Open MP

Assigning Iterations to Threads

- Schedule clause of the for directive deals with the assignment of iterations to threads.
- Schedule directive is schedule(scheduling_class[, parameter]).
- OpenMP supports four scheduling classes:
  - static
  - Dynamic
  - guided
  - runtime

Assigning Iterations to Threads

- Static N Schedule(interleaved) (Interleaved)
- The iteration space is broken in chunks of approximately size
- *N/num − threads*. Then these chunks are assigned to the threads in a Round-Robin fashion

- Characteristics of the static schedules
  - Low overhead
  - Good locality (usually)
  - Can have load imbalance problems

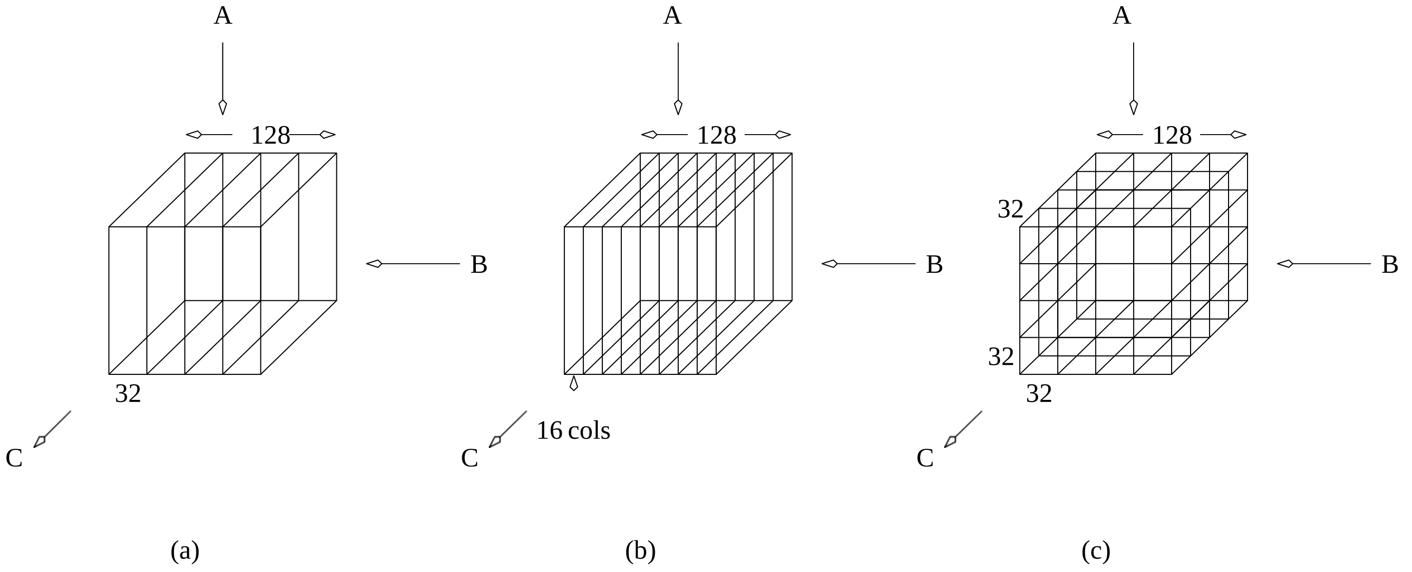
Assigning Iterations to Threads

```
/* static scheduling of matrix multiplication loops */
#pragma omp parallel default(private) shared (a, b, c, dim) \
                     num_threads(4)

 #pragma omp for schedule(static)
 for (i = 0; i < dim; i++) {
     for (j = 0; j < dim; j++) {

          c(i,j) = 0;
          for (k = 0; k < dim; k++) {
              c(i,j) += a(i, k) * b(k, j);
          }
      }
  }
```

Assigning Iterations to Threads: Example



Three different schedules using the static scheduling class of OpenMP.

