SHARED MEMORY PARALLEL COMPUTING

COMP4300/8300 PARALLEL SYSTEMS

PROF. JOHN TAYLOR

APRIL 2024

Australian National University

Logistics

- Attendance to the Lab sessions is highly encouraged. Most of the practical aspects of the programming models are covered in the Labs.

OpenMP:
Part II

OpenMP:
Quick Demo

OpenMP Reference Material

- **Using OpenMP – The Next Step**, R. van der Pas, E. Stotzer, and C. Terboven, Chapter 1
- http://www.openmp.org/
- *Introduction to High Performance Computing for Scientists and Engineers*, Hager and Wellein, Chapter 6 & 7
- *High Performance Computing*, Dowd and Severance, Chapter 11
- *Introduction to Parallel Computing, 2nd Ed*, A. Grama, A. Gupta, G. Karypis, V. Kumar
- *Parallel Programming in OpenMP*, R. Chandra, L.Dagum, D.Kohr, D.Maydan. J.McDonald, R.Menon

The `for` Work-Sharing Directives

- The order in which threads execute is not predictable (OS scheduled).
- The way to map iterations to threads can be specified by the programmer (see later `schedule` clause).
- If the programmer does not specify the mapping between threads and iterations, the compiler decides which strategy to use.

```
Thread 0 executes loop iteration 0
Thread 0 executes loop iteration 1
Thread 0 executes loop iteration 2
Thread 3 executes loop iteration 7
Thread 3 executes loop iteration 8
Thread 2 executes loop iteration 5
Thread 2 executes loop iteration 6
Thread 1 executes loop iteration 3
Thread 1 executes loop iteration 4
```

The `schedule` clause

- The `schedule` clause of the `for` directive specifies how iterations are mapped to threads

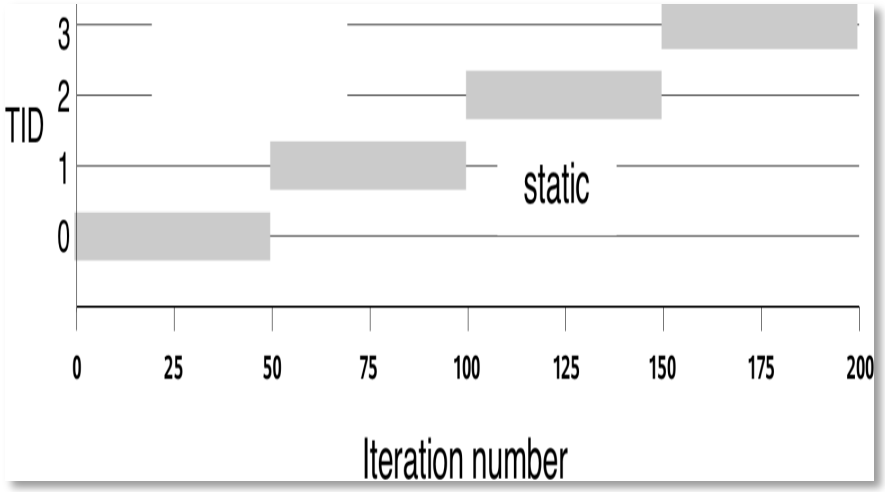
```
schedule(scheduling_clause[,chunk_size])
```

```
#pragma omp parallel for schedule ( scheduling_clause [, chunk_size ])
```

- The granularity of the workload distribution is a *chunk*, a continuous non-empty subset of the iteration space.

The `schedule` clause

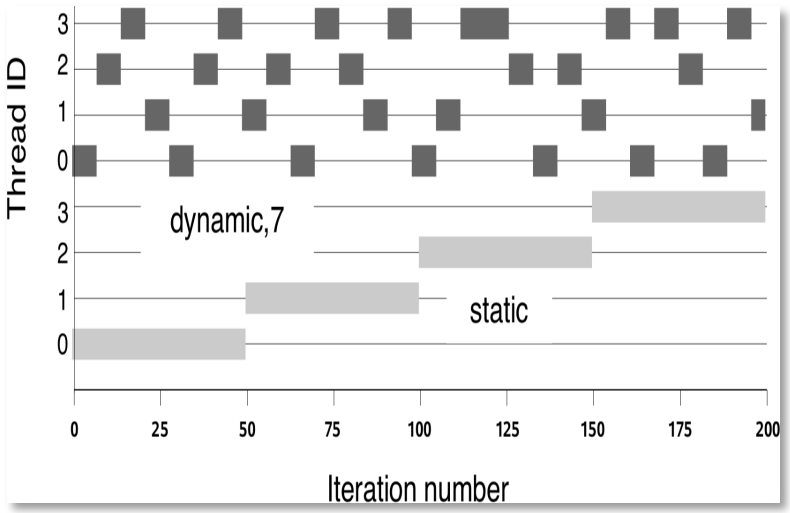
The most straightforward schedule is
`schedule(static[,chunk_size])`



