Hybrid CPU programming with OpenMP and MPI

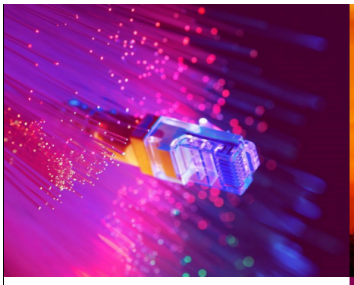
October 4 – 5, 2021

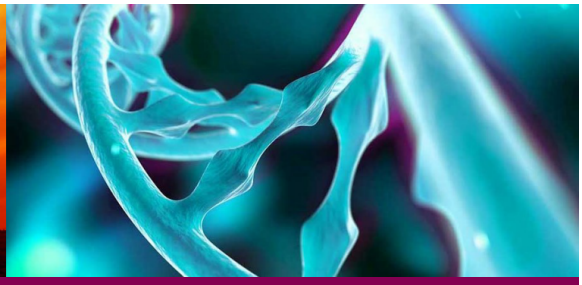
CSC – IT Center for Science Ltd., Espoo

Jussi Enkovaara
Martti Louhivuori

CSC
Training for Brilliant Minds

PRACE





Introduction to hybrid programming

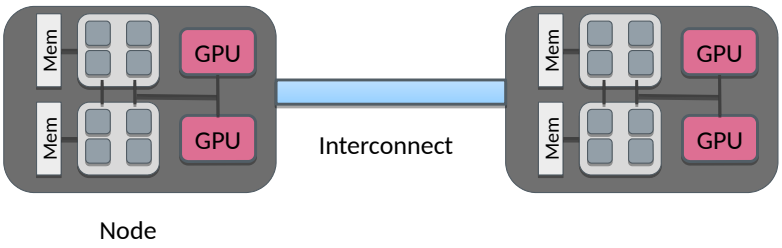
CSC Training, 2021

CSC – Finnish expertise in ICT for research, education and public administration

Introduction

Anatomy of supercomputer

- Supercomputers consist of nodes connected with high-speed network
  - Latency ~1 µs, bandwidth ~200 Gb / s
- A node can contain several multicore CPUS
- Additionally, node can contain one or more accelerators
- Memory within the node is directly usable by all CPU cores

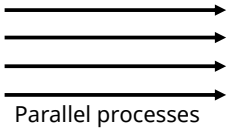


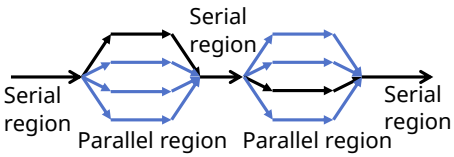
Node

Parallel programming models

- Parallel execution is based on threads or processes (or both) which run at the same time on different CPU cores
- Processes
  - Interaction is based on exchanging messages between processes
  - MPI (Message passing interface)
- Threads
  - Interaction is based on shared memory, i.e. each thread can access directly other threads data
  - OpenMP

Parallel programming models





MPI: Processes

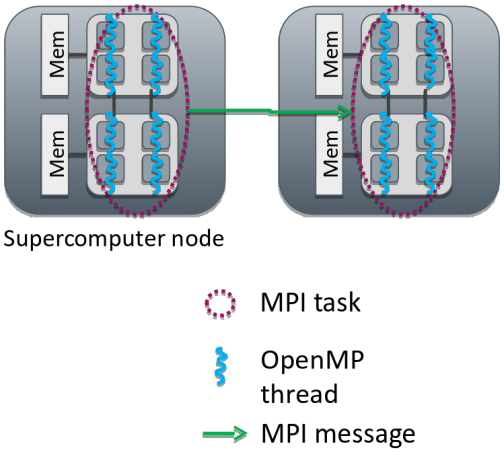
- Independent execution units
- Have their **own** memory space
- MPI launches N processes at application startup
- Works over multiple nodes

OpenMP: Threads

- Threads **share** memory space
- Threads are created and destroyed (parallel regions)
- Limited to a single node

Hybrid programming: Launch threads (OpenMP) *within* processes (MPI)

- Shared memory programming inside a node, message passing between nodes
- Matches well modern supercomputer hardware
- Optimum MPI task per node ratio depends on the application and should always be experimented.



Example: Hybrid hello

```
#include <mpi.h>
#include <omp.h>

int main(int argc, char *argv[]) {
    int my_id, omp_rank;
    int provided, required=MPI_THREAD_FUNNELED;

    MPI_Init_thread(&argc, &argv, required,
                    &provided);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
#pragma omp parallel private(omp_rank)
{
    omp_rank = omp_get_thread_num();
    printf("I'm thread %d in process %d\n",
           omp_rank, my_id);
}
    MPI_Finalize();
}
```

```
$ mpicc -fopenmp hybrid-hello.c -o hybrid-hello
$ srun  --ntasks=2 --cpus-per-task=4
  ./hybrid-hello

I'm thread 0 in process 0
I'm thread 0 in process 1
I'm thread 2 in process 1
I'm thread 3 in process 1
I'm thread 1 in process 1
I'm thread 3 in process 0
I'm thread 1 in process 0
I'm thread 2 in process 0
```


Potential advantages of the hybrid approach

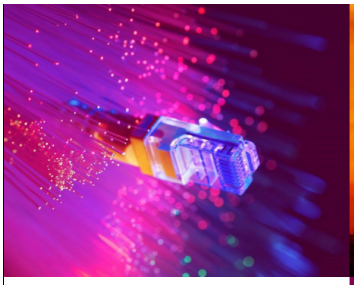
- Fewer MPI processes for a given amount of cores
  - Improved load balance
  - All-to-all communication bottlenecks alleviated
  - Decreased memory consumption if an implementation uses replicated data
- Additional parallelization levels may be available
- Possibility for dedicating threads for different tasks
  - e.g. dedicated communication thread or parallel I/O
- Dynamic parallelization patterns often easier to implement with OpenMP

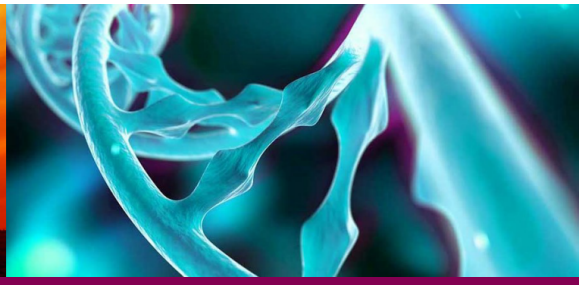
Disadvantages of hybridization

- Increased overhead from thread creation/destruction
- More complicated programming
  - Code readability and maitainability issues
- Thread support in MPI and other libraries needs to be considered

Alternatives to OpenMP within a node

- pthreads (POSIX threads)
- Multithreading support in C++ 11
- Performance portability frameworks (SYCL, Kokkos, Raja)
- Intel Threading Building Blocks



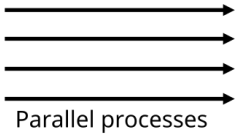


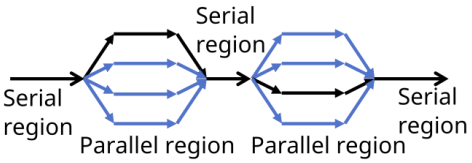
Introduction to OpenMP

CSC Training, 2021


CSC – Finnish expertise in ICT for research, education and public administration

Processes and threads





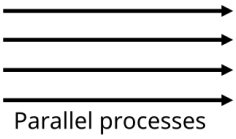
Process

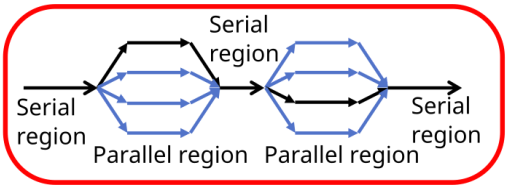
- Independent execution units
- Have their own state information and *own memory* address space

Thread

- A single process may contain multiple threads
- Have their own state information, but *share* the *same memory* address space

Processes and threads





Process

- Long-lived: spawned when parallel program started, killed when program is finished
- Explicit communication between processes

Thread

- Short-lived: created when entering a parallel region, destroyed (joined) when region ends
- Communication through shared memory

OpenMP

What is OpenMP?

- A collection of *compiler directives* and *library routines* , together with a *runtime system*, for **multi-threaded, shared-memory parallelization**
- Fortran 77/9X/03 and C/C++ are supported
- Latest version of the standard is 5.0 (November 2018)
  - Full support for accelerators (GPUs)
  - Support latest versions of C, C++ and Fortran
  - Support for a fully descriptive loop construct
  - and more
- Compiler support for 5.0 is still incomplete
- This course does not discuss any 5.0 specific features

Why would you want to learn OpenMP?

- OpenMP parallelized program can be run on your many-core workstation or on a node of a cluster
- Enables one to parallelize one part of the program at a time
  - Get some speedup with a limited investment in time
  - Efficient and well scaling code still requires effort
- Serial and OpenMP versions can easily coexist
- Hybrid MPI+OpenMP programming

Three components of OpenMP

- Compiler directives, i.e. language extensions
  - Expresses shared memory parallelization
  - Preceded by sentinel, can compile serial version
- Runtime library routines (Intel: libiomp5, GNU: libgomp)
  - Small number of library functions
  - Can be discarded in serial version via conditional compiling
- Environment variables
  - Specify the number of threads, thread affinity etc.

