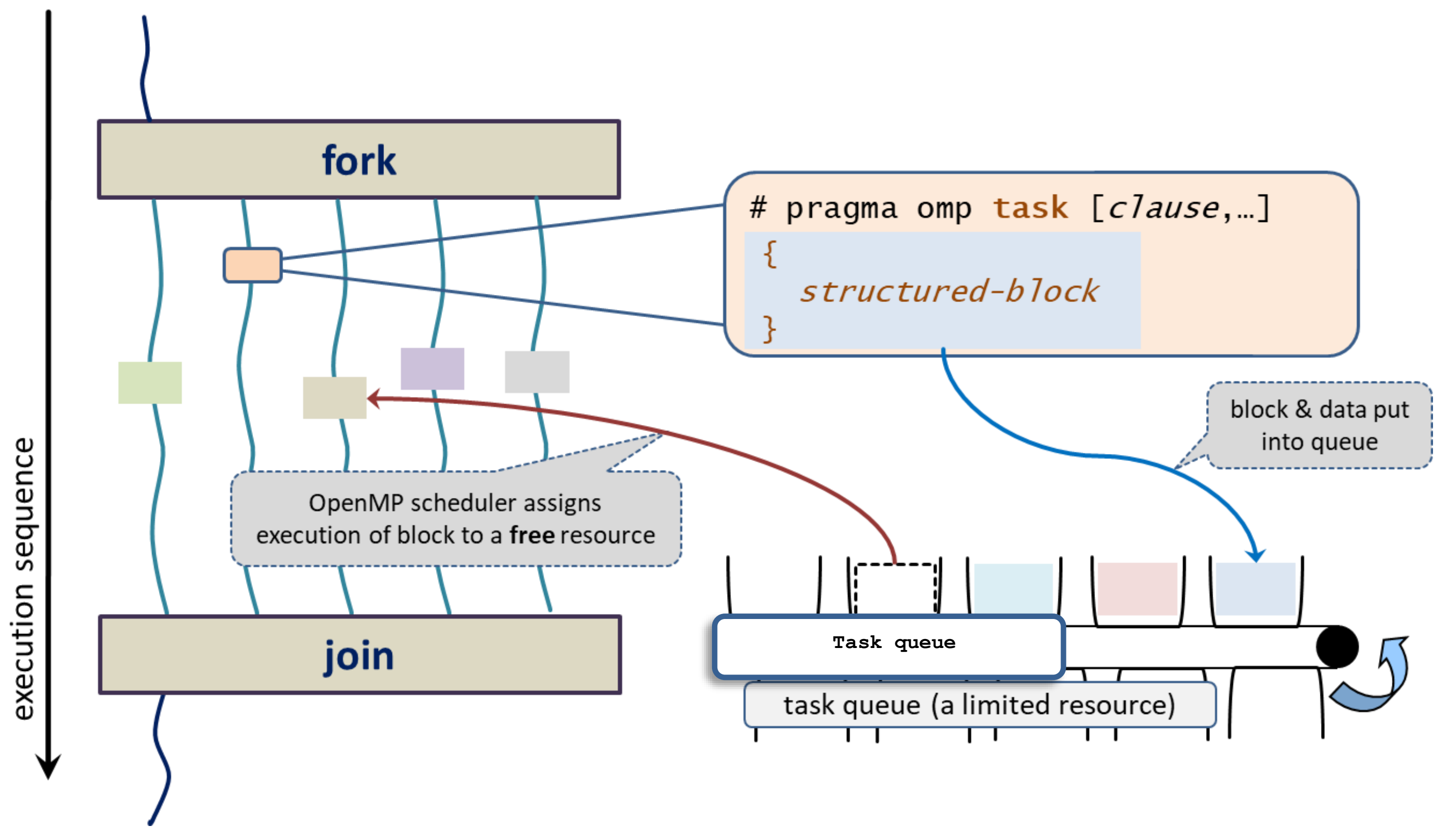
```
void myfunc(<args>)
{
   ...
   myfunc(<newargs>)
   ...
}
```