- Splits the iteration space into chunks of size `chunk_size` and allocates to threads statically in a round-robin fashion
- No specification implies the number of chunks equals the number of threads Iterations are assigned in the order of the thread number and the last chunk may have a smaller number of iterations.
- It has the lowest overhead and is the default setting for many OpenMP compilers.
- **Potential load imbalance** - if iterations have different workloads, consider dynamic scheduling (schedule(dynamic)).

The `schedule` clause

The schedule `(dynamic[,chunk_size])` enables basic dynamic load balancing

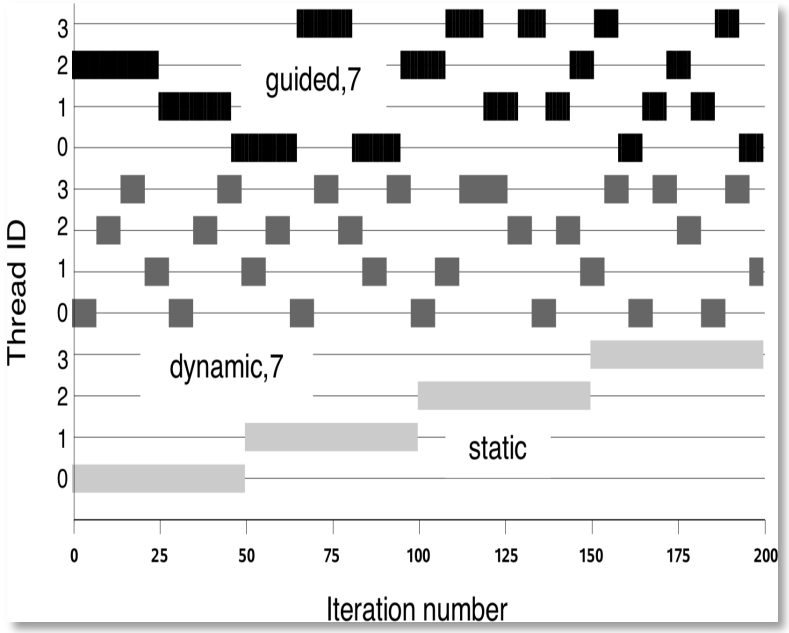


- iteration space split into `chunk_size` blocks that are scheduled dynamically as they complete the work on the current chunk, threads request additional chunks
- if not specified `chunk_size` is set by default to 1
- Do you think there is any disadvantage in this schedule?

The `schedule` clause

`schedule(runtime)`

- The choice of the optimal schedule may depend on the problem size
- The schedule and (optional) chunk size are set through the `OMP_SCHEDULE` environment variable



Loop Schedules

Example of loop schedules in OpenMP

- 20 iterations by three threads
- Default chunksize for `dynamic` and `guided` is one
- Note that only the `static` schedule guarantee that the distribution of chunks stays the same between runs