Assigning Iterations to Threads: Example

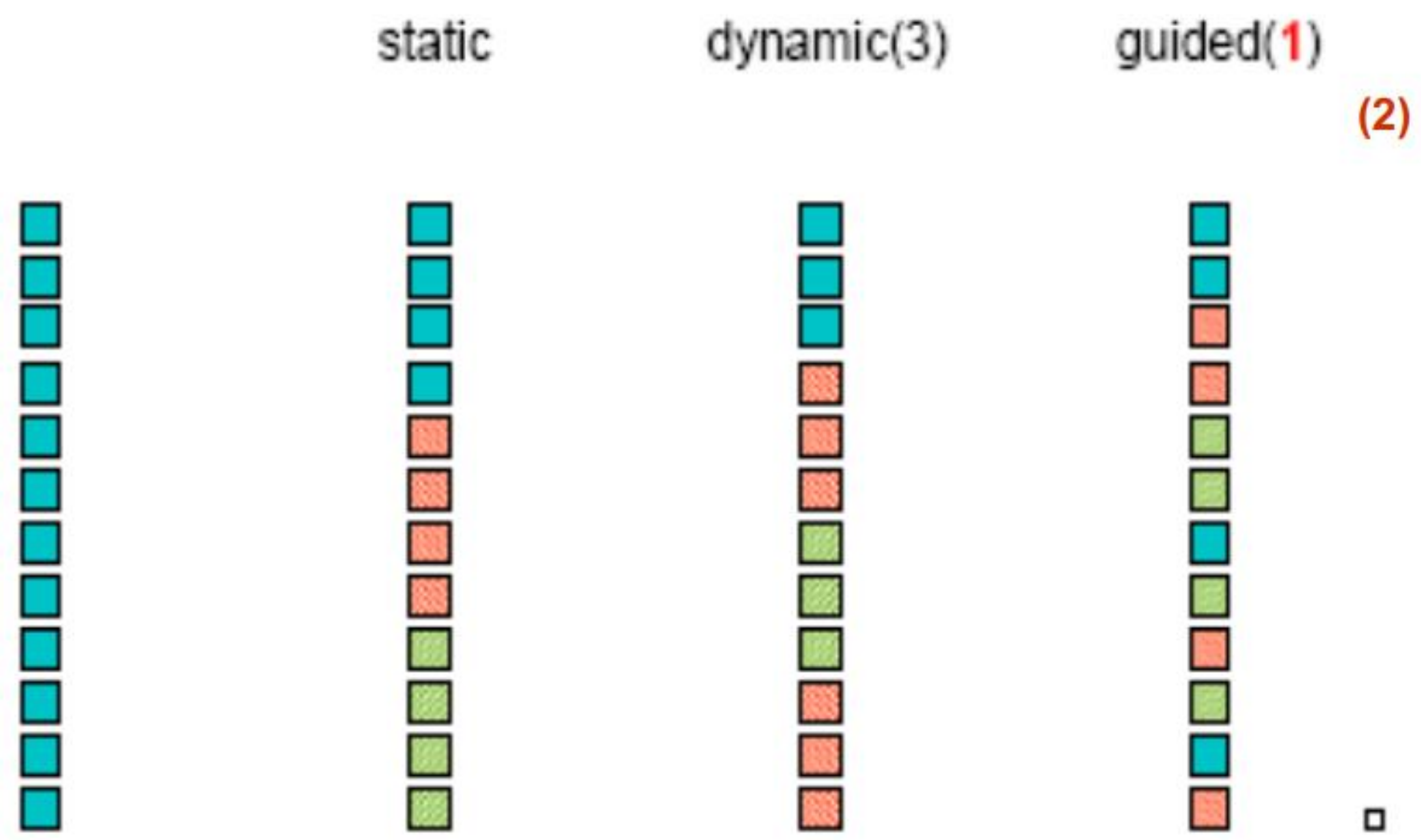
- Dynamic, N schedule
- Threads dynamically grab chunks of *N* iterations until all iterations have been executed. If no chunk is specified, *N* = 1.