- **Several scenarios are possible**
  - single creator, multiple creators, nested tasks,
  - All threads in the team are candidates to execute tasks

- **Example of unstructured parallelism**

```
#pragma omp parallel
#pragma omp single
while (elem != NULL) {
   #pragma omp task
   compute(elem);

   elem = elem->next;
}
```

The Execution Model



The `task` Construct

- **Deferring (or not) a unit of work (executable for any member of the team)**

```
#pragma omp task [clause[[,] clause]...]
{structured-block}
```

```
!$omp task [clause[[,] clause]...]
…structured-block…
!$omp end task
```

- **Clauses:**
  - data environment:
    - **`private`**, **`fistprivate`**, **`default(shared|none)`**, **`in_reduction(r-id:list)`**
  - Dependencies:
    - **`depend(dep-type: list)`**
  - Scheduler restriction:
    - **`untied`**
  - Scheduler hints:
    - **`priority(priority-value)`**
    - **`affinity(list)`**
  - cutoff strategies:
    - **`if(scalar-expression)`**
    - **`mergable`**
    - **`final(scalar-expression)`**
  - Other clauses:
    - **`allocate(allocator:] list)`**
    - **`detach(event-handler)`**

What is a Task?

- **Make OpenMP worksharing more flexible:**
  - allow the programmer to package code blocks and data items for execution
    - this by definition is a task
  - and assign these to an encountering thread
  - possibly defer execution to a later time („work queue“)
- **Introduced with OpenMP 3.0 and extended over time**

- **When a thread encounters a `task` construct, a task is generated from the code of the associated structured block.**
- **Data environment of the task is created (according to the data-sharing attributes, defaults, …)**
  - „Packaging of data“
- **The encountering thread may immediately execute the task, or defer its execution. In the latter case, any thread in the team may be assigned the task.**

Example: Processing a Linked List

```
typedef struct {
  list *next;
  contents *data;
} list;

void process_list(list *head)
{
  #pragma omp parallel
  {
    #pragma omp single
    {
      list *p = head;
      while(p) {
        #pragma omp task
        { do_work(p->data); }
        p = p->next;
      }
    } /* all tasks done */
  }
}
```

Typical task generation loop:

```
#pragma omp parallel
{
  #pragma omp single
  {
    while(p) {
      #pragma omp task
      { /* taks code */ }
    }
  } /* all tasks done */
}
```

Example: Processing a Linked List

```
typedef struct {
  list *next;
  contents *data;
} list;

void process_list(list *head)
{
  #pragma omp parallel
  {
    #pragma omp single
    {
      list *p = head;
      while (p) {
        #pragma omp task
        { do_work(p->data); }
        p = p->next;
      }
    } /* all tasks done */
  }
}
```

- **Features of this example:**
  - one of the threads has the job of generating all tasks
  - synchronization: at the end of the `single` block for all tasks created inside it
  - no particular order between tasks is enforced here
  - data scoping default for task block:
    - `firstprivate`
    - iterating through p is fine
    - this is the „packaging of data" mentioned earlier
  - task region: includes call of `do_work()`

The `if` Clause

- **When `if` argument is false –**
  - task becomes an undeferred task
  - task body is executed immediately by encountering thread
  - all other semantics stay the same (data environment, synchronization) as for a „deferred“ task

```
#pragma omp task if (sizeof(p->data) > threshold)
{ do_work(p->data); }
```

- **User-directed optimization:**
  - avoid overhead for deferring small tasks
  - cache locality / memory affinity may be lost by doing so

Task Synchronization

Task Synchronization with barrier and taskwait

- **OpenMP barrier (implicit or explicit)**
  - All tasks created by any thread of the current team are guaranteed to be completed at **barrier** exit

C/C++

```
#pragma omp barrier
```

Fortran

```
!$omp barrier
```

```
#pragma omp parallel
#pragma omp single
{
  while (elem != NULL) {
    #pragma omp task
    compute(elem);

    elem = elem->next;
  }
} /* impl. barrier */
```

- **Task barrier: taskwait**
  - Encountering task is suspended until child tasks are complete
  - Applies to direct children only, not descendants!

