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

```c
#include <omp.h>

main () {
int nthreads, tid;

#pragma omp parallel private(nthreads, tid) {
    printf("Hello World \n");
}
}
```
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

```c
#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

```c
#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

```c
#pragma omp parallel sections 
{ 
    #pragma omp section 
    structure-block i
    #pragma omp section 
    structure-block j
    ...
}
```

- tasks – dynamic mechanism

```c
#pragma omp parallel 
{ 
    ...
    #pragma omp task 
    ...
}
```

- depend clause for task

```c
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
flush directive

- Point where consistent view of memory is provided among the threads
- Thread-visible variables (global variables, shared variables etc.) are written to memory
- If var-list is used, only variables in the list are flushed
flush - Example

```c
int sync[NUMBER_OF_THREADS];
float work[NUMBER_OF_THREADS];
#pragma omp parallel private(iam,neighbor) shared(work,sync)
{
    iam = omp_get_thread_num();
    sync[iam] = 0;
    #pragma omp barrier

    /*Do computation into my portion of work array */
    work[iam] = ...;

    /* Announce that I am done with my work
    * The first flush ensures that my work is
    * made visible before sync.
    * The second flush ensures that sync is made visible.
    */
```
flush – Example (Contd…) 

```c
#pragma omp flush(work)
sync[iam] = 1;
#pragma omp flush(sync)

/*Wait for neighbor*/
neighbor = (iam>0 ? iam : omp_get_num_threads()) - 1;
while (sync[neighbor]==0) {
    #pragma omp flush(sync)
}

/*Read neighbor's values of work array */
... = work[neighbor];
```
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
threadprivate

• Global variable-list declared are made private to a thread
• Each thread gets its own copy
• Persist between different parallel regions

```
#include <omp.h>
int alpha[10], beta[10], i;
#pragma omp threadprivate(alpha)
main () {
  /* Explicitly turn off dynamic threads */
  omp_set_dynamic(0);
  /* First parallel region */
  #pragma omp parallel private(i,beta)
  for (i=0; i < 10; i++) alpha[i] = beta[i] = i;
  /* Second parallel region */
  #pragma omp parallel
```
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
int x, y, z[1000];
#pragma omp threadprivate(x)

void fun(int a) {
    //O.K. - j is declared within parallel region
    int j = omp_get_num_thread();
    int i = 0;
    const int c = 1;
    int i = 0;
    #pragma omp parallel default(none) private(a) shared(z)
    {
        int j = omp_get_num_thread();
        //O.K. - j is declared within parallel region
        a = z[j];

        x = c;

        z[i] = y;
    }
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()
- OMP_SET_NESTED(nested)
- OMP_GET_NESTED()
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()
Simple locks and nestable locks

Simple locks are not locked if they are already in a locked state

Nestable locks can be locked multiple times by the same thread

Simple locks are available if they are unlocked

Nestable locks are available if they are unlocked or owned by a calling thread
#include <omp.h>
typedef struct { int a, b; omp_nest_lock_t lck; } pair;

void incr_a(pair *p, int a)
{
   // Called only from incr_pair, no need to lock.
   p->a += a;
}

void incr_b(pair *p, int b)
{
   // Called both from incr_pair and elsewhere,
   // so need a nestable lock.

   omp_set_nest_lock(&p->lck);
   p->b += b;
   omp_unset_nest_lock(&p->lck);
}
Example – Nested lock (Contd..)

```c
void incr_pair(pair *p, int a, int b)
{
    omp_set_nest_lock(&p->lck);
    incr_a(p, a);
    incr_b(p, b);
    omp_unset_nest_lock(&p->lck);
}

void f(pair *p)
{
    extern int work1(), work2(), work3();
    #pragma omp parallel sections
    {
        #pragma omp section
            incr_pair(p, work1(), work2());
        #pragma omp section
            incr_b(p, work3());
    }
```
Example 1: Jacobi Solver

```c
#include "omp.h"

int main(int argc, char** argv)
{
    ... 
    int rows, cols;
    int* grid;
    int chink_size, threads = 16;
    ...

    /* Allocate and initialize the grid */
    grid = malloc(sizeof(int*)*N*N);
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            grid[i*cols+j] = ...;
        }
    }

    chunk_size = N/threads;

    #pragma omp parallel for num_threads(16) for private(i, j)
    shared(rows, cols, grid) schedule(static, chunk_size)
    collapse(2)
    for (i = 1; i < rows - 1; i++) {
        for (j = 1; j < cols - 1; j++) {
            grid[i*N+j] = 1/4 * (grid[i*N+j-1] + grid[i*N+j+1] +
                                 grid[(i-1)*N+j] + grid[(i+1)*N+j]);
        }
    }
```
Example 2: BFS Version 1
(Nested Parallelism)

```c
... level[0] = s;
curLevel = 0;
dist[s]=0; dist[v!=s]=−1;

while (level[curLevel] != NULL){
    # pragma omp parallel for ...
    for (i=0; i<length(level[curLevel]); i++){
        v = level[curLevel][i];
        neigh = neighbors(v);

        # pragma omp parallel for ...
        for (j=0; j<length(neigh); j++){
            w = neigh[j];
            if (dist[w] = 1)
                level[curLevel + 1] = union(level[curLevel + 1], w);
            dist[w] = dist[v] + 1;
        }
    }
}
...
Example 3: BFS Version 3 (Using Task Construct)

```c
...
level[0] = s;
curLevel = 0;
dist[s] = 0; dist[v!=s] = -1;

while (level[curLevel] != NULL)
{
    # pragma omp parallel ....
    for (v in level[curLevel])
    {
        for (w in neighbors(v))
        {
            # pragma omp task ...
            {
                if (dist[w] = 1)
                {
                    level[curLevel + 1] = union(level[curLevel + 1], w)
                    ;
                    dist[w] = dist[v] + 1;
                }
            }
        }
    }
}
```

...
Hybrid Programming – Combining MPI and OpenMP benefits

- **MPI**
  - explicit parallelism, no synchronization problems
  - suitable for coarse grain

- **OpenMP**
  - easy to program, dynamic scheduling allowed
  - only for shared memory, data synchronization problems

- **MPI/OpenMP Hybrid**
  - Can combine MPI data placement with OpenMP fine-grain parallelism
  - Suitable for cluster of SMPs (Clumps)
  - Can implement hierarchical model
END
Definitions

- **Construct** – statement containing directive and structured block
- **Directive** – Based on C #pragma directives

```c
#pragma <omp id> <other text>
#pragma omp directive-name [clause [, clause] …] new-line
```

**Example:**
```c
#pragma omp parallel default(shared) private(beta,pi)
```
Parallel construct

- Parallel region executed by multiple threads
- If num_threads, omp_set_num_threads(), OMP_SET_NUM_THREADS not used, then number of created threads is implementation dependent
- Number of physical processors hosting the thread also implementation dependent
- Threads numbered from 0 to N-1
- Nested parallelism by embedding one parallel construct inside another