- Guided, N schedule
- Variant of **`dynamic`**. The size of the chunks deceases as the threads grab iterations, but it is at least of size *N*. If no chunk is specified,
- *N* = 1.

- Characteristics of dynamic schedules
  - Higher overhead
  - Not very good locality (usually)
    Can solve imbalance problems

- Runtime, N Schedule
- it is desirable to delay scheduling decisions until runtime.

Assigning Iterations to Threads



Parallel For Loops

- Often, it is desirable to have a sequence of `for`-directives within a parallel construct that do not execute an implicit barrier at the end of each `for` directive.

- OpenMP provides a clause – `nowait`, which can be used with a `for` directive.

Parallel For Loops: Example

```
#pragma omp parallel
{
    #pragma omp for nowait
        for (i = 0; i < nmax; i++)
            if (isEqual(name,
                current_list[i])
                processCurrentName(name);
    #pragma omp for
        for (i = 0; i < mmax; i++)
            if (isEqual(name,
                past_list[i])
                processPastName(name);


 }
```

The `sections` Directive

- OpenMP supports non-iterative parallel task assignment using the `sections` directive.
- The general form of the `sections` directive is as follows:

```
• #pragma omp sections [clause list]
• {
  • [#pragma omp section
    • /* structured block */
  • ]
  • [#pragma omp section
    • /* structured block */
  • ]
  • ...
• }
```

The `sections` Directive: Example

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        {
            taskA();
        }
        #pragma omp section
        {
            taskB();
        }
        #pragma omp section
        {
            taskC();
        }
    }
}
```

Merging Directives

```
#pragma omp parallel default (private) shared (n)
{
#pragma omp for
for (i = 0 < i < n; i++) {
/* body of parallel for loop */
}
}
```

Merging Directives

Identical to

```
#pragma omp parallel for default (private) shared (n)
{
for (i = 0 < i < n; i++) {
/* body of parallel for loop */
}
}
```

Merging Directives

```
#pragma omp parallel
{
#pragma omp sections
{
#pragma omp section
{
taskA();
}
#pragma omp section
{
taskB();
}
/* other sections here */
}
}
```

Merging Directives

```
#pragma omp parallel sections
{
#pragma omp section
{
 taskA();
}
 #pragma omp section
{
taskB();
}
/* other sections here */
}
```

Nesting parallel Directives

```
#pragma omp parallel for default(private) shared (a, b, c, dim) \
num_threads(2)
for (i = 0; i < dim; i++) {
#pragma omp parallel for default(private) shared (a, b, c, dim) \
num_threads(2)
for (j = 0; j < dim; j++) {
c(i,j) = 0;
#pragma omp parallel for default(private) \
shared (a, b, c, dim) num_threads(2)
for (k = 0; k < dim; k++) {
c(i,j) += a(i, k) * b(k, j);
}
}
}
```

Synchronization Constructs in OpenMP

- Synchronization Point: The barrier Directive

```
#pragma omp barrier
```

- all threads in a team wait until others have caught up, and then release.
- nested parallel directives, the barrier directive binds to the closest parallel directive.

Single Thread Executions: The single and master Directives

- #pragma omp single [clause list]

/* structured block */

- A single directive specifies a structured block that is executed by a single (arbitrary) thread.
- clauses list can be private, firstprivate, and nowait

Single Thread Executions: The single and master Directives

- The master directive is a specialization of the single directive in which only the master thread executes the structured block.
- The syntax of the master directive is as follows:
- #pragma omp master

  structured block

Critical Sections: The critical and atomic Directives

- Critical Sections: The critical and atomic directives
- #pragma omp critical [(name)]

  structured block
- name can be used to identify a critical region.
- The use of name allows different threads to execute different code while being protected from each other.

Critical Sections: The critical and atomic Directives

- Critical Sections: The critical and atomic directives
- #pragma omp critical [(name)]
  structured block
- Critical Directive: allows only one thread is inside the critical region. All the others must wait.
- name can be used to identify a critical region.
- name field is optional. If no name is specified, the critical section maps to a default name that is the same for all unnamed critical sections
- The use of name allows different threads to execute different code while being protected from each other.

Critical Sections: The critical and atomic Directives