OpenMP directives

- OpenMP directives consist of a *sentinel*, followed by the directive name and optional clauses
- C/C++:

```
#pragma omp directive [clauses]
```

- Fortran:

```
!$omp directive [clauses]
```

- Directives are ignored when code is compiled without OpenMP support

Compiling an OpenMP program

- Compilers that support OpenMP usually require an option (flag) that enables the feature
  - GNU: `-fopenmp`
  - Intel: `-qopenmp`
  - Cray: `-h omp`
    - OpenMP enabled by default, -h noomp disables
  - PGI: `-mp[=nonuma,align,allcores,bind]`

Parallel construct

- Defines a *parallel region*
  - C/C++:

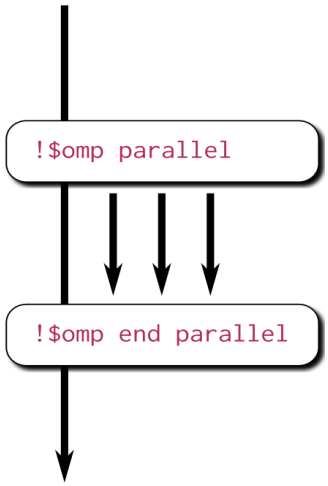
```
#pragma omp parallel [clauses]
   structured block
```

  - Fortran:

```
!$omp parallel [clauses]
   structured block
!$omp end parallel
```

- Prior to it only one thread (main)
- Creates a team of threads: main + workers
- Barrier at the end of the block

SPMD: Single Program Multiple Data



Example: "Hello world" with OpenMP

```
program omp_hello

   write(*,*) "Hello world! -main"
!$omp parallel
   write(*,*) ".. worker reporting for duty."
!$omp end parallel
   write(*,*) "Over and out! -main"

end program omp_hello
```

```
> gfortran -fopenmp omp_hello.F90 -o omp
> OMP_NUM_THREADS=3 ./omp
 Hello world! -main
 .. worker reporting for duty.
 .. worker reporting for duty.
 .. worker reporting for duty.
 Over and out! -main
```

```
#include <stdio.h>

int main(int argc, char* argv[])
{
  printf("Hello world! -main\n");
#pragma omp parallel
  {
    printf(".. worker reporting for duty.\n");
  }
  printf("Over and out! -main\n");
}
```

```
> gcc -fopenmp omp_hello.c -o omp
> OMP_NUM_THREADS=3 ./omp
Hello world! -main
.. worker reporting for duty.
.. worker reporting for duty.
.. worker reporting for duty.
Over and out! -main
```


How to distribute work?

- Each thread executes the same code within the parallel region
- OpenMP provides several constructs for controlling work distribution
  - Loop construct
  - Single/Master construct
  - Sections construct
  - Task construct
  - Workshare construct (Fortran only)
- Thread ID can be queried and used for distributing work manually (similar to MPI rank)

Loop construct

- Directive instructing compiler to share the work of a loop
  - Each thread executes only part of the loop

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

```
!$omp do [clauses]
...
!$omp end do
```

- in C/C++ limited only to "canonical" for-loops. Iterator base loops are also possible in C++
- Construct must reside inside a parallel region
  - Combined construct with omp parallel:

    `#pragma omp parallel for` / `!$omp parallel do`

Loop construct

```
!$omp parallel
!$omp do
  do i = 1, n
     z(i) = x(i) + y(i)
  end do
!$omp end do
!$omp end parallel
```

```
#pragma omp parallel
{
  #pragma omp for
  for (i=0; i < n; i++)
    z[i] = x[i] + y[i];
}
```

Workshare directive (Fortran only)

- In Fortran many array operations can be done conveniently with array syntax, *i.e.* without explicit loops
  - Array assignments, forall and where statements *etc.*
- The `workshare` directive divides the execution of the enclosed structured block into separate units of work, each of which is executed only once

```
real :: a(n,n), b(n,n), c(n,n) d(n,n)
...
!$omp parallel
  !$omp workshare
     c = a * b
     d = a + b
  !$omp end workshare
!$omp end parallel
```

- Note that performance may be bad in some compilers, in particular with Intel

Sections construct

- Sections construct enables parallel execution of independent tasks
- Each section is run by a single thread

```
#pragma omp sections [clauses]
{
  #pragma omp section
  task1()
  #pragma omp section
  task2()
  #pragma omp section
  task3()
}
```

```
!$omp sections [clauses]
  !$omp section
  call task1()
  !$omp section
  call task2()
  !$omp section
  call task3()
!$omp end sections
```

- If nested parallelism is supported, "tasks" can in principle contain parallel regions
  - Performance may not be optimal
- Often, OpenMP **task** construct is more suitable

Summary

- OpenMP can be used with compiler directives
  - Ignored when code is build without OpenMP
- Threads are launched within **parallel** regions
- `for`/`do` loops can be parallelized easily with loop construct

OpenMP runtime library and environment variables

OpenMP runtime library and environment variables

- OpenMP provides several means to interact with the execution environment. These operations include e.g.
  - Setting the number of threads for parallel regions
  - Requesting the number of CPUs
  - Changing the default scheduling for work-sharing clauses
- Improves portability of OpenMP programs between different architectures (number of CPUs, etc.)

Environment variables

- OpenMP standard defines a set of environment variables that all implementations have to support
- The environment variables are set before the program execution and they are read during program start-up
  - Changing them during the execution has no effect
- We have already used `OMP_NUM_THREADS`

Some useful environment variables

Variable	Action
OMP_NUM_THREADS	Number of threads to use
OMP_PROC_BIND	Bind threads to CPUs
OMP_PLACES	Specify the bindings between threads and CPUs
OMP_DISPLAY_ENV	Print the current OpenMP environment info on stderr

Runtime functions

- Runtime functions can be used either to read the settings or to set (override) the values
- Function definitions are in
  - C/C++ header file `omp.h`
  - `omp_lib` Fortran module (`omp_lib.h` header in some implementations)
- Two useful routines for finding out thread ID and number of threads:
  - `omp_get_thread_num()`
  - `omp_get_num_threads()`

OpenMP conditional compilation

- Conditional compilation with _OPENMP macro:

```
#ifdef _OPENMP
    OpenMP specific code with, e.g., library calls
#else
    Code without OpenMP
#endif
```


Example: Hello world with OpenMP

```
program hello
  use omp_lib
  integer :: omp_rank
!$omp parallel
#ifdef _OPENMP
  omp_rank = omp_get_thread_num()
#else
  omp_rank = 0
#endif
  print *, 'Hello world! by &
        thread ', omp_rank
!$omp end parallel
end program hello
```

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char argv[]) {
  int omp_rank;
#pragma omp parallel
  {
#ifdef _OPENMP
    omp_rank = omp_get_thread_num();
#else
    omp_rank = 0;
#endif
    printf("Hello world! by thread %d\n",
           omp_rank);
  }
}
```

Parallel regions and data sharing

How do the threads interact?

- Because of the shared address space threads can interact using *shared variables*
- Threads often need some *private work space* together with shared variables
  - for example the index variable of a loop
- If threads write to the same shared variable a **race condition** appears
  - Undefined end result
- Visibility of different variables is defined using *data-sharing clauses* in the parallel region definition

Race condition in Hello world

```
#include <stdio.h>
#include <omp.h>
#include <unistd.h>
int main(int argc, char argv[]) {
  int omp_rank;
#pragma omp parallel
  {
    omp_rank = omp_get_thread_num();
    sleep(1);
    printf("Hello world! by thread %d\n", omp_rank);
  }
}
```

- All the threads write out the same thread number
- The result varies between successive runs

omp parallel: data-sharing clauses

- **private(list)**
  - Private variables are stored in the private stack of each thread
  - Undefined initial value
  - Undefined value after parallel region
- **firstprivate(list)**
  - Same as private variable, but with an initial value that is the same as the original objects defined outside the parallel region

omp parallel: data-sharing clauses

- **shared(list)**
  - All threads can write to, and read from a shared variable
  - Variables are shared by default
- **default(private/shared/none)**
  - Sets default for variables to be shared, private or not defined
  - In C/C++ default(private) is not allowed
  - default(none) can be useful for debugging as each variable has to be defined manually

Default behaviour

- Most variables are *shared* by default
  - Global variables are shared among threads
    - C: static variables, file scope variables
    - Fortran: save and module variables, common blocks
    - `threadprivate(list)` can be used to make a private copy
- Private by default:
  - Local variables of functions called from parallel region
  - Variables declared within a block (C/C++)
- Good programming practice: declare all variables either shared or private

Hello world without a race condition

```c
#include <stdio.h>
#include <omp.h>
#include <unistd.h>
int main(int argc, char argv[]) {
  int omp_rank;
#pragma omp parallel private(omp_rank)
  {
    omp_rank = omp_get_thread_num();
    sleep(1);
    printf("Hello world! by thread %d\n", omp_rank);
  }
}
```


Data sharing example

main.c

```c
int A[5];  // shared

int main(void) {
  int B[2];  // shared
#pragma omp parallel
  {
    float c;  // private
    do_things(B);
    ...
  }
  return 0;
}
```

kernel.c

```c
extern int A[5];  // shared

void do_things(int *var) {
  double wrk[10];  // private
  static int status;
  ...
}
```

Summary

- OpenMP runtime behavior can be controlled using environment variables
- OpenMP provides also library routines
- Visibility of variables in parallel region can be specified with data sharing clauses
  - **private** : each thread works with their own variable
  - **shared** : all threads can write to and read from a shared variable
- Race conditions possible when writing to shared variables
- Avoiding race conditions is key to correctly functioning OpenMP programs

OpenMP reductions and execution control

CSC Training, 2021

CSC – Finnish expertise in ICT for research, education and public administration

Processing math: 100%

OpenMP reductions

Processing math: 100%

Race condition in reduction

- Race conditions take place when multiple threads read and write a variable simultaneously, for example:

