OpenMP Supplementary

Scott Cheng

OpenMP Outline

- **Parallel Region Constructs**
    - `parallel` Directive
    - `simd` Directive
- Working-Sharing Constructs
    - `for` Directive
    - `sections` Directive
    - `single` Directive
- Task Constructs
- Synchronization Constructs
- Data Scope Attribute Clauses
- Run-Time Library Functions

Parallel Region Constructs: `parallel` Directive

- A parallel region is a block of code executed by multiple threads.

```
#pragma omp parallel [clause ...] if(scalar-expression) num_threads(integer-expression)
```

```
// A total of 6 “hello world!” is printed
#pragma omp parallel num_threads(2)
{
    #pragma omp parallel num_threads(3)
    {
        printf(“hello world!”);
    }
}
```

When *parallel* is reached, a team of threads is created. The parallel region code is duplicated and executed by all threads.

There is an **implicit barrier** at the end of a parallel section. One thread terminates, all threads terminate.

- It is illegal to branch (goto/return) into or out of a parallel region, but you could call other functions within a parallel region.

Parallel Region Constructs: How Many Threads?

The number of threads in a parallel region is determined in the following precedence:

1. Evaluation of the ***`if`*** clause
   - **If `false`, it is executed serially** by the master thread
   - E.g., `#pragma omp parallel` **`if`**`(is_parallel == 1)`
2. Setting of the ***`num_threads`*** clause
   - E.g., `#pragma omp parallel` **`num_threads`**`(10)`
3. Use of the ***`omp_set_num_threads()`*** library function (use **BEFORE** the parallel region)
4. Setting of the ***OMP_NUM_THREADS*** environment variable (set **BEFORE** the parallel region)
5. By default - **usually the number of core on the node**

Several ways to SIMD

- **Auto-vectorization**
  - loop vectorization
  - basic block vectorization
- Language extension/directives
  - **OpenMP `simd for`**
  - **OpenMP `simd declare`**
  - Cilk Plus
  - GCC vector extensions
- Intrinsics (e.g., `_mm512_maskz_fmadd_ps`)
- Assembly

Several ways to SIMD

- **Auto-vectorization**

```
void axpy_autovec(float a, float *x, float c, int N) {
    for (int i=0; i < N; i++) {
        x[i] = a * x[i] + c;
    }
}
```

	Report options
GCC	`-fopt-info-vec-{optimized,missed}`
Clang	`-R{pass,pass-missed,pass-analysis}=vectorize`
Intel	`-qopt-report-{phase,phase-missed}=vectorize`

Parallel Region Constructs: `simd` Directive

- Partition loop into chunks that fit a SIMD vector register

```
#pragma omp simd [clause …] if([simd :] scalar-expr) simdlen(length) collapse(n) …
```

```
#pragma omp simd reduction(+:res)
for (int i=0; i<N; i++) {
    res += a[i] * b[i];
}
```

```
#pragma omp simd for [clause …] schedule(type [,chunk]) collapse(n) …
```

```
#pragma omp simd for reduction(+:res)
for (int i=0; i<N; i++) {
    res += a[i] * b[i];
}
```