C/C++

```
#pragma omp taskwait
```

Fortran

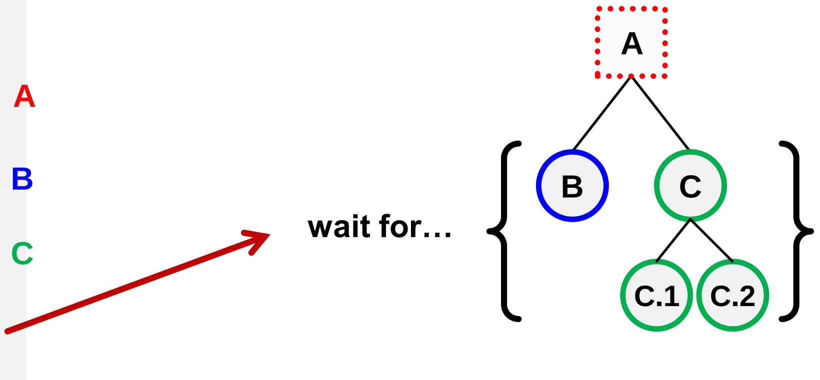
```
!$omp taskwait
```

```
#pragma omp parallel
#pragma omp single
{
  #pragma omp task
  { ... } /* A */

  #pragma omp task
  { ... } /* B */

  #pragma omp taskwait
  ...
} /* impl. barrier */
```

- **`taskgroup` construct**
  - deep task synchronization
  - attached to a structured block
    - completion of all descendants of the current task
    - task synchronization point (TSP) at the end, see later slides

```
#pragma omp taskgroup [allocate] \
                      [task_reduction(list)]
{structured-block}
```

```
#pragma omp parallel
#pragma omp single
{
   #pragma omp taskgroup
   {
       #pragma omp task
       { … }
       #pragma omp task
       { … #C.1; #C.2; …}

   } // end of taskgroup
}
```



Task Synchronization Constructs: Comparison

Task Synchronization Construct	Description
`barrier`	Either an implicit, or explicit barrier.
`taskwait`	Wait on the completion of child tasks of the current task.
`taskgroup`	Wait on the completion of child tasks of the current task and their descendants.

- **Example:**
  - Assure leaf-to-root traversal for a binary tree

```
void process_tree(tree *root)
{
  if (root->left) {
    #pragma omp task
    { process_tree(root->left); }
  }

  if (root->right) {
    #pragma omp task
    { process_tree(root->right);}
  }

  #pragma omp taskwait
  do_work(root->data);
}
```

- **What if we run out of threads?**
  - Do we hang?

Task Switching

- **It is allowed for a thread to**
  - suspend a task during execution
  - start (or resume) execution of another task (assigned to the same team)
  - resume original task later
- **Pre-condition:**
  - a task scheduling point (TSP) is reached
- **Example from previous slide:**
  - the `taskwait` directive implies a task scheduling point
- **Another example:**
  - very many tasks are generated
  - implementation can suspend generation of tasks and start processing the existing queue
- **Nearly all task scheduling points are implicit**
  - `taskyield` construct only explicit one

Task Scheduling Points

- The point immediately following the generation of an explicit task.
- After the point of completion of a `task` region.
- At a `taskyield` directive.
- At a `taskwait` directive.
- At the end of a `taskgroup` region.
- At an implicit or explicit `barrier` directive.

```
#define LARGE_N 10000000
double item[LARGE_N]
extern void process(double);

int main()
{
  #pragma omp parallel
  {
    #pragma omp single
    {
      for(int i = 0; i< LARGE_N; i++) {
        #pragma omp task
        process(item[i]);
      }
    }
  return 0;
}
```

task scheduling point

item is shared

i is firstprivate

- **Features of this example:**
  - generates a large number of tasks with one thread and executes them with the threads in the team
  - implementation may reach its limit on unassigned tasks
  - if it does, the implementation is allowed to cause the thread executing the task generating loop to suspend its task at the scheduling point and start executing unassigned tasks.
  - once the number of unassigned tasks is sufficiently low, the thread may resume executing of the task generating loop.

