
Shared Memory Parallelism - OpenMP

Sathish Vadhiyar

Credits/Sources:

OpenMP C/C++ standard (openmp.org)

OpenMP tutorial (<http://www.llnl.gov/computing/tutorials/openMP/#Introduction>)

OpenMP sc99 tutorial presentation (openmp.org)

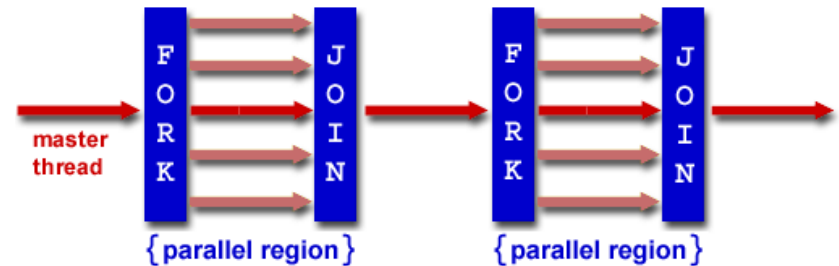
Dr. Eric Strohmaier (University of Tennessee, CS594 class, Feb 9, 2000)

Introduction

- A portable programming model and standard for shared memory programming using compiler directives
 - Directives?: constructs or statements in the program applying some action on a block of code
 - A specification for a set of compiler directives, library routines, and environment variables – standardizing pragmas
 - Easy to program; easy for code developer to convert his sequential to parallel program by throwing directives
 - First version in 1997, development over the years till the latest 4.5 in 2015
-

Fork-Join Model

- Begins as a single thread called master thread
- **Fork:** When **parallel** construct is encountered, team of threads are created
- Statements in the parallel region are executed in parallel
- **Join:** At the end of the parallel region, the team threads synchronize and terminate



OpenMP consists of...

- Work-sharing constructs
 - Synchronization constructs
 - Data environment constructs
 - Library calls, environment variables
-

Introduction

- Mainly supports *loop-level parallelism*
 - Specifies parallelism for a region of code: *fine-level parallelism*
 - The number of threads can be varied from one region to another – *dynamic parallelism*
 - Follows Amdahl's law – sequential portions in the code
 - Applications have varying phases of parallelism
 - Also supports
 - Coarse-level parallelism – sections and tasks
 - Executions on accelerators
 - SIMD vectorizations
-
- task-core affinity

parallel construct

```
#pragma omp parallel [clause [, clause] ...] new-line  
    structured-block
```

Clause:

```
if ([parallel :] scalar-expression)
```

```
num_threads (integer-expression)
```

```
default (shared | none)
```

```
private (list)
```

```
firstprivate (list)
```

```
shared (list)
```

```
copyin (list)
```

```
reduction (reduction-identifier : list)
```

```
proc_bind(master | close | spread)
```

Can support nested
parallelism

Parallel construct - Example

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

```
main () {
```

```
int nthreads, tid;
```

```
    #pragma omp parallel private(nthreads, tid) {
```

```
        ...
```

```
    }
```

```
}
```

Work sharing construct

- For distributing the execution among the threads that encounter it
 - 3 types of work sharing constructs – loops, sections, single
-

for construct

- For distributing the iterations among the threads

`#pragma omp for [clause [, clause] ...] new-line`

for-loop

Clause:

`private (list)`

`firstprivate (list)`

`lastprivate (list)`

`linear (list[: linear-step])`

`reduction (reduction-identifier : list)`

`schedule ([modifier [, modifier]:] kind[, chunk_size])`

`collapse (n)`

`ordered[(n)]`

`nowait`

for construct

- Restriction in the structure of the for loop so that the compiler can determine the number of iterations – e.g. no branching out of loop
- The assignment of iterations to threads depends on the **schedule** clause
- Implicit barrier at the end of **for** if not **nowait**

schedule clause

1. `schedule(static, chunk_size)` – iterations/`chunk_size` chunks distributed in round-robin
 2. `Schedule(dynamic, chunk_size)` – same as above, but chunks distributed dynamically.
 3. `schedule(runtime)` – decision at runtime. Implementation dependent
-

for - Example

```
include <omp.h>
#define CHUNKSIZE 100
#define N 1000

main () {
int i, chunk; float a[N], b[N], c[N];

/* Some initializations */
for (i=0; i < N; i++)
    a[i] = b[i] = i * 1.0;

chunk = CHUNKSIZE;
#pragma omp parallel shared(a,b,c,chunk) private(i) {
    #pragma omp for schedule(dynamic,chunk) nowait
    for (i=0; i < N; i++)
        c[i] = a[i] + b[i];
    } /* end of parallel section */
}
```

Coarse level parallelism – sections and tasks

- sections

```
#pragma omp parallel sections \<clause-list\>
{
    #pragma omp section
    structure-block i
    #pragma omp section
    structure-block j
    ...
}
```

- tasks – dynamic mechanism

```
#pragma omp parallel \<clause-list\>
{
    ...
    #pragma omp task \<clause-list\>
    ...
}
```