```
asum = 0.0d0
!$omp parallel do shared(x,y,n,asum) private(i)
  do i = 1, n
     asum = asum + x(i)*y(i)
  end do
!$omp end parallel do
```

- Random results depending on the order the threads access **asum**
- We need some mechanism to control the access

Reductions

- Summing elements of array is an example of reduction operation

$$S = \sum_{j=1}^{N} A_j = \sum_{j=1}^{\frac{N}{2}} A_j + \sum_{j=\frac{N}{2}+1}^{N} A_j = B_1 + B_2 = \sum_{j=1}^{2} B_j$$

- OpenMP provides support for common reductions within parallel regions and loops with the reduction clause

Reduction clause

`reduction(operator:list)`

Performs reduction on the (scalar) variables in list

- Private reduction variable is created for each thread's partial result
- Private reduction variable is initialized to operator's initial value
- After parallel region the reduction operation is applied to private variables and result is aggregated to the shared variable

Reduction operators in C/C++

Operator	Initial value
+	0
-	0
*	1
&&	1
||	0

Bitwise Operator	Initial value
&	~0
|	0
^	0

Reduction operators in Fortran

Operator	Initial value
+	0
-	0
*	1
max	least
min	largest
.and.	.true.
.or.	.false.
.eqv.	.true.
.neqv.	.false.

Bitwise Operator	Initial value
.iand.	all bits on
.ior.	0
.ieor.	0

Race condition avoided with reduction clause

```
!$omp parallel do shared(x,y,n) private(i) reduction(+:asum)
  do i = 1, n
     asum = asum + x(i)*y(i)
  end do
!$omp end parallel do
```

```
#pragma omp parallel for shared(x,y,n) private(i) reduction(+:asum)
for(i=0; i < n; i++) {
  asum = asum + x[i] * y[i];
}
```

OpenMP execution controls

Execution controls

- Sometimes a part of parallel region should be executed only by the master thread or by a single thread at time
  - IO, initializations, updating global values, etc.
  - Remember the synchronization!
- OpenMP provides clauses for controlling the execution of code blocks

Failing example

```
#pragma omp parallel shared(global_counter) private(tnum, delay, rem)
{
  tnum = omp_get_thread_num();
  delay.tv_sec = 0;
  delay.tv_nsec = 10000 * tnum;
  do {
    printf("This is iteration %i\n", global_counter);
    global_counter++; /* Race condition! */
    nanosleep(&delay, &rem);
  } while(global_counter < 10);
}
```


Execution control constructs

`barrier`

- When a thread reaches a barrier it only continues after all the threads in the same thread team have reached it
  - Each barrier must be encountered by all threads in a team, or none at all
  - The sequence of work-sharing regions and barrier regions encountered must be same for all threads in team
- Implicit barrier at the end of: do, `parallel`, `single`, `workshare`

Execution control constructs

`master`

- Specifies a region that should be executed only by the master thread
- Note that there is no implicit barrier at end

`single`

- Specifies that a regions should be executed only by a single (arbitrary) thread
- Other threads wait (implicit barrier)

Execution control constructs

`critical[(name)]`

name Optional name specifies global identifier for critical section

- A section that is executed by only one thread at a time
- Unnamed critical sections are treated as the same section

Execution control constructs

`atomic`

- Strictly limited construct to update a single value, can not be applied to code blocks
- Only guarantees atomic update, does not protect function calls
- Can be faster on hardware platforms that support atomic updates

Example: reduction using critical section

```
!$OMP PARALLEL SHARED(x,y,n,asum) PRIVATE(i, psum)
  psum = 0.0d
  !$OMP DO
  do i = 1, n
    psum = psum + x(i)*y(i)
  end do
  !$OMP END DO
  !$OMP CRITICAL(dosum)
  asum = asum + psum
  !$OMP END CRITICAL(dosum)
!$OMP END PARALLEL
```

Example: initialization and output

```
int total = 0;
#pragma omp parallel shared(total) private(sum,new)
{
#pragma omp single
  initialise();

  int new, sum = 0;
  do {
    new = compute_something();
    sum += new;
  } while (new);
#pragma omp barrier
#pragma omp critical(addup)
  total += sum;
#pragma omp master
  printf("Grand total is: %5.2f\n", total);
}
```


Summary

- Several parallel reduction operators available via `reduction` clause
- OpenMP has many synchronization pragmas
  - Critical sections
  - Atomic
  - Single and Master
  - And some that we did not present

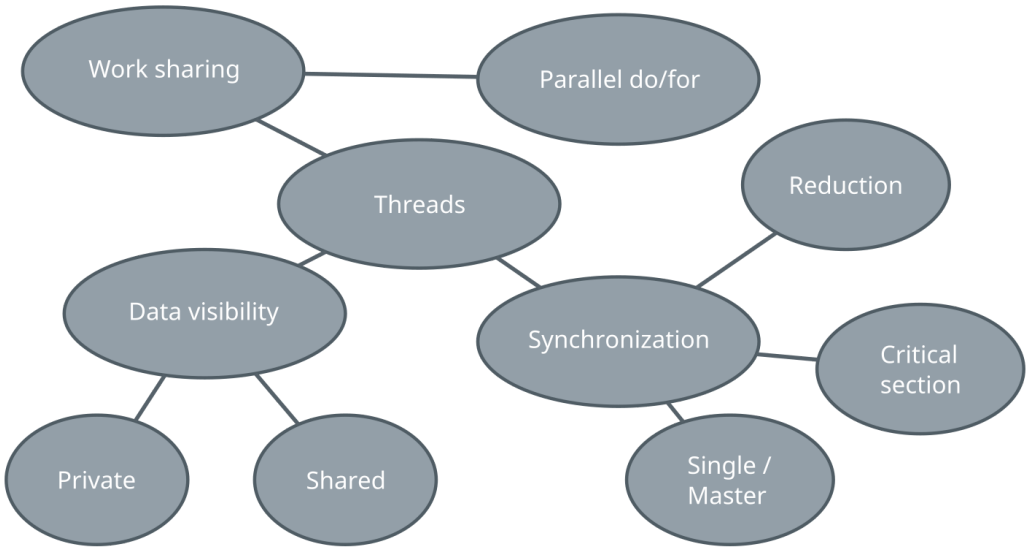
OpenMP programming best practices

- Maximise parallel regions
  - Reduce fork-join overhead, e.g. combine multiple parallel loops into one large parallel region
  - Potential for better cache re-usage
- Parallelise outermost loops if possible
  - Move PARALLEL DO construct outside of inner loops
- Reduce access to shared data
  - Possibly make small arrays private
- Use more tasks than threads
  - Too large number of tasks leads to performance loss

OpenMP summary

- OpenMP is an API for thread-based parallelisation
  - Compiler directives, runtime API, environment variables
  - Relatively easy to get started but specially efficient and/or real-world parallelisation non-trivial
- Features touched in this intro
  - Parallel regions, data-sharing attributes
  - Work-sharing, reductions, execution control

OpenMP summary



Web resources

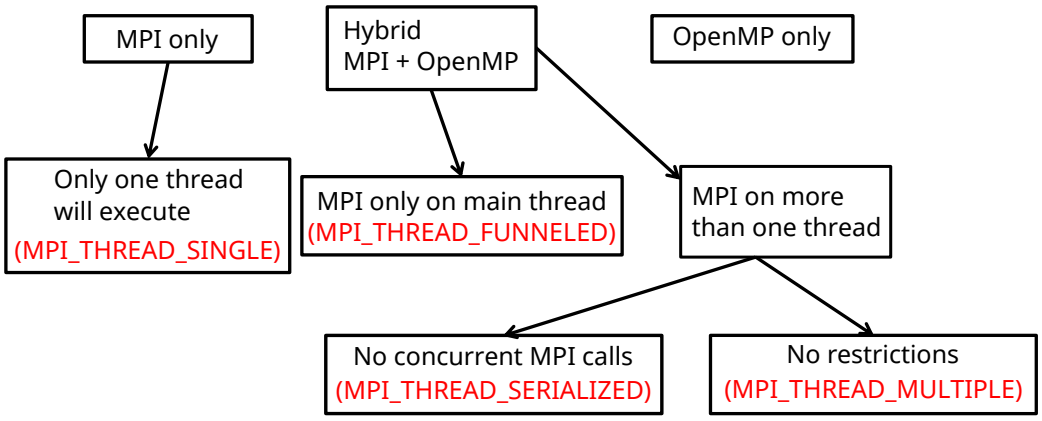
- OpenMP homepage: http://openmp.org/
- Good online tutorial: https://computing.llnl.gov/tutorials/openMP/
- More online tutorials: http://openmp.org/wp/resources/#Tutorials

Using MPI with multithreading

CSC Training, 2021

CSC – Finnish expertise in ICT for research, education and public administration

Thread support in MPI



Thread safe initialization

`MPI_Init_thread(required, provided)`

`argc, argv`
Command line arguments in C

`required`
Required thread safety level

`provided`
Supported thread safety level

`error`
Error value; in C/C++ it's the return value of the function, and in Fortran an additional output parameter

- Pre-defined integer constants: `MPI_THREAD_SINGLE < MPI_THREAD_FUNNELED < MPI_THREAD_SERIALIZED < MPI_THREAD_MULTIPLE`

Hybrid programming styles: fine/coarse grained

- Fine-grained
  - Use **omp parallel do/for** on the most intensive loops
  - Possible to hybridize an existing MPI code with little effort and in parts
- Coarse-grained
  - Use OpenMP threads to replace MPI tasks
  - Whole (or most of) program within the same parallel region
  - More likely to scale over the whole node, enables all cores to communicate (if supported by MPI implementation)

Multiple thread communication

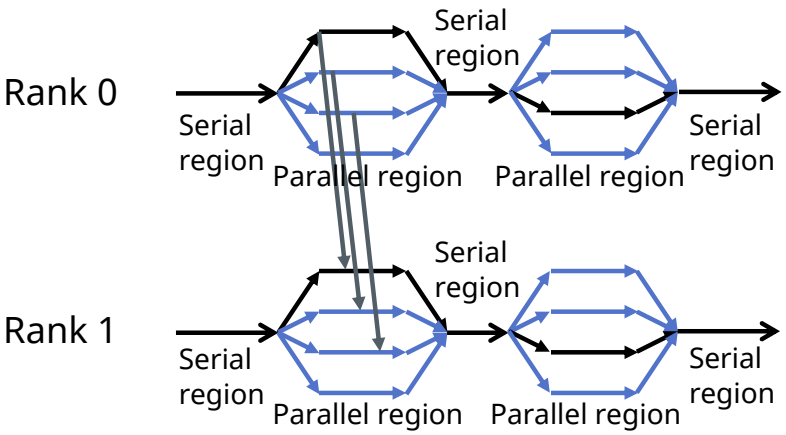
- Hybrid programming is relatively straightforward in cases where communication is by only single thread at time
- With the so called multiple mode all threads can make MPI calls independently