Parallel Region Constructs: `simd` Directive

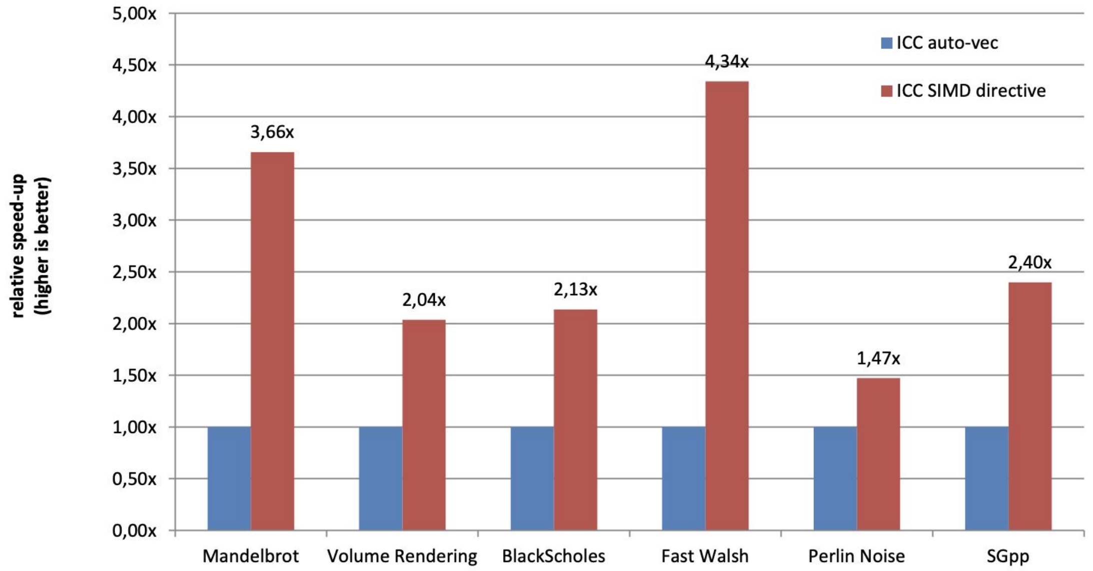
```
#pragma omp declare simd
float min(float a, float b) {
    return a < b ? a : b;
}
#pragma omp declare simd
float distsq(float x, float y) {
    return (x - y) * (x - y);
}
void example() {
#pragma omp parallel for simd
    for (int i=0; i<N; i++) {
      d[i] = min(distsq(a[i], b[i]), c[i]);
    }
}
```

```
vec min_v(vec a, vec b) {
    return a < b ? a : b;
}
```

```
vec distsq_v(vec x, vec y) {
    return (x - y) * (x - y);
}
```

```
vd = min_v(distsq_v(va, vb), vc)
```

SIMD Constructs & Performance



Why auto-vec failed?

- Data dependencies
- Alignment
- Function calls in loop block
- Complex control flow
- Conditional branches
- Loop bound not a constant
- Mixed data types
- Non-unit stride between elements
- Register pressure in loop body

and more...

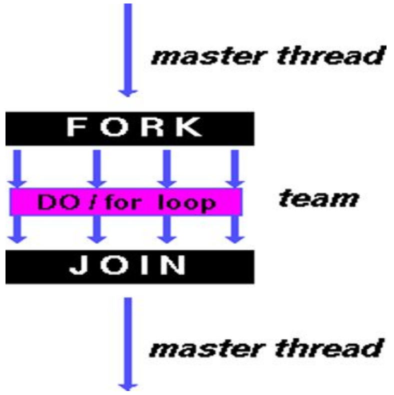
M.Klemm, A.Duran, X.Tian, H.Saito, D.Caballero, and X.Martorell. Extending OpenMP with Vector Constructs for Modern Multicore SIMD Architectures. In Proc. of the Intl. Workshop on OpenMP, pages 59-72, Rome, Italy, June 2012. LNCS 7312.

OpenMP Outline

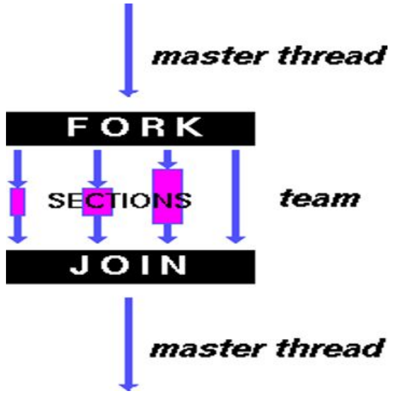
- Parallel Region Constructs
  - `parallel` Directive
  - `simd` Directive
- **Working-Sharing Constructs**
  - **`for` Directive**
  - **`sections` Directive**
  - **`single` Directive**
- Task Constructs
- Synchronization Constructs
- Data Scope Attribute Clauses
- Run-Time Library Functions

Work-Sharing Constructs

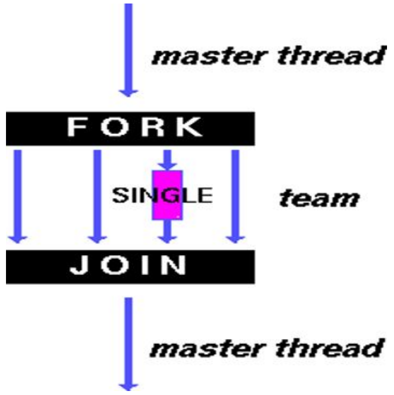
for - shares iterations of a loop across the team.



sections *- break works into sections. Each section is executed by a thread.*



single *-* ***serializes*** *a section of code by running with a* ***single thread****.*



- Work-sharing constructs **DO NOT** launch new threads.
- There is no implied barrier upon entry to a work-sharing construct, however there is an implied barrier at the end of a work sharing construct.