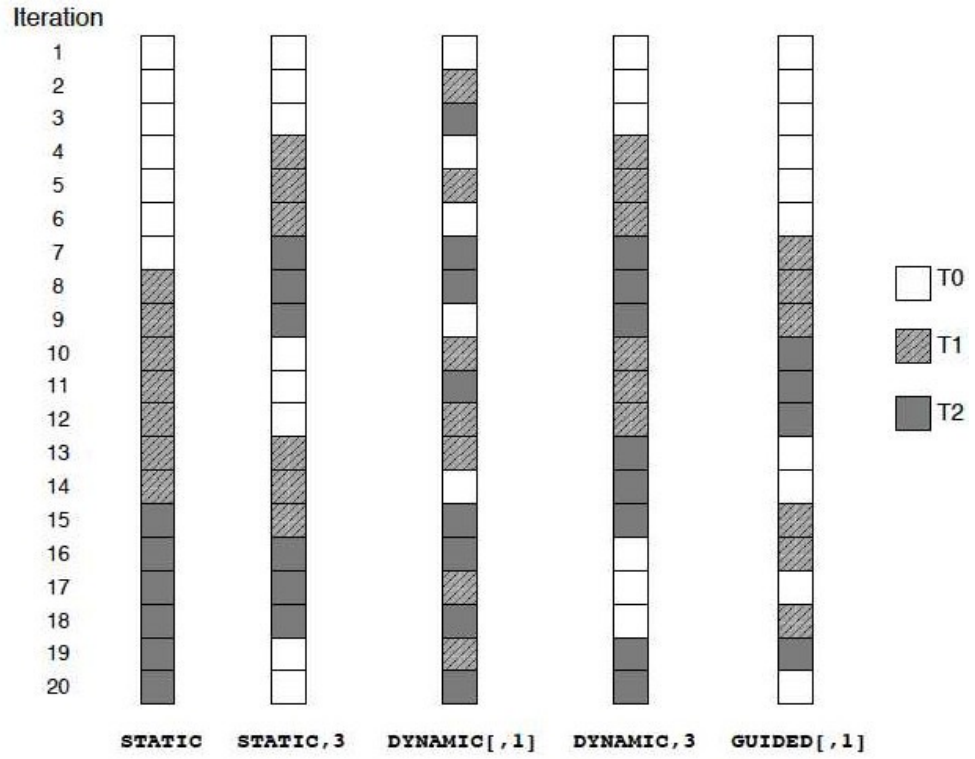


Figure 6.2: Loop schedules in OpenMP. The example loop has 20 iterations and is executed by three threads (T0, T1, T2). The default chunksize for DYNAMIC and GUIDED is one. If a chunksize is specified, the last chunk may be shorter. Note that only the STATIC schedules guarantee that the distribution of chunks among threads stays the same from run to run.

Introduction to High Performance Computing for Scientists and Engineers, Hager and Wellein, Figure 6.2

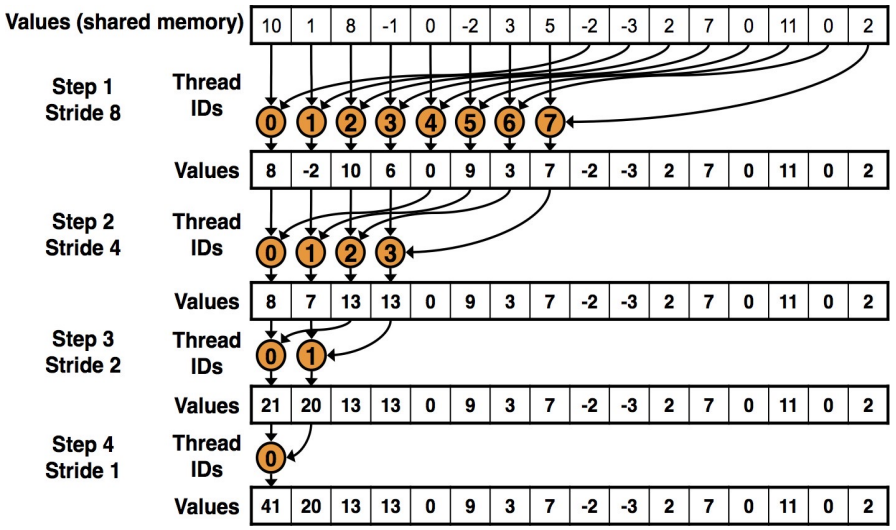
The `reduction` clause

```
sum = 0;
partial_sum [ NTHREADS ] = {0};
#pragma omp parallel for default (none) shared ( n, a, partial_sum )
 for (i = 0; i < n; ++i) {
  partial_sum [ omp_get_thread_num ()] += a[ i]
 } /* End of parallel region */
 for (i = 0; i < NTHREADS ; ++i){
   sum += partial_sum [ i];
}
```

The `reduction` clause

```
sum = 0;
partial_sum [ NTHREADS ] = {0};
#pragma omp parallel for default (none) shared ( n, a, partial_sum )
 for (i = 0; i < n; ++i) {
  partial_sum [ omp_get_thread_num ()] += a[ i]
 } /* End of parallel region */
 for (i = 0; i < NTHREADS ; ++i){
   sum += partial_sum [ i];
}
```

Is there a better algorithm?