```
int required=MPI_THREAD_MULTIPLE, provided;
MPI_Init_thread(&argc, &argv, required, &provided)
```

- When multiple threads communicate, the sending and receiving threads normally need to match
  - Thread-specific tags
  - Thread-specific communicators

Thread-specific tags

- In point-to-point communication the thread ID can be used to generate a tag that guides the messages to the correct thread



Thread-specific tags

- In point-to-point communication the thread ID can be used to generate a tag that guides the messages to the correct thread

```
!$omp parallel private(tid, tidtag, ierr)
tid = omp_get_thread_num()
tidtag = 2**10 + tid

! mpi communication to the corresponding thread on another process
call mpi_sendrecv(senddata, n, mpi_real, pid, tidtag, &
                  recvdata, n, mpi_real, pid, tidtag, &
                  mpi_comm_world, stat, ierr)

!$omp end parallel
```


Collective operations in the multiple mode

- MPI standard allows multiple threads to call collectives simultaneously
  - Programmer must ensure that the same communicator is not being concurrently used by two different collective communication calls at the same process
- In most cases, even with `MPI_THREAD_MULTIPLE` it is beneficial to perform the collective communication from a single thread (usually the master thread)
- Note that MPI collective communication calls do not guarantee synchronization of the thread order

Thread-specific communicators

- Collective calls do not have tag arguments
- Instead, one can generate thread-specific *communicators*

```
! total number of openmp threads
nthr = omp_get_max_threads()
allocate(tcomm(nthr))

! split the communicator
do thrid=1,nthr
    col = thrid
    call mpi_comm_split(mpi_comm_world, col, procid, tcomm(thrid), ierr)
end do

!$omp parallel private(tid, ierr)
tid = omp_get_thread_num() + 1
call mpi_bcast(..., tcomm(tid))
!$omp end parallel
```


MPI thread support levels

- Modern MPI libraries support all threading levels
  - OpenMPI: Build time configuration, check with

    ```
    ompi_info | grep 'Thread support'
    ```

  - Intel MPI: When compiling with `-qopenmp` a thread safe version of the MPI library is automatically used
  - Cray MPI: Set `MPICH_MAX_THREAD_SAFETY` environment variable to `single`, `funneled`, `serialized`, or `multiple` to select the threading level
- Note that using `MPI_THREAD_MULTIPLE` requires the MPI library to internally lock some data structures to avoid race conditions
  - May result in additional overhead in MPI calls

Summary

- Multiple threads may make MPI calls simultaneously
- Thread specific tags and/or communicators
- For collectives it is often better to use a single thread for communication

Thread and process affinity

Thread and process affinity

- Normally, operating system can run threads and processes in any logical core
- Operating system may even move running task from one core to another
  - Can be beneficial for load balancing
  - For HPC workloads often detrimental as private caches get invalidated and NUMA locality is lost
- User can control where tasks are run via affinity masks
  - Task can be *pinned* to a specific logical core or set of logical cores

Controlling affinity

- Affinity for a *process* can be set with a `numactl` command
  - Limit the process to logical cores 0,3,7:
    ```
    numactl --physcpubind=0,3,7 ./my_exe
    ```
  - Threads "inherit" the affinity of their parent process
- Affinity of a thread can be set with OpenMP environment variables
  - `OMP_PLACES=[threads,cores,sockets]`
  - `OMP_PROC_BIND=[true, close, spread, master]`
- OpenMP runtime prints the affinity with `OMP_DISPLAY_AFFINITY=true`

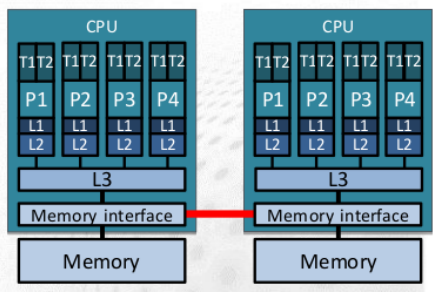
Controlling affinity

```
export OMP_AFFINITY_FORMAT="Thread %0.3n affinity %A"
export OMP_DISPLAY_AFFINITY=true
./test
Thread 000 affinity 0-7
Thread 001 affinity 0-7
Thread 002 affinity 0-7
Thread 003 affinity 0-7
```

```
OMP_PLACES=cores ./test
Thread 000 affinity 0,4
Thread 001 affinity 1,5
Thread 002 affinity 2,6
Thread 003 affinity 3,7
```


Non-uniform memory access

- A node can have multiple sockets with memory attached to each socket
- Non Uniform Memory Access (NUMA)
  - All memory within a node is accessible, but latencies and bandwidths vary
- Hardware needs to maintain cahce coherency also between different NUMA nodes (ccNUMA)



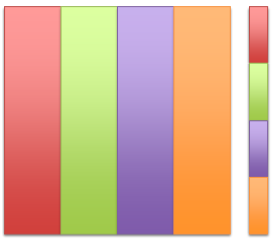
First touch policy

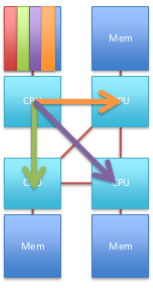
- Modern operating systems use virtual memory
- The OS typically optimizes memory allocations
  - `malloc()` does not allocate the memory directly
  - Memory management only "knows" about the allocation, but no memory pages are made available
  - At first memory access (write), the OS physically allocates the corresponding page (First touch policy)
- On NUMA systems this might lead to performance issues in threaded or multi-process applications

NUMA aware initialization

- No NUMA awareness

```
// Initialize data
for (int i=0; i < N; i++)
   data[i] = ...
...
// Perform work
#pragma omp parallel for
for (int i=0; i < N; i++)
   process(data[i])
```

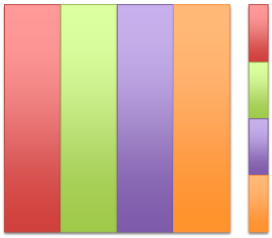


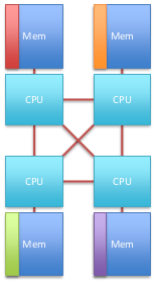


NUMA aware initialization

- With NUMA awareness

```
// Initialize data
#pragma omp parallel for
for (int i=0; i < N; i++)
   data[i] = ...
...
// Perform work
#pragma omp parallel for
for (int i=0; i < N; i++)
   process(data[i])
```

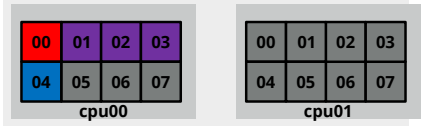




MPI+OpenMP thread affinity

- MPI library must be aware of the underlying OpenMP for correct allocation of resources
  - Oversubscription of CPU cores may cause significant performance penalty
- Additional complexity from batch job schedulers
- Heavily dependent on the platform used!

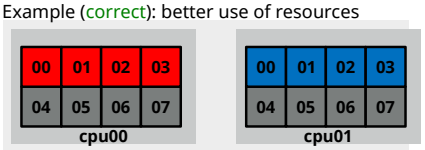
Example (incorrect): oversubscription of resources



MPI task 0: cpu00:00, cpu00:01, cpu00:02, cpu00:03

MPI task 1: cpu00:00, cpu00:01, cpu00:02, cpu00:03

Example (correct): better use of resources



MPI task 0: cpu00:00, cpu00:01, cpu00:02, cpu00:03

MPI task 1: cpu00:00, cpu00:01, cpu00:02, cpu00:03

Slurm configuration at CSC

- Within a node, `--tasks-per-node` MPI tasks are spread `--cpus-per-task` apart
- Threads within a MPI tasks have the affinity mask for the corresponging `--cpus-per-task` cores