Example: The `taskyield` Directive

```
#include <omp.h>

void something_useful();
void something_critical();

void foo(omp_lock_t * lock, int n)
{
  for(int i = 0; i < n; i++) {
    #pragma omp task
    {
      something_useful();
      while (!omp_test_lock(lock)) {
        #pragma omp taskyield
      }
      something_critical();
      omp_unset_lock(lock);
    }
  }
}
```

- **`taskyield` directive introduces an explicit task scheduling point (TSP).**
- **May cause the calling task to be suspended.**

The waiting task may be suspended here and allow the executing thread to perform other work. This may also avoid deadlock situations.

Task Switching: Tied and Untied Tasks

- **Default behavior:**
  - a task assigned to a thread must be (eventually) completed by that thread
  - task is tied to the thread
- **Change this via the `untied` clause**
  - execution of task block may change to another thread of the team at any task scheduling point
  - implementation may add task scheduling points beyond those previously defined (outside programmer‘s control!)

```
#pragma omp task untied
structured-block
```

- **Deployment of untied tasks**
  - starvation scenarios: running out of tasks while generating thread is still working on something
- **Dangers:**
  - more care required (compared with tied tasks) wrt. scoping and synchronization

Final and mergeable Tasks

- **Final Tasks**
  - use a **`final`** clause with a condition
  - always undeferred,
    - executed immediately by the encountering thread
  - reducing the overhead of placing tasks in the "task pool"
  - all tasks created inside final task region are also final
    - different from an **`if`** clause
  - use **`omp_in_final()`** to test if task is final
- **Merged Tasks**
  - using a **`mergeable`** clause may create a merged task if it is **undeferred** or **final**
  - a merged task has the same data environment as its creating task region
  - Clause was introduced to reduce data / memory requirements
- **Final and/or mergeable**
  - can be used for optimization purposes
  - optimize wind-down phase of a recursive algorithm

Task Data Scoping

Data Environment

- **The task directive takes the following data attribute clauses that define the data environment of the task:**

  - **`default (private | firstprivate | shared | none)`**
  - **`private (list)`**
  - **`firstprivate (list)`**
  - **`shared (list)`**

Data Scoping

- **Some rules from Parallel Regions apply:**
  - Static and global variables are **`shared`**
  - Automatic storage (stack) variables are **`private`**
- **The OpenMP Standard says:**
  - The data-sharing attributes of variables that are not listed in data attribute clauses of a task construct, and are not predetermined according to the OpenMP rules, are implicitly determined as follows:
  - In a task construct, if no **`default`** clause is present:
    a) a variable that is determined to be **`shared`** in all enclosing constructs, up to and including the innermost enclosing parallel construct, is **`shared`**.
    b) a variable whose data-sharing attribute is not determined by rule (a) is **`firstprivate`**.

```
int i = 0, j = 1;
#pragma omp parallel private(j)
#pragma omp single
{
  int k = 0;
  #pragma omp task
  { /* use i, j, k */ }
}
```

shared (i)

firstprivate (j, k)

Task Reductions

Reminder: `taskgroup` construct

- **`taskgroup` construct**
  - deep task synchronization
  - attached to a structured block
    - completion of all descendants of the current task
    - task synchronization point (TSP) at the end, see later slides