The `reduction` clause

```
#pragma omp parallel for default (none) shared ( n, a) reduction (+: sum )
  for (i = 0; i < n; ++i) {
     sum += a[ i]
  } /* End of parallel reduction */
```

- `reduction(+:sum)`
  - Implements a reduction operation - The clause ensures that each thread maintains a private copy of the reduction variable, and at the end of the parallel region, the private copies are combined into a single result.
  - It is not necessary to specify the reduction variable as shared
  - The order in which thread-specific values are combined is unspecified
  - The operators supported are `+, ¡, *, &, , ^, && and`

- Some constraints
  - Aggregate types, pointer types and references types are not supported
  - A reduction variable must not be `const`
  - No overloaded operators with respect to the variable that appears in the clause

Nested Parallelism

```
#pragma omp parallel for num_threads (2)
  for (i = 0; i < Ni; i ++}
     #pragma omp parallel for num_threads (2)
     for (j = 0; j < Nj; j ++) {
```

- By default, the inner loop is serialized and run by one thread
- To enable multiple threads in nested parallel loops requires environment variable `OMP_NESTED` to be `TRUE`
- You can also control the maximum level of nesting through `OMP_MAX_ACTIVE_LEVELS`
- Each thread from the first `parallel for` will spawn a new team.
- Note - the use of synchronization constructions in nested parallel sections requires care (see OpenMP specs, *e.g.* there is no restriction on synchronizing threads across different teams!).

Nested Parallelism

```
#pragma omp parallel for collapse (2)
  for (i = 0; i < Ni; i ++) {
    for (j = 0; j < Nj; j ++) { …
```

A disadvantage of nested parallelism is parallel overhead

- `Collapse` is usually better way of parallelizing nested loops
- The `collapse` clause turns the two loops into a combined single loop.
- Threads work on a larger chunk of work, which can improve parallel efficiency.
- By collapsing loops, you reduce the overhead of managing multiple parallel loops.

OpenMP: Synchronization Directives

The `barrier` directive

```
#pragma omp barrier
/* structured block */
```

- `barrier`: Each thread waits at the `barrier` until all threads arrive

Restrictions:

- Each barrier must be encountered by all threads in a team, or by none at all.
- The sequence of work-sharing regions and barrier regions encountered must be the same for each thread in the team.

The `ordered` directive

```
#pragma omp ordered
/* structured block */
```

- `ordered`: threads execute the structured block in sequential order.

```
cumul_sum [0] = list [0];
#pragma omp parallel for ordered shared ( cumul_sum , list
, n)
for (i =1; i<n; i ++) {
 /* other processing on list [ i] if required */
 #pragma omp ordered
 {
  cumul_sum [ i] = cumul_sum [ i -1] + list [ i];
 }
}
```

- The first thread that encounters this directive enters the structured block without waiting.
- Any subsequent threads wait until previous threads have completed the block execution.
- Restrictions: An `ordered`clause must be added to the parallel region in which this construct appears.

The `atomic` directive

```
#pragma omp atomic [seq_cst[,]] atomic-clause [[,] seq_cst]
/* single expression statement */
#pragma omp atomic [seq_cst[,]]
/* single expression statement */
#pragma omp atomic [seq_cst[,]] capture [[,] seq_cst]
/* structured block *
```

- `atomic`: This is a hint for the compiler to use low-level atomic instructions if available.
- Atomic clauses
  - `read`: performs an atomic read of the input variable and stores it into the output. This is guaranteed regardless of the size of the variable.
  - `write`: the output variable is written atomically. This is guaranteed regardless of the size of the variable.

```
x = expr ; // x is written atomically
```

The `atomic` directive

Atomic clauses

- `update`: causes an atomic update of the location designated by x using the designated operator or intrinsic.

```
x++;
++x; x- -;
-- x;
x binop = expr ;
x = x binop expr ; x = expr binop x;
```

The following rules also apply

- The evaluation of `expr` need not be atomic with respect to the read or write of the location designated by x.
- No task scheduling points occur between the read and the write of the location designated by x.
- Binop (binary operators) is one of the following operators `+, * , -, & , ^, <<, >>.`

The `atomic` directive

Atomic clauses

- `capture`: causes an atomic update of the location designated by x using the designated operator or intrinsic while also capturing the original or final value of the location (v) designated by x with respect to the atomic update.

```
v = x ++;
v = ++x;
v = x - -;
v = -- x;
v = x binop = expr;
v = x = x binop expr;
v = x = expr binop x;
```

The following rules also apply

- Only the read and write of the location designated by x are performed mutually atomically.
- The evaluation of expr, and the write to the location designated by v do not need to be atomic with respect to the read or write of the location designated by x.
- No task scheduling points occur between the read and the write of the location designated by x.

```
{ v = x; x += n; } // atomically update x, but capture original value in v
```

OpenMP: Library Functions and Environment Variables

Internal Control Variables

The OpenMP standard defines some internal control variables (ICVs) controlled by the implementations that govern the behaviour of a program at run time. Here are some of them:

- `nthreads-var` – stores the number of threads requested for the execution of future parallel regions.
- `dyn-var` – control whether dynamic adjustment of the number of threads to be used in future parallel regions is enabled.
- `nest-var` – controls whether nested parallelism is enabled for future parallel regions.
- `run-sched-var` – stores scheduling information to be used for loop regions using the runtime schedule.
- `def-sched-var` – stores implementation-dependent scheduling information to be used for loop regions.