```
export OMP_AFFINITY_FORMAT="Process %P thread %0.3n affinity %A"
export OMP_DISPLAY_AFFINITY=true
srun ... --tasks-per-node=2 --cpus-per-task=4 ./test
Process 250545 thread 000 affinity 0-3
...
Process 250546 thread 000 affinity 4-7
...
```

- Slurm configurations in other HPC centers can be very different
  - Always experiment before production calculations!

Summary

- Performance of HPC applications is often improved when processes and threads are pinned to CPU cores
- MPI and batch system configurations may affect the affinity
  - very system dependent, try to always investigate

OpenMP

Programming OpenMP

Tasking Introduction

OpenMP

Sudoko for Lazy Computer Scientists

- Lets solve Sudoku puzzles with brute multi-core force

	6						8	11			15	14			16
15	11				16	14				12			6		
13		9	12					3	16	14		15	11	10	
2		16		11		15	10	1							
	15	11	10			16	2	13	8	9	12				
12	13			4	1	5	6	2	3					11	10
5		6	1	12		9		15	11	10	7	16			3
	2				10		11	6		5			13		9
10	7	15	11	16				12	13						6
9						1			2		16	10			11
1		4	6	9	13			7		11		3	16		
16	14			7		10	15	4	6	1				13	8
11	10		15				16	9	12	13			1	5	4
		12		1	4	6		16				11	10		
		5		8	12	13		10			11	2			14
3	16			10			7			6				12	

- (1) Search an empty field
- (2) Try all numbers:
  - (2 a) Check Sudoku
    - If invalid: skip
    - If valid: Go to next field
- Wait for completion

Parallel Brute-force Sudoku

- This parallel algorithm finds all valid solutions

	6						8	11			15	14			16
15	11				16	14				12			6		
13		9	12					3	16	14		15	11	10	
2		16		11		15	10	1							
	15	11	10			16	2	13	8	9	12				
12	13			4	1	5	6	2	3					11	10
5		6	1	12		9		15	11	10	7	16			3
	2				10		11	6		5			13		9
10	7	15	11	16				12	13						6
9						1			2		16	10			11
1		4	6	9	13			7		11		3	16		
16	14			7		10	15	4	6	1				13	8
11	10		15				16	9	12	13			1	5	4
		12		1	4	6		16				11	10		
		5		8	12	13		10			11	2			14
3	16			10			7			6				12	

- (1) Search an empty fie

first call contained in a
`#pragma omp parallel`
`#pragma omp single`
such that one tasks starts the execution of the algorithm

- (2) Try all numbers:
  - (2 a) Check Sudoku
    - If invalid: skip
    - If valid: Go to ne

`#pragma omp task`
needs to work on a new copy of the Sudoku board

- Wait for completion

`#pragma omp taskwait`
wait for all child tasks

Tasking Overview

What is a task in OpenMP?

- Tasks are work units whose execution
  - → may be deferred or…
  - → … can be executed immediately
- Tasks are composed of
  - → **code** to execute, a **data** environment (initialized at creation time), internal **control** variables (ICVs)
- Tasks are created…
  - … when reaching a parallel region → implicit tasks are created (per thread)
  - … when encountering a task construct → explicit task is created
  - … when encountering a taskloop construct → explicit tasks per chunk are created
  - … when encountering a target construct → target task is created

Tasking execution model

- Supports unstructured parallelism
  - → unbounded loops

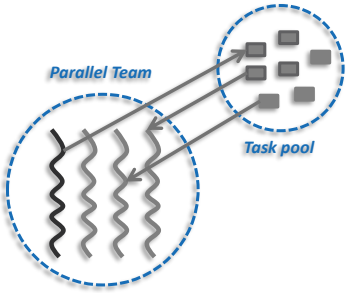
```
while ( <expr> ) {
   ...
}
```

  - → recursive functions

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

- Example (unstructured parallelism)

```
#pragma omp parallel
#pragma omp master
while (elem != NULL) {
   #pragma omp task
      compute(elem);
   elem = elem->next;
}
```



- Several scenarios are possible:
  - → single creator, multiple creators, nested tasks (tasks & WS)
- All threads in the team are candidates to execute tasks

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
```

- Where clause is one of:

→ private(list) → firstprivate(list) → shared(list) → default(shared | none) → in_reduction(r-id: list)	Data Environment
→ allocate([allocator:] list) → detach(event-handler)	Miscellaneous
→ if(scalar-expression) → mergeable → final(scalar-expression)	Cutoff Strategies
→ depend(dep-type: list)	Synchronization
→ untied → priority(priority-value) → affinity(list)	Task Scheduling

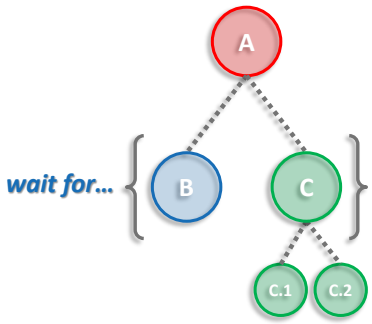
Task synchronization: taskwait directive

- The taskwait directive (shallow task synchronization)
  - → It is a stand-alone directive

```
#pragma omp taskwait
```

  - → wait on the completion of child tasks of the current task; just direct children, not all descendant tasks; includes an implicit task scheduling point (TSP)

```
#pragma omp parallel
#pragma omp single
{
   #pragma omp task                          :A
   {
      #pragma omp task                       :B
      { … }
      #pragma omp task                       :C
      { … #C.1; #C.2; …}
      #pragma omp taskwait
   }
} // implicit barrier will wait for C.x
```



Task synchronization: barrier semantics

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

```
#pragma omp barrier
```

  - → And all other implicit barriers at parallel, sections, for, single, etc…

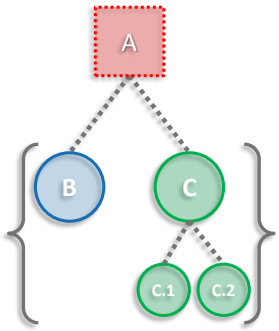
Task synchronization: taskgroup construct

- The taskgroup construct (deep task synchronization)
  - → attached to a structured block; completion of all descendants of the current task; TSP at the end

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

  - → where clause (could only be): reduction(reduction-identifier: list-items)

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

   } // end of taskgroup
}
```



Tasking illustrated

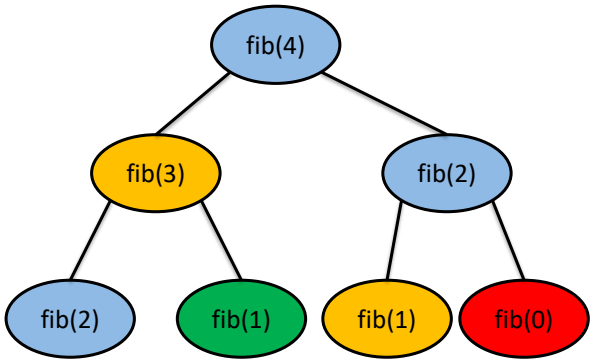
Fibonacci illustrated

```
 1  int main(int argc,
 2           char* argv[])
 3  {
 4      [...]
 5      #pragma omp parallel
 6      {
 7          #pragma omp single
 8         {
 9             fib(input);
10         }
11      }
12      [...]
13  }
```

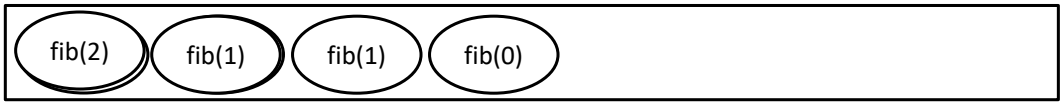
```
14  int fib(int n)   {
15      if (n < 2) return n;
16      int x, y;
17      #pragma omp task shared(x)
18      {
19          x = fib(n - 1);
20      }
21      #pragma omp task shared(y)
22      {
23          y = fib(n - 2);
24      }
25      #pragma omp taskwait
26          return x+y;
27  }
```

- Only one Task / Thread enters fib() from main(), it is responsible for creating the two initial work tasks
- Taskwait is required, as otherwise x and y would get lost

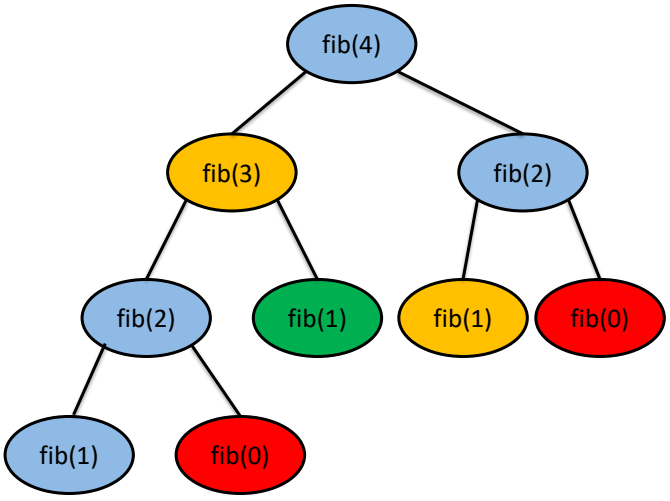
- T1 enters fib(4)
- T1 creates tasks for fib(3) and fib(2)
- T1 and T2 execute tasks from the queue
- T1 and T2 create 4 new tasks
- T1 - T4 execute tasks



Task Queue



- T1 enters fib(4)
- T1 creates tasks for fib(3) and fib(2)
- T1 and T2 execute tasks from the queue
- T1 and T2 create 4 new tasks
- T1 - T4 execute tasks
- …



Data Environment

Explicit data-sharing clauses

- Explicit data-sharing clauses (shared, private and firstprivate)

```
#pragma omp task shared(a)
{
  // Scope of a: shared
}
```

```
#pragma omp task private(b)
{
  // Scope of b: private
}
```

```
#pragma omp task firstprivate(c)
{
  // Scope of c: firstprivate
}
```

- If **default** clause present, what the clause says
  - → shared: data which is not explicitly included in any other data sharing clause will be **shared**
  - → none: compiler will issue an error if the attribute is not explicitly set by the programmer (very useful!!!)