```
#pragma omp parallel sections
 {
 #pragma parallel section
 {
 /* producer thread */
 task = produce_task();
 #pragma omp critical ( task_queue)
 {
 insert_into_queue(task);
 }
```

Critical Sections: The critical and atomic Directives

```
}
 #pragma parallel section
{
 /* consumer thread */
 #pragma omp critical ( task_queue)
 {
 task = extract_from_queue(task);
 }
 consume_task(task);
 }
 }
```

Note : queue full and queue empty conditions must be explicitly handled here in functions insert_into_queue and extract_from_queue.

Critical Sections: The critical and atomic Directives

- The atomic directive specifies that the memory location update in the following instruction should be performed as an atomic operation.
- The update instruction can be one of the following forms:

    x binary_operation = expr

    x++

    ++x

    x--

    --x

Ordered Directive

- Enclosed code is executed in the same order as would occur in sequential execution of the loop
- #pragma omp ordered

  structured block

Ordered Directive

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

Memory Consistency: The flush Directive

- synchronization point at which the system must provide a consistent view of memory
  - all thread visible variables must be written back to memory (if no list is provided), otherwise only those in the list are written back
- Implicit flushes of all variables occur automatically at
  - all explicit and implicit barriers
  - entry and exit from critical regions
  - entry and exit from lock routines
- Directives
  - #pragma omp flush [(list)]

Data Handling in OpenMP

- #pragma omp parallel[data scope clauses ...]
  - Shared
  - private
  - firstprivate
  - lastprivate
  - threadprivate
  - default

Shared

- Shared data among team of threads
- Each thread can modify shared variables
- Data corruption is possible when multiple threads attempt to update the same memory location
- Data correctness is user's responsibility
- float dot_prod(float* a, float* b, int N)
- {
- float sum = 0. 0 ;
- #pragma omp parallel for shared(sum)
- for(int i=0; i<n; i++)
- sum+=a[i]*b[i];
- }
- return sum
- }

Example

```
int x=1;
#pragma omp parallel shared (x) num_threads(2)
{
    x++;
    printf("%d\n",x);
}
printf("%d\n",x);
```

Example

```
int x=1;
#pragma omp parallel shared(x) num_threads(2)
{
    x++;
    printf("%d\n",x);
}
printf("%d\n",x);
```

Prints 2 or 3 (three printfs in total)

Private

- The values of private data are undefined upon entry to and exit from the specific construct. Loop iteration variable is private by default Example:

```
   void* work(float* c, int N)
{
 float x, y;
int i;
#pragma omp parallel for private(x,y)
for(i=0; i<N; i++)
X=a[i]; y=b[i];
C[i]=x+y;
}
}
```

Example

```
int x=1;
#pragma omp parallel private (x) num_threads(2)
{
    x++;
    printf("%d\n",x);
}
printf("%d\n",x);
```

Can print anything (twice, same or different)

Example

```
int x=1;
#pragma omp parallel private (x) num_threads(2)
{
    x++;
    printf("%d\n",x);
}
printf("%d\n",x);
```

Prints 1

firstprivate clause(Data Scope)

- The clause combines behavior of private clause with automatic initialization of the variables in its list with values prior to parallel region.

Example:

int b=51, n=100 ;

printf(“Before parallel loop: b=%d ,n=%d\n”,b,n) #pragma omp parallel for private(i), firstprivate(b)

for(i=0;i<n;i++)

{

a[i]=i+b;

}

firstprivate

- incr=0;
- #pragma omp parallel for firstprivate(incr);
- for (I=0;I<=MAX;I++)
- {
- if ((I%2)==0) incr++;
- A(I)=incr;
- }

Example