Work-Sharing Constructs: for Directive

```
#pragma omp for [clause …] schedule(type [,chunk]) ordered nowait collapse(n)
```

- `nowait`: Do not synchronize threads at the end of the loop.
- `schedule`: Describes how iterations are divided among threads.
- `ordered`: Iterations must be executed as in a serial program.
- `collapse`: Specifies how many nested loop should be collapsed into one large iteration space.

```
int chunk_size = 100;
#pragma omp parallel num_thread(2) shared(a,b,c) private(i)
{
    #pragma omp for schedule(dynamic, chunk_size) nowait
    for (int i=0; i < N; i++)
        c[i] = a[i] + b[i];
}
```

Work-Sharing Constructs: `sections` Directive

- It specifies that the enclosed sections are to be divided among the threads in the team. Each *`section`* is executed **ONCE** by **ONE** thread.

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

Work-Sharing Constructs: `single` Directive

- It specifies that the enclosed code is to be executed by only one thread.
- Useful when dealing with sections of code that are not thread-safe, such as I/O.
- Threads that do not execute the *`single`* directive, wait at the end of the enclosed code block, unless the *`nowait`* clause is specified.

```
#pragma omp parallel num_thread(10) shared(a)
{
    a[i] += calc_fn(a[i]);
    #pragma omp single
    {
        for (int i=0; i < N; i++) fprintf(f, “%d ”, a[i]);
    }
}
```

OpenMP Outline

- Parallel Region Constructs
  - `parallel` Directive
  - `simd` Directive
- Working-Sharing Constructs
  - `for` Directive
  - `sections` Directive
  - `single` Directive
- **Task Constructs**
- Synchronization Constructs
- Data Scope Attribute Clauses
- Run-Time Library Functions

Task Constructs: task Directive

```
#pragma omp task [clause …] if(scalar-expr) final(scalar-expr) untied mergeable
```

```
int fib(int n) {
    if (n < 2) return n;
    int x, y;
    #pragma omp taskgroup
    {
        #pragma omp task shared(x)
        {
            x = fib(n - 1);
        }
        #pragma omp task shared(y)
        {
            y = fib(n - 2);
        }
    }
    return x + y;
}
```

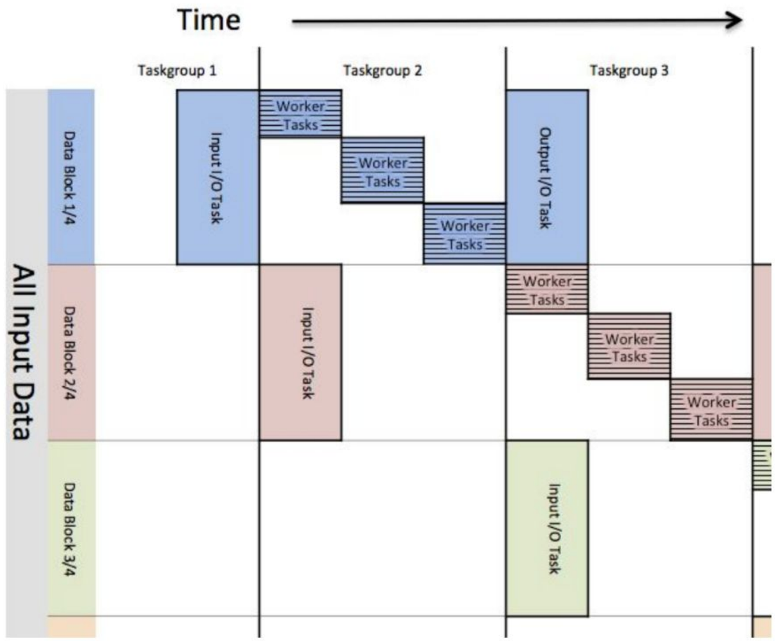
```
// in main/caller
#pragma omp parallel
{
    #pragma omp master
    {
        printf("%d\n", fib(n));
    }
}
```

Scheduling

- Default: Tasks are tied to the thread that first executes them (normally not the creator). Scheduling constraints:
  - Only the thread to which a task is tied can execute it
  - A task can only be suspended at task scheduling points
    - task creation, task finish, `taskwait`, `barrier`, `taskyield`
  - If task is not suspended in a barrier, the executing thread can only switch to a direct descendant of a task tied to the thread
- Tasks created with the *`untied`* clause are never tied
  - Allowed to resume at task scheduling points in a different thread
  - No scheduling restrictions, e.g., can be suspended at any point
  - Gives more freedom to the implementation, e.g., load balancing