```
#pragma omp task default(shared)
{
 // Scope of all the references, not explicitly
 // included in any other data sharing clause,
 // and with no pre-determined attribute: shared
}
```

```
#pragma omp task default(none)
{
 // Compiler will force to specify the scope for
 // every single variable referenced in the context
}
```

Hint: Use default(none) to be forced to think about every variable if you do not see clearly.

Pre-determined data-sharing attributes

- threadprivate variables are threadprivate **(1)**
- dynamic storage duration objects are shared (malloc, new,…) **(2)**
- static data members are shared **(3)**
- variables declared inside the construct
  - → static storage duration variables are shared **(4)**
  - → automatic storage duration variables are private **(5)**
- the loop iteration variable(s)…

```
#pragma omp task
{
   int x = MN;
   // Scope of x: private
}
```
5

```
#pragma omp task
{
   static int y;
   // Scope of y: shared
}
```
4

```
int A[SIZE];
#pragma omp threadprivate(A)

// ...
#pragma omp task
{
  // A: threadprivate
}
```
1

```
int *p;

p = malloc(sizeof(float)*SIZE);

#pragma omp task
{
   // *p: shared
}
```
2

```
void foo(void){
   static int s = MN;
}

#pragma omp task
{
   foo(); // s@foo(): shared
}
```
3

Implicit data-sharing attributes (in-practice)

- Implicit data-sharing rules for the task region
  - → the **shared** attribute is lexically inherited
  - → in any other case the variable is **firstprivate**
  - → Pre-determined rules (could not change)
  - → Explicit data-sharing clauses (+ default)
  - → Implicit data-sharing rules
- (in-practice) variable values within the task:
  - → value of a: 1
  - → value of b: x // undefined (undefined in parallel)
  - → value of c: 3
  - → value of d: 4
  - → value of e: 5

```
int a = 1;
void foo() {
   int b = 2, c = 3;
   #pragma omp parallel private(b)
   {
      int d = 4;
      #pragma omp task
      {
         int e = 5;
         // Scope of a:
         // Scope of b:
         // Scope of c:
         // Scope of d:
         // Scope of e:
      }
   }
}
```

Task reductions (using taskgroup)

- 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 [participating]

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

  - → Task participates in a reduction operation [2]

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


Task reductions (+ modifiers)

- Reduction modifiers
  - → Former reductions clauses have been extended
  - → task modifier allows to express task reductions
  - → Registering a new task reduction [1]
  - → Implicit tasks participate in the reduction [2]
  - → Compute final result after [4]
- The (task) in_reduction clause [participating]

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

  - → Task participates in a reduction operation [3]

```
int res = 0;
node_t* node = NULL;
...
#pragma omp parallel reduction(task,+: res)
{ // [1][2]
  #pragma omp single
  {
    #pragma omp taskgroup
    {
      while (node) {
        #pragma omp task in_reduction(+: res) \
                firstprivate(node)
        { // [3]
          res += node->value;
        }
        node = node->next;
      }
    }
  }
} // [4]
```

The `taskloop` Construct

Tasking use case: saxpy (taskloop)

```
for ( i = 0; i<SIZE; i+=1) {
   A[i]=A[i]*B[i]*S;
}
```

```
for ( i = 0; i<SIZE; i+=TS) {
   UB = SIZE < (i+TS)?SIZE:i+TS;
   for ( ii=i; ii<UB; ii++) {
      A[ii]=A[ii]*B[ii]*S;
   }
}
```

```
#pragma omp parallel
#pragma omp single
for ( i = 0; i<SIZE; i+=TS) {
   UB = SIZE < (i+TS)?SIZE:i+TS;
   #pragma omp task private(ii) \
    firstprivate(i,UB) shared(S,A,B)
   for ( ii=i; ii<UB; ii++) {
      A[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

```
#pragma omp taskloop grainsize(TS)
for ( i = 0; i<SIZE; i+=1) {
   A[i]=A[i]*B[i]*S;
}
```

  - → Hiding the internal details
  - → Grain size ~ Tile size (TS) → but implementation decides exact grain size

The taskloop Construct

- Task generating construct: decompose a loop into chunks, create a task for each loop chunk

```
#pragma omp taskloop [clause[[,] clause]…]
{structured-for-loops}
```

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

- Where clause is one of:

→ shared(list) → private(list) → firstprivate(list) → lastprivate(list) → default(sh | *pr* | *fp* | none) → reduction(r-id: list) → in_reduction(r-id: list)	Data Environment
→ grainsize(grain-size) → num_tasks(num-tasks)	Chunks/Grain
→ if(scalar-expression) → final(scalar-expression) → mergeable	Cutoff Strategies
→ untied → priority(priority-value)	Scheduler (R/H)
→ collapse(n) → nogroup → allocate([allocator:] list)	Miscellaneous

Worksharing vs. taskloop constructs (1/2)

```
subroutine worksharing
    integer :: x
    integer :: i
    integer, parameter :: T = 16
    integer, parameter :: N = 1024

    x = 0
!$omp parallel shared(x) num_threads(T)

!$omp do
    do i = 1,N
!$omp atomic
        x = x + 1
!$omp end atomic
    end do
!$omp end do

!$omp end parallel
    write (*,'(A,I0)') 'x = ', x
end subroutine
```

Result: x = 1024

```
subroutine taskloop
    integer :: x
    integer :: i
    integer, parameter :: T = 16
    integer, parameter :: N = 1024

    x = 0
!$omp parallel shared(x) num_threads(T)

!$omp taskloop
    do i = 1,N
!$omp atomic
        x = x + 1
!$omp end atomic
    end do
!$omp end taskloop

!$omp end parallel
    write (*,'(A,I0)') 'x = ', x
end subroutine
```

Result: x = 16384

Worksharing vs. taskloop constructs (2/2)

```
subroutine worksharing
    integer :: x
    integer :: i
    integer, parameter :: T = 16
    integer, parameter :: N = 1024

    x = 0
!$omp parallel shared(x) num_threads(T)

!$omp do
    do i = 1,N
!$omp atomic
        x = x + 1
!$omp end atomic
    end do
!$omp end do

!$omp end parallel
    write (*,'(A,I0)') 'x = ', x
end subroutine
```

Result: x = 1024

```
subroutine taskloop
    integer :: x
    integer :: i
    integer, parameter :: T = 16
    integer, parameter :: N = 1024

    x = 0
!$omp parallel shared(x) num_threads(T)
!$omp single
!$omp taskloop
    do i = 1,N
!$omp atomic
        x = x + 1
!$omp end atomic
    end do
!$omp end taskloop
!$omp end single
!$omp end parallel
    write (*,'(A,I0)') 'x = ', x
end subroutine
```

Result: x = 1024

Taskloop decomposition approaches

- Clause: grainsize(grain-size)
  - → Chunks have at least grain-size iterations
  - → Chunks have maximum 2x grain-size iterations

```
int TS = 4 * 1024;
#pragma omp taskloop grainsize(TS)
for ( i = 0; i<SIZE; i+=1) {
   A[i]=A[i]*B[i]*S;
}
```

- Clause: num_tasks(num-tasks)
  - → Create num-tasks chunks
  - → Each chunk must have at least one iteration

```
int NT = 4 * omp_get_num_threads();
#pragma omp taskloop num_tasks(NT)
for ( i = 0; i<SIZE; i+=1) {
   A[i]=A[i]*B[i]*S;
}
```

- If none of previous clauses is present, the *number of chunks* and the *number of iterations per chunk* is implementation defined
- Additional considerations:
  - → The order of the creation of the loop tasks is unspecified
  - → Taskloop creates an implicit taskgroup region; **nogroup** → no implicit taskgroup region is created

Collapsing iteration spaces with taskloop

- The collapse clause in the taskloop construct

```
#pragma omp taskloop collapse(n)
{structured-for-loops}
```

  - → Number of loops associated with the taskloop construct (n)
  - → Loops are collapsed into one larger iteration space
  - → Then divided according to the **grainsize** and **num_tasks**
- Intervening code between any two associated loops
  - → at least once per iteration of the enclosing loop
  - → at most once per iteration of the innermost loop

```
#pragma omp taskloop collapse(2)
for ( i = 0; i<SX; i+=1) {
   for (  j= 0; i<SY; j+=1) {
      for ( k = 0; i<SZ; k+=1) {
         A[f(i,j,k)]=<expression>;
      }
   }
}
```

```
#pragma omp taskloop
for ( ij = 0; i<SX*SY; ij+=1) {
   for ( k = 0; i<SZ; k+=1) {
      i = index_for_i(ij);
      j = index_for_j(ij);
      A[f(i,j,k)]=<expression>;
   }
}
```


Task reductions (using taskloop)

- Clause: `reduction(r-id: list)`
  - → It defines the scope of a new reduction
  - → All created tasks participate in the reduction
  - → It cannot be used with the **nogroup** clause

```
double dotprod(int n, double *x, double *y) {
  double r = 0.0;
  #pragma omp taskloop reduction(+: r)
  for (i = 0; i < n; i++)
    r += x[i] * y[i];

  return r;
}
```

- Clause: `in_reduction(r-id: list)`
  - → Reuse an already defined reduction scope
  - → All created tasks participate in the reduction
  - → It can be used with the **nogroup*** clause, but it is user responsibility to guarantee result

```
double dotprod(int n, double *x, double *y) {
  double r = 0.0;
  #pragma omp taskgroup task_reduction(+: r)
  {
    #pragma omp taskloop in_reduction(+: r)*
    for (i = 0; i < n; i++)
      r += x[i] * y[i];
  }
  return r;
}
```

Improving Tasking Performance: Task dependences

Motivation

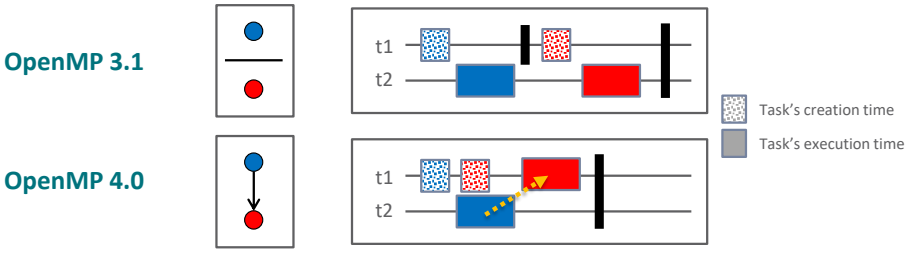
- Task dependences as a way to define task-execution constraints