```
int x=1;
#pragma omp parallel firstprivate (x) num_threads(2)
{
    x++;
    printf("%d\n",x);
}
printf("%d\n",x);
```

Prints 2 (twice)

Example

```
int x=1;
#pragma omp parallel firstprivate (x) num_threads(2)
{
   x++;
   printf("%d\n",x);
}
printf("%d\n",x);  <- Prints 1
```

Lastprivate

- Variables update shared variable using value from last iteration

```
void sq2(int n, double *last term)
{
double x; int i;
#pragma omp parallel
#pragma omp for lastprivate(x)
for ( i = 0; i < n; i++)
{
x = a[i]*a[i] + b[i]*b[i];
b[i] = sqrt( x);
}
last term = x;
 }
```

Threadprivate and copyin clause

- Preserves global scope for per-thread storage
- Legal for name-space-scope and file-scope
- Use copyin to initialize from master thread

```
struct A;
#pragma omp threadprivate(A)
#pragma omp parallel copyin(A)
do_something_to(&A)
#pragma omp paralle l do_something_else_to(&A)
```

Private copies to thread

Example

```
char *foo ()
{
   static char buffer[BUF_SIZE];
#pragma omp threadprivate(buffer)/* Creates one static copy of buffer per thread*/

   ...

   return buffer;
}
```

Example

```
char *foo ()
{
   static char buffer[BUF_SIZE];
   #pragma omp threadprivate(buffer)

   ...

   return buffer;
}
```

Now foo can be called by multiple threads at the same time

foo returns correct address to caller

OpenMP Library Functions

In addition to directives, OpenMP also supports a number of functions that allow a programmer to control the execution of threaded programs.

Thread control:

- `void omp_set_num_threads(int number)`

  Set the `number` of threads to be used for subsequent parallel regions (without a `num_threads` clause)

- `int omp_get_thread_num(void)`

  Get my thread ID in the current team

- `int omp_get_num_threads(void)`

  Get the number of threads in the team that currently executes the region

- `int omp_get_max_threads(void)`

  Get the maximum number of threads that could be used to form a new team in a parallel region without a num_threads clause

Processor level:

- `int omp_get_num_procs(void)`

  Get the number of processors available on the device

Check if inside a parallel region:

- `int omp_in_parallel(void)`

  If execution is inside an active parallel region, the returned value evaluates to *true*

You can change the scheduling policy in your code for places where you have used the `schedule(runtime)` clause:

Schedule control:

- `void omp_set_schedule(omp_sched_t kind, int chunk_size)`
- `void omp_get_schedule(omp_sched_t* kind, int* chunk_size)`

```
typedef enum omp_sched_t {
    omp_sched_static  = 1,
    omp_sched_dynamic = 2,
    omp_sched_guided  = 3,
    omp_sched_auto    = 4
} omp_sched_t;
```

Defined in the header `omp.h`

```
#include <omp.h>

int main(int argc, char* argv[])
{
#pragma omp parallel
    {
        // set a desired schedule.  This will overwrite what has been set in
        // the OMP_SCHEDULE environment variable
        omp_set_schedule(omp_sched_dynamic, 1);

        // test it
        omp_sched_t sched;
        int chunk;
        omp_get_schedule(&sched, &chunk);
#pragma omp master
        cout << "schedule = " << sched << ", chunk = " << chunk << endl;

#pragma omp for schedule(runtime)
        for (int i = 0; i < N; ++i)
            // do work
```

OpenMP provides convenient timing routines:

Timer:

- `double omp_get_wtime(void)`
- `double omp_get_wtick(void)`

Returns the per-thread wall time between two measurement points

Returns the number of seconds between two consecutive clock cycles.