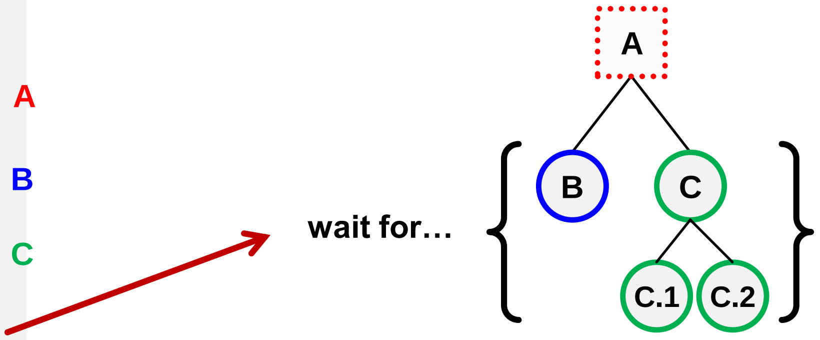
```
#pragma omp taskgroup [allocate] \
                      [task_reduction(list)]
{structured-block}
```

```
#pragma omp parallel
#pragma omp single
{
    #pragma omp taskgroup
    {
        #pragma omp task
        { … }
        #pragma omp task
        { … #C.1; #C.2; …}

    } // end of taskgroup
}
```

A

B

C



Task Reductions using the `taskgroup` Construct

- **Reduction operation**
  - perform some forms of recurrence calculations
  - associative and commutative operators
- **The `taskgroup` scoping reduction clause**

```
#pragma omp taskgroup \
            task_reduction(op: list)
{structured-block}
```

  - Register a new reduction at (1)
  - Computes the final result after (3)

- **The task `in_reduction` clause**
  - Task participates in a reduction operation (2)

```
#pragma omp task in_reduction(op: list)
{structured-block}
```

≥ v5.0

```
int res = 0;
node_t* node = head;
...
#pragma omp parallel
{
  #pragma omp single
  {
    #pragma omp taskgroup \
                task_reduction(+: res)  (1)
    {
      while (node) {
        #pragma omp task \
                    in_reduction(+: res) \  (2)
                    firstprivate(node)
        {
          res += node->value;
        }
        node = node->next;
      }
    } (3)
  }
}
```

Task Loops

Example: saxpy Kernel with OpenMP task

```
for (int i=0; i<N; ++i) {
   a[i] += b[i] * s;
}
```

```
for (int i=0; i<N; i+=TS) {
   int ub = min(N, i+TS);
   for (int ii=i; ii<ub; ii++) {
      a[ii] += b[ii] * s;
   }
}
```

```
#pragma omp parallel
#pragma omp single
for (int i=0; i<N; i+=TS) {
   int ub = min(N, i+TS);
   #pragma omp task shared(a, b, s)
   for (int ii=i; ii<ub; ii++) {
      a[ii] += b[ii] * s;
   }
}
```

- **Difficult to determine grain**
  - 1 single iteration: to fine
  - whole loop: no parallelism
- **Manually transform the code**
  - blocking techniques
- **Improving programmability**
  - OpenMP `taskloop`

Outer loop iterates across the tiles

Inner loop iterates within a tile

You have to rename all your loop indices!

The taskloop Construct

- **Parallelize a loop using OpenMP tasks**
  - Cut loop into chunks
  - Create a task for each loop chunk

- **C/C++**

```
#pragma omp taskloop [simd] [clause[[,] clause],…]
for-loops
```

- **Fortran**

```
!$omp taskloop[simd] [clause[[,] clause],…]
do-loops
[!$omp end taskloop [simd]]
```

- **Loop iterations are distributed over the tasks.**
- **With `simd` the resulting loop uses SIMD instructions.**

Clauses for the taskloop Construct

- **`taskloop` constructs inherit clauses both from worksharing constructs and the task construct**
  - **`shared`**, **`private`**
  - **`firstprivate`**, **`lastprivate`**
  - **`default`**
  - **`collapse`**
  - **`final`**, **`untied`**, **`mergeable`**
  - **`allocate`**
  - **`in_reduction`** / **`reduction`**

- **`grainsize(grain-size)`**
  - Chunks have at least **`grain-size`** and maximally 2 × **`grain-size`** loop iterations

- **`num_tasks(num-tasks)`**
  - Create `num-tasks` tasks for iterations of the loop