HMMER3: Use task and taskgroup to Overlap I/O and Compute

```
#pragma omp parallel {
    #pragma omp single {
        #pragma omp task { load_seq_buffer(); }
        #pragma omp task { load_hmm_buffer(); }
        #pragma omp taskwait
        while( more HMMs ) {
            #pragma omp task { write_output();
                               load_hmm_buffer(); }
            while( more sequences ) {
                #pragma omp taskgroup {
                    #pragma omp task {
                    load_seq_buffer(); }
                    for ( each hmm in hmm_buffer )
                        #pragma omp task {
                        task_kernel(); }
                    swap_I/O_and_working_seq_buffers()
                    ;
                }
            }
            #pragma omp taskwait
            swap_I/O_and_working_hmm_buffers();
        }
    }
}
```



Courtesy of William Arndt, NERSC

OpenMP Outline

- Parallel Region Constructs
  - `parallel` Directive
  - `simd` Directive
- Working-Sharing Constructs
  - `for` Directive
  - `sections` Directive
  - `single` Directive
- Task Constructs
- **Synchronization Constructs**
- Data Scope Attribute Clauses
- Run-Time Library Functions

Synchronization Constructs

`#pragma omp [synchronization_directive] [clause …]`

Synchronization Directives:

- **`master`**: only executed by the master thread. No implicit barrier at the end. More efficient than `single` directive.
- **`critical`**: must be executed by **only one thread at a time**. Threads will be blocked until the critical section is clear.
- **`barrier`**: blocked until all threads reach the call.
- **`atomic`**: memory location must be updated atomically provide a mini-critical section.

```
int count=0;
#pragma omp parallel num_thread(10)
    #pragma omp critical
        count++;
```

```
int count=0;
#pragma omp parallel num_thread(10)
    #pragma omp atomic
        count++;
```

OpenMP Library Function: Lock

- `void omp_init_lock(omp_lock_t *lock)`
    - Initializes a lock associated with the lock variable
- `void omp_destroy_lock(omp_lock_t *lock)`
    - Disassociates the given lock variable from any locks
- `void omp_set_lock(omp_lock_t *lock)`
    - Force the thread to wait until the specified lock is available
- `void omp_unset_lock(omp_lock_t *lock)`
    - Releases the lock from the executing subroutine
- `int omp_test_lock(omp_lock_t *lock)`
    - Attempts to set a lock, but does NOT block if unavailable

OpenMP Outline

- Parallel Region Constructs
  - `parallel` Directive
  - `simd` Directive
- Working-Sharing Constructs
  - `for` Directive
  - `sections` Directive
  - `single` Directive
- Task Constructs
- Synchronization Constructs
- **Data Scope Attribute Clauses**
- Run-Time Library Functions

Data Scope Attribute Clauses

- `default(private | firstprivate | shared | none)`: Allows the user to specify a default scope for ALL variables in the parallel region.
- `private(var_list)`: Declares variables in its list to be **private to each thread**; variable value is **NOT initialized & will not be maintained outside the parallel region**.
- `shared(var_list)`: Declares variables in its list to be **shared among all threads**. By default, all variables in the work sharing region are shared except the loop iteration counter.
- `firstprivate(var_list)`: Same as *`private`* clause, but the **variable is INITIALIZED** according to the value of their original objects prior to entry into the parallel region.
- `lastprivate(var_list)`: Same as *`private`* clause, with a **copy from the LAST loop iteration or section to the original variable object.**

Examples

```
int var1 = 10;
#pragma omp parallel firstprivate(var1)
{
    printf(“var1: %d” var1);
}
```

```
int var1 = 10;
#pragma omp parallel lastprivate(var1)
{
    int id = omp_get_thread_num();
    sleep(id);
    var1 = id;
}
printf(“var1: %d”, var1);
```

OpenMP Outline

- Parallel Region Constructs
  - `parallel` Directive
  - `simd` Directive
- Working-Sharing Constructs
  - `for` Directive
  - `sections` Directive
  - `single` Directive
- Task Constructs
- Synchronization Constructs
- Data Scope Attribute Clauses
- **Run-Time Library Functions**