```
int main(int argc, char* argv[])
{
#pragma omp parallel
    {
        const int tid = omp_get_thread_num();
        random_device rd; // #include <random>
        mt19937 gen(rd());
        uniform_int_distribution<> uniform(1, 500);

        const int mytime = uniform(gen) * 1000; // micro seconds
#pragma omp critical
        cout << "thread " << tid << ": " << mytime*1.0e-6 << " sec (expect)" << endl;

        const double start = omp_get_wtime();
        usleep(mytime); // #include <unistd.h>
        const double duration = omp_get_wtime() - start;
#pragma omp critical
        cout << "thread " << tid << ": " << mytime*1.0e-6 << " sec (measured)" << endl;
    }
    return 0;
}
```

Output:

```
thread 1: 0.031 sec (expect)
thread 3: 0.204 sec (expect)
thread 2: 0.269 sec (expect)
thread 0: 0.382 sec (expect)
thread 1: 0.031 sec (measured)
thread 3: 0.204 sec (measured)
thread 2: 0.269 sec (measured)
thread 0: 0.382 sec (measured)
```

OpenMP Library Functions

```
/* controlling and monitoring thread creation
*/  void omp_set_dynamic (int dynamic_threads);
int omp_get_dynamic ();
void omp_set_nested (int nested);
int omp_get_nested ();

/* mutual exclusion */
void omp_init_lock (omp_lock_t *lock);
void omp_destroy_lock (omp_lock_t
void *lock);  omp_set_lock (omp_lock_t
void *lock);  omp_unset_lock (omp_lock_t
int omp_test_lock (omp_lock_t *lock);
    *lock);
```

In addition, all lock routines also have a nested lock counterpart for recursive mutexes.

Lock routines in OpenMP:

Simple locks:

- `void omp_init_lock(omp_lock_t* mutex)`
- `void omp_destroy_lock(omp_lock_t* mutex)`
- `void omp_set_lock(omp_lock_t* mutex)`
- `void omp_unset_lock(omp_lock_t* mutex)`

Initialize lock to unlocked state. Must be called before the lock can be used.

Ensure the lock is uninitialized. Must be called when the lock is not used anymore

Acquire the lock

Release the lock

Mutual exclusion using a lock

```c
#include <omp.h>

int main(int argc, char* argv[])
{
    omp_lock_t mutex;
    omp_init_lock(&mutex);
#pragma omp parallel
    {
        omp_set_lock(&mutex);
        // critical section
        omp_unset_lock(&mutex);
    }
    omp_destroy_lock(&mutex);
    return 0;
}
```

Mutual exclusion using a `critical` construct

```c
int main(int argc, char* argv[])
{
#pragma omp parallel
    {
#pragma omp critical
        {
            // critical section
        }
    }
    return 0;
}
```

Environment Variables in OpenMP

- `OMP NUM THREADS`: This environment variable specifies the default number of threads created upon entering a `parallel` region.

- `OMP SET DYNAMIC`: Determines if the number of threads can be dynamically changed.

- `OMP NESTED`: Turns on nested parallelism.

- `OMP SCHEDULE`: Scheduling of for-loops if the clause specifies runtime.

Note: The name of the environment variable must be upper case. Assigned values are case-insensitive and may have leading and/or trailing white space

OpenMP behavior:

- `OMP_DYNAMIC='true' or 'FALSE'`
- `OMP_NESTED='True' or ' false  '`

Set dynamic mode. Its default is implementation defined. If dynamic adjustment of the number of threads is supported it will be *true*, otherwise *false*.

Set nested parallelism. Its default is implementation defined. If nested parallelism is supported it will be *true*, otherwise *false*.

Schedule control:

- `OMP_SCHEDULE='schedule[, chunk_size]'`

Set the scheduling policy for the **`schedule`**`(runtime)` clause. The `omp_set_schedule()` API call can overwrite the value at runtime.

Thread control:

- `OMP_NUM_THREADS=positive_integer`

Set the maximum number of threads. The value of this environment variable can be obtained with the `omp_get_max_threads()` API call at runtime. The **`num_threads`**`(n)` clause overwrites the value for the given region. If dynamic mode is enabled, your code may run with fewer threads depending on available system resources.