- depend clause for task

```
depend (dependence type : variable\_list)
```

Synchronization directives

```
#pragma omp master new-line  
    structured-block
```

```
#pragma omp critical [(name)] new-line  
    structured-block
```

```
#pragma omp barrier new-line
```

```
#pragma omp atomic new-line  
    expression-stmt
```

```
#pragma omp flush [(variable-list)] new-line
```

```
#pragma omp ordered new-line  
    structured-block
```

Data Scope Attribute Clauses

Most variables are shared by default

Data scopes explicitly specified by data scope attribute clauses

Clauses:

1. private
 2. firstprivate
 3. lastprivate
 4. shared
 5. default
 6. reduction
 7. copyin
 8. copyprivate
-

private, firstprivate & lastprivate

- private (*variable-list*)
 - variable-list private to each thread
 - A new object with automatic storage duration allocated for the construct
 - firstprivate (*variable-list*)
 - The new object is initialized with the value of the old object that existed prior to the construct
 - lastprivate (*variable-list*)
 - The value of the private object corresponding to the last iteration or the last section is assigned to the original object
-

shared, default, reduction

- `shared(variable-list)`
 - `default(shared | none)`
 - Specifies the sharing behavior of all of the variables visible in the construct
 - `Reduction(op: variable-list)`
 - Private copies of the variables are made for each thread
 - The final object value at the end of the reduction will be combination of all the private object values
-

Library Routines (API)

- Querying function (number of threads etc.)
 - General purpose locking routines
 - Setting execution environment (dynamic threads, nested parallelism etc.)
-

API

- `OMP_SET_NUM_THREADS(num_threads)`
 - `OMP_GET_NUM_THREADS()`
 - `OMP_GET_MAX_THREADS()`
 - `OMP_GET_THREAD_NUM()`
 - `OMP_GET_NUM_PROCS()`
 - `OMP_IN_PARALLEL()`
 - `OMP_SET_DYNAMIC(dynamic_threads)`
 - `OMP_GET_DYNAMIC()`
-

API(Contd..)

- `omp_init_lock(omp_lock_t *lock)`
- `omp_init_nest_lock(omp_nest_lock_t *lock)`
- `omp_destroy_lock(omp_lock_t *lock)`
- `omp_destroy_nest_lock(omp_nest_lock_t *lock)`
- `omp_set_lock(omp_lock_t *lock)`
- `omp_set_nest_lock(omp_nest_lock_t *lock)`
- `omp_unset_lock(omp_lock_t *lock)`
- `omp_unset_nest__lock(omp_nest_lock_t *lock)`
- `omp_test_lock(omp_lock_t *lock)`
- `omp_test_nest_lock(omp_nest_lock_t *lock)`

- `omp_get_wtime()`
- `omp_get_wtick()`

- `omp_get_thread_num()`
- `omp_get_num_proc()`
- `omp_get_num_devices()`

Example 1: Jacobi Solver

```
1 #include "omp.h"
2 int main(int argc, char** argv){
3     ...
4     int rows, cols;
5     int* grid;
6     int chunk_size, threads=16;
7     ...
8
9     /* Allocate and initialize the grid */
10    grid = malloc(sizeof(int)*N*N);
11    for(i=0; i<N; i++){
12        for(j=0; j<N; j++){
13            grid[i*cols+j] = ...;
14        }
15    }
16
17    chunk_size = N/threads;
18    # pragma omp parallel for num_threads(16) for private(i,j)
19        shared(rows,cols,grid) schedule(static,chunk_size)
20        collapse(2)
21    for(i=1; i<rows-1; i++){
22        for(j=1; j<cols-1; j++){
23            grid[i*N+j] = 1/4 * (grid[i*N+j-1] + grid[i*N+j+1] +
24                grid[(i-1)*N+j] + grid[(i+1)*N+j]);
25        }
26    }
27 }
```

Example 2: BFS Version 1 (Nested Parallelism)

```
1  ...
2  level[0] = s;
3  curLevel = 0;
4  dist[s]=0; dist[v!=s]=-1;
5
6  while (level[curLevel] != NULL){
7      # pragma omp parallel for ....
8      for(v in level[curLevel]){
9          v=level[curLevel][i];
10         neigh=neighbors(v);
11
12         # pragma omp parallel for ....
13         for(j=0; j<length(neigh); j++){
14             w=neigh[j];
15             if(dist[w] == -1){
16                 level[curLevel + 1] = union(level[curLevel + 1], w);
17                 dist[w] = dist[v] + 1;
18             }
19         }
20     }
21 }
22 ...
```

Example 3: BFS Version 2 (Using Task Construct)

```
1  ...
2  level[0] = s;
3  curLevel = 0;
4  dist[s]=0; dist[v!=s]=-1;
5
6  while(level[curLevel] != NULL){
7      # pragma omp parallel ....
8      for(v in level[curLevel]){
9          for(w in neighbors(v)){
10             # pragma omp task ...
11             {
12                 if(dist[w] == -1){
13                     level[curLevel + 1] = union(level[curLevel + 1], w)
14                     ;
15                     dist[w] = dist[v] + 1;
16                 }
17             }
18         }
19     }
20  ...
```

- END