Run-Time Library Functions

- `void omp_set_num_threads(int num_threads)`
  - Sets the number of threads that will be used in the next parallel region
- `int omp_get_num_threads(void)`
  - Returns the number of threads currently executing for the parallel region
- `int omp_get_thread_num(void)`
  - Returns the thread number of the thread, within the team, making this call
  - The master thread of the team is thread 0
- `int omp_get_num_procs(void)`
  - Returns the number of processors that are available to the program
- `int omp_in_parallel(void)`
  - determine if the section of code which is executing is parallel or not

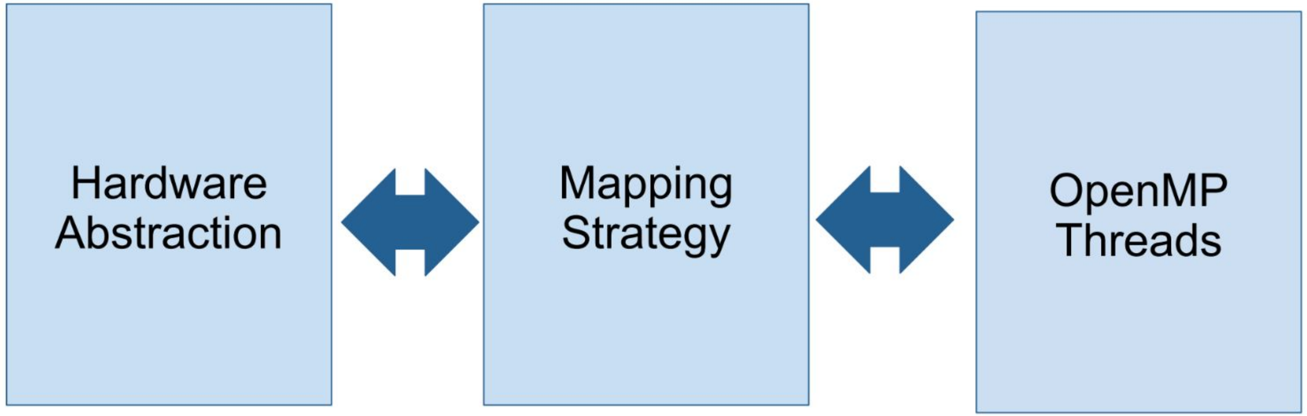
And more ...

Environment Variables

- `OMP_SCHEDULE`: determines how loop iterations are scheduled on processors
- `OMP_NUM_THREADS`: sets the maximum number of threads during execution
- `OMP_DYNAMIC`: enables or disables dynamic adjustment of the number of threads available for execution of parallel regions
- `OMP_PROC_BIND`: enables or disables threads binding to processors
- `OMP_NESTED`: enables or disables nested parallelism
- `OMP_STACKSIZE`: controls the stack size for created (non-master) threads

And more ...

NUMA-awareness: Thread Affinity



OMP_PLACES
Environment Variable
(e.g. threads, cores, sockets)

OMP_PROC_BIND
Environment Variable
Or
proc_bind() clause
of parallel region
(e.g. close, spread, master)

OMP_NUM_THREADS
Environment Variable
Or
num_threads() clause
of parallel region

Considerations for OMP_PROC_BIND Choices

- Selecting the "right" binding is dependent on the architecture topology but also on the application characteristics
- Putting threads apart (e.g. different sockets): spread
  - Can help to improve aggregated memory bandwidth
  - Combine the cache sizes across cores
  - May increase the overhead of synchronization across far apart threads
  - Aggregates memory bandwidth to/from accelerator(s)
- Putting threads near (e,g. hardware threads or cores sharing caches): master, close
  - Good for synchronization and data reuse
  - May decrease total memory bandwidth

Reference

- OpenMP reference guide: https://www.openmp.org/resources/refguides/
- LLNL OpenMP tutorial: https://hpc-tutorials.llnl.gov/openmp/
- GNU OpenMP
- Intel OpenMP
- Slides
    - https://www.nic.uoregon.edu/iwomp2005/iwomp2005_tutorial_openmp_rvdp.pdf
    - https://www.nersc.gov/assets/Uploads/SC16-Programming-Irregular-Applications-with-OpenMP.pdf
    - https://openmpcon.org/wp-content/uploads/openmpcon2017/Tutorial2-Advanced_OpenMP.pdf
    - https://openmpcon.org/wp-content/uploads/2018_Session2_Hernandez.pdf
    - https://openmpcon.org/wp-content/uploads/2018_Tutorial3_Martorell_Teruel_Klemm.pdf
    - https://www.eidos.ic.i.u-tokyo.ac.jp/~tau/lecture/parallel_distributed/2018/slides/pdf/simd2.pdf