OpenMP 3.1

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

  #pragma omp taskwait

  #pragma omp task
  x++;
}
```

OpenMP 4.0

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


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



Motivation

- Task dependences as a way to define task-execution constraints

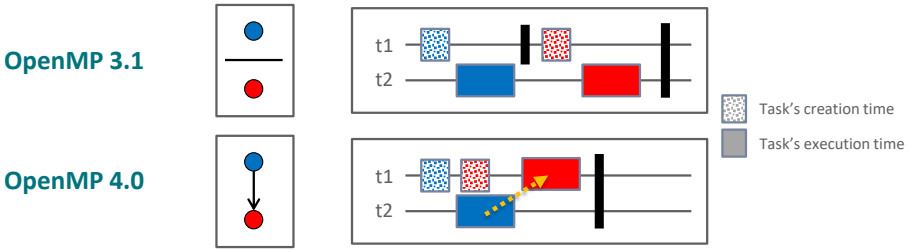
OpenMP 3.1

```
int x = 0;
#pragma omp parallel
#pragma omp single
{
  #pragma omp task
    std::cout << x << std
  #pragma omp taskwait
  #pragma omp task
    x++;
}
```

OpenMP 4.0

```
int x = 0;
#pragma omp parallel
#pragma omp single
{
  #pragma omp task depend(in: x)
                                d::endl;
                                end(inout: x)
    x++;
}
```

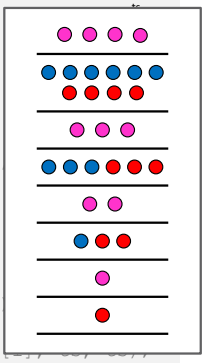
Task dependences can help us to remove "strong" synchronizations, increasing the look ahead and, frequently, the parallelism!!!!



Motivation: Cholesky factorization

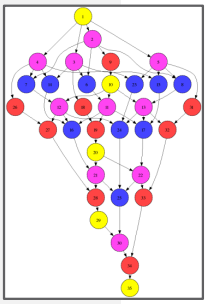
```
void cholesky(int ts, int nt, double* a[nt][nt]) {
  for (int k = 0; k < nt; k++) {
    // Diagonal Block factorization
    potrf(a[k][k], ts, ts);

    // 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]
      }
      #pragma omp task
      syrk(a[k][i], a[i][i], ts, ts);
    }
    #pragma omp taskwait
  }
}
```

OpenMP 3.1



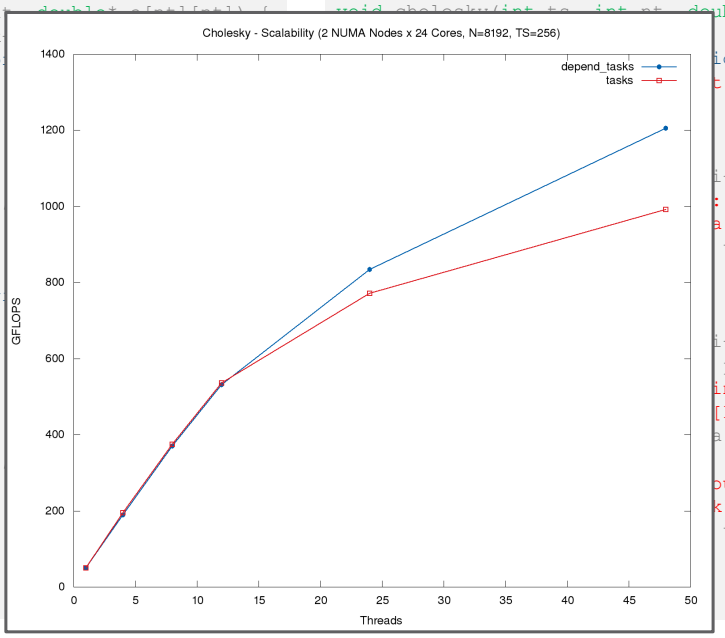
```
void cholesky(int ts, int nt, double* a[nt][nt]) {
  for (int k = 0; k < nt; k++) {
    // Diagonal Block factorization
    #pragma omp task depend(inout: a[k][k])
    potrf(a[k][k], ts, ts);

    // 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);
    }
  }
}
```

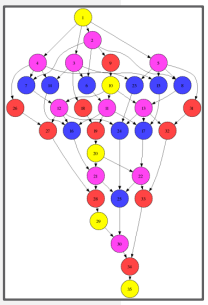
OpenMP 4.0



Motivation: Cholesky factorization

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

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

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





OpenMP 4.0

Using 2017 Intel compiler

What's in the spec

What's in the spec: a bit of history

OpenMP 4.0

- The `depend` clause was added to the `task` construct

OpenMP 4.5

- The `depend` clause was added to the target constructs
- Support to doacross loops

OpenMP 5.0

- `lvalue` expressions in the depend clause
- New dependency type: `mutexinoutset`
- Iterators were added to the `depend` clause
- The `depend` clause was added to the `taskwait` construct
- Dependable objects

What's in the spec: syntax depend clause

```
depend([depend-modifier,] dependency-type: list-items)
```

where:

- `depend-modifier` is used to define iterators
- `dependency-type` may be: `in`, `out`, `inout`, `mutexinoutset` and `depobj`
- A `list-item` may be:
  - C/C++: A `lvalue` expr or an array section `depend(in: x, v[i], *p, w[10:10])`
  - Fortran: A variable or an array section `depend(in: x, v(i), w(10:20))`

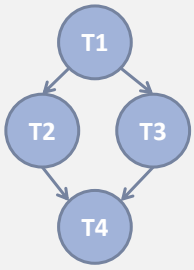
What's in the spec: sema depend clause (1)

- A task cannot be executed until all its predecessor tasks are completed
- If a task defines an `in` dependence over a list-item
  - → the task will depend on all previously generated sibling tasks that reference that list-item in an `out` or `inout` dependence
- If a task defines an `out`/`inout` dependence over list-item
  - → the task will depend on all previously generated sibling tasks that reference that list-item in an `in`, `out` or `inout` dependence

What's in the spec: depend clause (1)

- A task cannot be executed until all its predecessor tasks are completed
- If a task defi
  - → the task will d ne of the list items in an `out` or `in`
- If a task defi
  - → the task will d ne of the list items in an `in`, `out` d

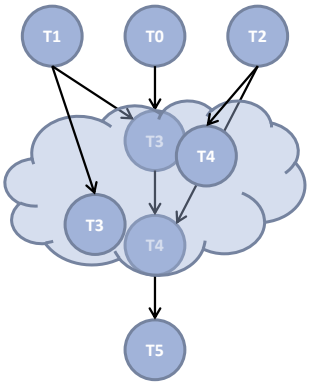
```
int x = 0;
#pragma omp parallel
#pragma omp single
{
  #pragma omp task depend(inout: x) //T1
  { ... }

  #pragma omp task depend(in: x)    //T2
  { ... }

  #pragma omp task depend(in: x)    //T3
  { ... }

  #pragma omp task depend(inout: x) //T4
  { ... }
}
```



What's in the spec: depend clause (2)

- New dependency type: `mutexinoutset`

```
int x = 0, y = 0, res = 0;
#pragma omp parallel
#pragma omp single
{
  #pragma omp task depend(out: res)  //T0
   res = 0;

  #pragma omp task depend(out: x)  //T1
  long_computation(x);

  #pragma omp task depend(out: y)  //T2
  short_computation(y);

  #pragma omp task depend(in: x) depend(mutexinoutset: res) //T3
  res += x;

  #pragma omp task depend(in: y) depend(mutexinoutset: res) //T4
  res += y;

  #pragma omp task depend(in: res)  //T5
  std::cout << res << std::endl;
}
```



1. *inoutset property*: tasks with a `mutexinoutset` dependence create a cloud of tasks (an inout set) that synchronizes with previous & posterior tasks that dependent on the same list item

2. *mutex property*: Tasks inside the inout set can be executed in any order but with mutual exclusion

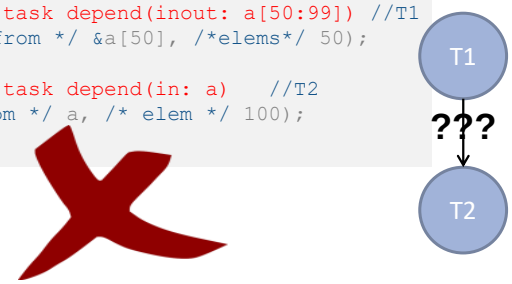
What's in the spec: depend clause (3)

- Task dependences are defined among **sibling tasks**

```
//test1.cc
int x = 0;
#pragma omp parallel
#pragma omp single
{
  #pragma omp task depend(inout: x)    //T1
  {
    #pragma omp task depend(inout: x) //T1.1
    x++;

    #pragma omp taskwait
  }
  #pragma omp task depend(in: x) //T2
  std::cout << x << std::endl;
}
```

- List items used in the depend clauses […] must indicate **identical** or **disjoint storage**

```
//test2.cc
int a[100] = {0};
#pragma omp parallel
#pragma omp single
{
  #pragma omp task depend(inout: a[50:99]) //T1
  compute(/* from */ &a[50], /*elems*/ 50);

  #pragma omp task depend(in: a)    //T2
  print(/* from */ a, /* elem */ 100);
}
```

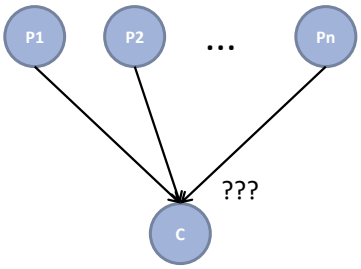


What's in the spec: depend clause (4)

- Iterators + deps: a way to define a dynamic number of dependences