Example: saxpy Kernel with OpenMP taskloop

```
for (int i=0; i<N; ++i) {
  a[i] += b[i] * s;
}
```

manual blocking

```
for (int i=0; i<N; i+=TS) {
   int ub = min(N, i+TS);
   for (int ii=i; ii<ub; ii++) {
      a[ii] += b[ii] * s;
   }
}
```

```
#pragma omp parallel
#pragma omp single
for (int i=0; i<N; i+=TS) {
   int ub = min(N, i+TS);
   #pragma omp task shared(a, b, s)
   for (int ii=i; ii<ub; ii++) {
      a[ii] += b[ii] * s;
   }
}
```

taskloop

```
#pragma omp taskloop grainsize(TS)
for (int i=0; i<N; ++i) {
   a[i] += b[i] * s;
}
```

- **Easier to apply than manual blocking:**
  - Compiler implements mechanical transformation
  - Less error-prone, more productive

Task Dependencies

The depend Clause

C/C++

```
#pragma omp task depend(dependency-type: list)
structured block
```

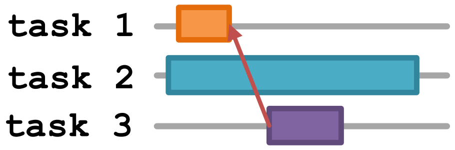
Fortran

```
!$omp task depend(dependency-type: list)
structured block
!$omp end task
```

- **The task dependence is fulfilled when the predecessor task has completed**

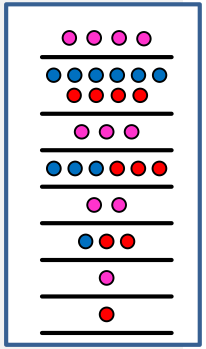
- **`in` dependency-type:**
  The generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an `out` or `inout` clause.
- **`out` and `inout` dependency-type:**
  The generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an `in`, `out`, or `inout` clause.
- **`mutexinoutset` dependency-type:**
  Support mutually exclusive `inout` sets, requires ≥v5.0.
- **The list items in a depend clause may include array sections.**

```
int x = 0;
#pragma omp parallel
#pragma omp single
{
  #pragma omp task depend(in: x)
  std::cout << x << std::endl;

  #pragma omp task
  long_running_task();

  #pragma omp task depend(inout: x)
  x++;
}
```

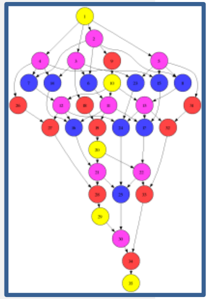


Example: Cholesky Factorization

```
void cholesky(int ts, int nt, double* a[nt][nt]) {
  for (int k = 0; k < nt; k++) {
    // Diagonal Block factorization
    potrf(a[k][k], ts, ts);

    // Triangular systems
    for (int i = k + 1; i < nt; i++) {
      #pragma omp task
      trsm(a[k][k], a[k][i], ts, ts);
    }
    #pragma omp taskwait

    // Update trailing matrix
    for (int i = k + 1; i < nt; i++) {
      for (int j = k + 1; j < i; j++) {
        #pragma omp task
        dgemm(a[k][i], a[k][j], a[j][i],
      }
      #pragma omp task
      syrk(a[k][i], a[i][i], ts, ts);
    }
    #pragma omp taskwait
  }
}
```



```
void cholesky(int ts, int nt, double* a[nt][nt]) {
  for (int k = 0; k < nt; k++) {
    // Diagonal Block factorization
    #pragma omp task depend(inout: a[k][k])
    potrf(a[k][k], ts, ts);

    // Triangular systems
    for (int i = k + 1; i < nt; i++) {
      #pragma omp task depend(in: a[k][k])
                       depend(inout: a[k][i])
      trsm(a[k][k], a[k][i], ts, ts);
    }

    // Update trailing matrix
    for (int i = k + 1; i < nt; i++) {
      for (int j = k + 1; j < i; j++) {
        #pragma omp task depend(inout: a[j][i]) \
                         depend(in: a[k][i], a[k][j])
        dgemm(a[k][i], a[k][j], a[j][i], ts, ts);
      }
      #pragma omp task depend(inout: a[i][i])
                       depend(in: a[k][i])
      syrk(a[k][i], a[i][i], ts, ts);
    }
  }
}
```



© 2010-2023