These variables cannot be accessed directly, but via either library functions or environment variables.

Library Functions

Defined in header file

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

Controlling threads and processors

```
void omp_set_num_threads(int num_threads)
int omp_get_num_threads()
int omp_get_max_threads() // Number of threads used in the next parallel region
int omp_get_thread_num()
int omp_get_num_procs() // Returns the number of processor cores available
int omp_in_parallel() // Check if within a parallel region
```

Controlling thread creation

```
void omp_set_dynamic(int dynamic_threads) // Enable/disable dynamic thread adjustment
int omp_get_dynamic() // Check whether dynamic thread adjustment is enabled
void omp_set_nested(int nested) int omp_get_nested()
void omp_set_dynamic(int dynamic_threads) // Enable/disable dynamic thread adjustment
int omp_get_dynamic() // Check whether dynamic thread adjustment is enabled
void omp_set_nested(int nested) int omp_get_nested()
```

OpenMP Environment Variables

`OMP_NUM_THREADS`: default number of threads entering parallel region

`OMP_DYNAMIC`: if `TRUE` it permits the number of threads to change during execution, in order to optimize system resources

`OMP_NESTED`: if `TRUE` it permits nested parallel regions

`OMP_SCHEDULE`: determines scheduling for loops that are defined to have `runtime` scheduling

```
export OMP_SCHEDULE =" static ,4 "
export OMP_SCHEDULE =" dynamic "
export OMP_SCHEDULE =" guided "
```

OpenMP Environment Variables

`OMP_DISPLAY_ENV`: this is a very useful variable to verify all the settings. If set to `true` or `verbose` all the relevant environment variables are printed at the beginning of the program.

`OMP_STACKSIZE`: this variable allows to increase the default stack size allocated to each thread.

- The syntax supports a case-insensitive unit qualifier that is appended to the number: `B` for bytes, `K` for 1024 bytes, `M` for 1024 bytes Kbytes, and `G` for 1024 Mbytes.

OpenMP and Pthreads

	OpenMP	Pthreads
Definition	Specification for compiler directives, library routines, and environment variables.	POSIX standard for libraries, providing low-level thread management.
Parallelism Model	**Shared Memory**: Utilizes multiple threads within a single process.	**Shared Memory**: Also operates within a single process, but with more manual control over threads.
Communication	Implicit (compiler handles thread synchronization).	Requires explicit handling of thread communication using functions like pthread_create and pthread_join.
Ease of Use	Easier to program and debug due to directives.	Requires more manual management of threads and synchronization.
Scalability	Limited scalability within a single node.	Scalable to multiple cores within a single machine.
Use Cases	Best for parallelizing loops and simple tasks within a single program.	Suitable for applications that need fine-grained control over threads and synchronization.

OpenMP and Pthreads

- OpenMP removes the need for a programmer to initialize task attributes, set up arguments to threads, partition iteration spaces, etc.
- OpenMP code can closely resemble serial code – (verification)
- OpenMP users require availability of an OpenMP compiler
  - performance dependent on quality of compiler — hardly a problem today
- Well-engineered OpenMP code causes no loss of performance with respect to lower-level APIs.
- Pthreads has a lower-level API that is slightly more flexible and rich, (e.g. condition waits, locks of different types etc) but also more error prone
- Pthreads is library based and not compiler-based
- OpenMP is now the *de facto* standard in High-Performance Computing

OpenMP and MPI

	OpenMP	MPI
Definition	Specification for compiler directives, library routines, and environment variables.	Library specification for message-passing, proposed as a standard by a committee of vendors, implementors, and users.
Parallelism Model	**Shared Memory**: Utilizes multiple threads within a single process.	**Distributed Memory**: Operates across a network of distributed nodes.
Communication	Implicit (compiler handles thread synchronization).	Explicit (programmer manages message passing using API calls like MPI_Send and MPI_Recv).
Architecture	Suitable for multi-core processors.	Works on both shared-memory and distributed-memory architectures.
Ease of Use	Easier to program and debug due to directives.	Requires more explicit handling of communication and synchronization.
Scalability	Limited to scalability within a single node.	Scalable to large clusters and supercomputers.
Use Cases	Best for parallelizing loops and simple tasks within a single program.	Ideal for distributed computing, large-scale simulations, and complex parallel applications.