```
std::list<int> list = ...;
int n = list.size();

#pragma omp parallel
#pragma omp single
{
  for (int i = 0; i < n; ++i)
    #pragma omp task depend(out: list[i])      //Px
     compute_elem(list[i]);

  #pragma omp task depend(iterator(j=0:n), in : list[j]) //C
  print_elems(list);
}
```

It seems innocent but it's not:
`depend(out: list.operator[](i))`



Equivalent to:
`depend(in: list[0], list[1], …, list[n-1])`

Philosophy

Philosophy: data-flow model

- Task dependences are orthogonal to data-sharings
  - → **Dependences** as a way to define **a task-execution constraints**
  - → Data-sharings as **how the data is captured** to be used inside the task

```
// test1.cc
int x = 0;
#pragma omp parallel
#pragma omp single
{
  #pragma omp task depend(inout: x) \
                   firstprivate(x) //T1
  x++;

  #pragma omp task depend(in: x)  //T2
  std::cout << x << std::endl;
}
```

OK, but it always prints '0' :(

```
// test2.cc
int x = 0;
#pragma omp parallel
#pragma omp single
{
  #pragma omp task depend(inout: x) //T1
  x++;

  #pragma omp task depend(in: x) \
                   firstprivate(x) //T2
  std::cout << x << std::endl;
}
```

We have a data-race!!

Philosophy: data-flow model (2)

- Properly combining dependences and data-sharings allow us to define a **task data-flow model**
  - →Data that is read in the task → input dependence
  - →Data that is written in the task → output dependence
- A task data-flow model
  - →Enhances the **composability**
  - →**Eases the parallelization** of new regions of your code

Philosophy: data-flow model (3)

```cpp
//test1_v1.cc
int x = 0, y = 0;
#pragma omp parallel
#pragma omp single
{
  #pragma omp task depend(inout: x) //T1
  {
    x++;
    y++;    // !!!
  }
  #pragma omp task depend(in: x)    //T2
  std::cout << x << std::endl;

  #pragma omp taskwait
  std::cout << y << std::endl;
}
```

```cpp
//test1_v2.cc
//test1_v3.cc
//test1_v4.cc
int x = 0, y = 0;
#pragma omp parallel
#pragma omp single
{
  #pragma omp task depend(inout: x, y) //T1
  {
    x++;
    y++;
  }
  #pragma omp task depend(in: x)       //T2
  std::cout << x << std::endl;

  #pragma omp task depend(in: y)       //T3
  std::cout << y << std::endl;
}
```

If all tasks are **properly annotated**,
we only have to worry about the
dependendences & data-sharings of the new task!!!

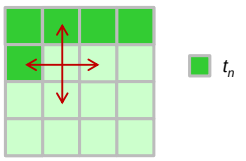
Use case

Use case: intro to Gauss-seidel

```
void serial_gauss_seidel(int tsteps, int size, int (*p)[size]) {
  for (int t = 0; t < tsteps; ++t) {
    for (int i = 1; i < size-1; ++i) {
      for (int j = 1; j < size-1; ++j) {
        p[i][j] = 0.25 * (p[i][j-1] * // left
                          p[i][j+1] * // right
                          p[i-1][j] * // top
                          p[i+1][j]); // bottom
      }
    }
  }
}
```

Access pattern analysis

For a specific t, i and j



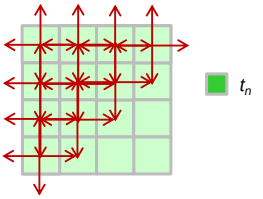
Each cell depends on:
- two cells (north & west) that are computed in the current time step, and
- two cells (south & east) that were computed in the previous time step

Use case: Gauss-seidel (2)

```
void serial_gauss_seidel(int tsteps, int size, int (*p)[size]) {
  for (int t = 0; t < tsteps; ++t) {
    for (int i = 1; i < size-1; ++i) {
      for (int j = 1; j < size-1; ++j) {
        p[i][j] = 0.25 * (p[i][j-1] * // left
                          p[i][j+1] * // right
                          p[i-1][j] * // top
                          p[i+1][j]); // bottom
      }
    }
  }
}
```

1st parallelization strategy

For an specific t



We can exploit the wavefront to obtain parallelism!!

Use case : Gauss-seidel (3)

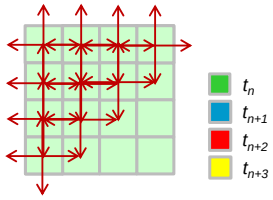
```
void gauss_seidel(int tsteps, int size, int TS, int (*p)[size]) {
  int NB = size / TS;
  #pragma omp parallel
  for (int t = 0; t < tsteps; ++t) {
    // First NB diagonals
    for (int diag = 0; diag < NB; ++diag) {
      #pragma omp for
      for (int d = 0; d <= diag; ++d) {
        int ii = d;
        int jj = diag - d;
        for (int i = 1+ii*TS; i < ((ii+1)*TS); ++i)
          for (int j = 1+jj*TS; i < ((jj+1)*TS); ++j)
            p[i][j] = 0.25 * (p[i][j-1] * p[i][j+1] *
                              p[i-1][j] * p[i+1][j]);
      }
    }
    // Lasts NB diagonals
    for (int diag = NB-1; diag >= 0; --diag) {
      // Similar code to the previous loop
    }
  }
}
```


Use case : Gauss-seidel (4)

```
void serial_gauss_seidel(int tsteps, int size, int (*p)[size]) {
  for (int t = 0; t < tsteps; ++t) {
    for (int i = 1; i < size-1; ++i) {
      for (int j = 1; j < size-1; ++j) {
        p[i][j] = 0.25 * (p[i][j-1] * // left
                          p[i][j+1] * // right
                          p[i-1][j] * // top
                          p[i+1][j]); // bottom
      }
    }
  }
}
```

2nd parallelization strategy

multiple time iterations

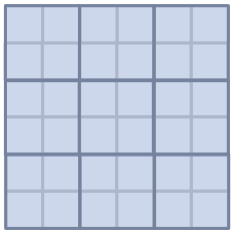


We can exploit the wavefront of multiple time steps to obtain MORE parallelism!!

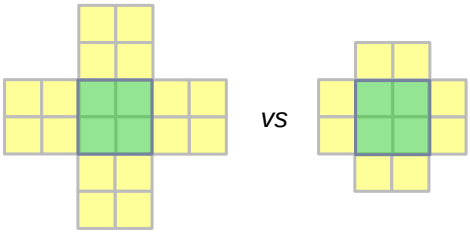
Use case : Gauss-seidel (5)

```
void gauss_seidel(int tsteps, int size, int TS, int (*p)[size]) {
  int NB = size / TS;

  #pragma omp parallel
  #pragma omp single
  for (int t = 0; t < tsteps; ++t)
    for (int ii=1; ii < size-1; ii+=TS)
      for (int jj=1; jj < size-1; jj+=TS) {
        #pragma omp task depend(inout: p[ii:TS][jj:TS])
            depend(in: p[ii-TS:TS][jj:TS], p[ii+TS:TS][jj:TS],
                       p[ii:TS][jj-TS:TS], p[ii:TS][jj:TS])
        {
          for (int i=ii; i<(1+ii)*TS; ++i)
            for (int j=jj; j<(1+jj)*TS; ++j)
               p[i][j] = 0.25 * (p[i][j-1] * p[i][j+1] *
                                 p[i-1][j] * p[i+1][j]);
        }
      }
}
```

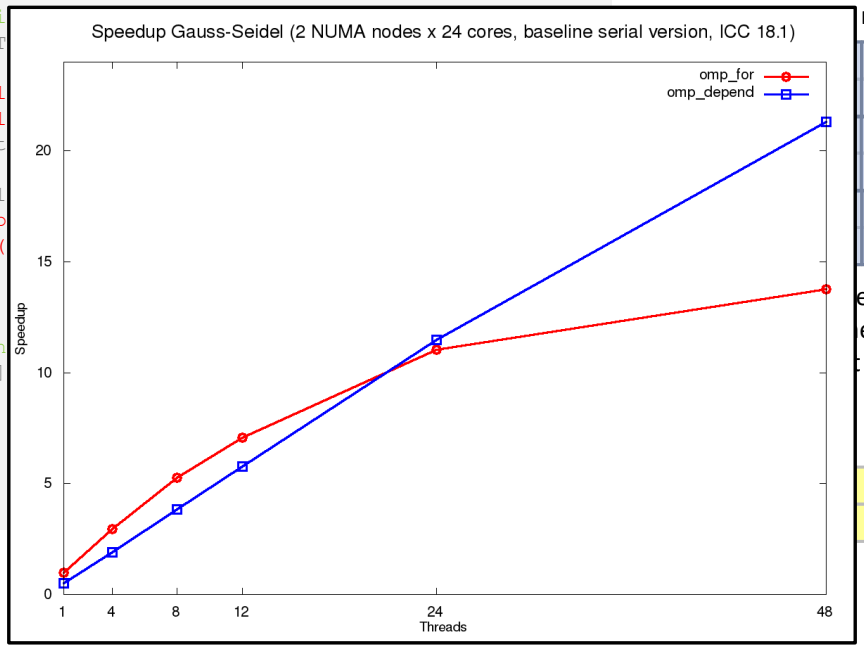
inner matrix region



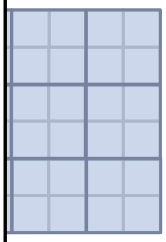
Q: Why do the input dependences depend on the whole block rather than just a column/row?



Use case : Gauss-seidel (5)



matrix region



e input dependences
e whole block rather
a column/row?