Parallel Programming Practice

- **Current**
  - Start with a parallel algorithm
  - Implement, *keeping in mind*
    - Data races
    - Synchronization
    - Threading syntax
  - Test & Debug
  - Debug ....

- **Ideal way**
  - Start with some algorithm
  - Implement serially, *ignoring*
    - Data races
    - Synchronization
    - Threading syntax
  - Test & Debug
  - *Auto-magically* parallelize
Implementation on Shared Memory

- **Thread Library**
  - Library calls
  - Low level programming
    - Explicit thread creation & work assignment
    - Explicit handling of synchronization
  - Parallelism expression
    - Task: create/join thread
    - Data: detailed programming
  - Design concurrent version from the start

- **OpenMP**
  - Compiler directives
  - Higher abstraction
    - Compilers convert code to use OpenMP library, which is actually implemented with thread APIs
  - Parallelism expression
    - Task: task/taskwait, parallel sections
    - Data: parallel for
  - Incremental development
    - Start with sequential version
    - Insert necessary directives
Implementation Examples

- Threaded functions
  - Exploit data parallelism

```c
node A[N], B[N];

main() {
    for (i=0; i<nproc; i++)
        thread_create(par_distance);
    for (i=0; i<nproc; i++)
        thread_join();
}

void par_distance() {
    tid = thread_id();
    n = ceiling(N/nproc);
    s = tid * n;
    e = MIN((tid+1)*n, N);
    for (i=s; i<e; i++)
        for (j=0; j<N; j++)
            C[i][j] = distance(A[i], B[j]);
}
```

- Parallel loops
  - Exploit data parallelism

```c
node A[N], B[N];

#pragma omp parallel for
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        C[i][j] = distance(A[i], B[j]);
```
Implementation on Distributed Memory

- MPI (message passing interface)
  - Language independent communication library
  - Freely available implementation
    - MPICH (Argonne Lab), Open MPI

```c
/* processor 0 */
node A[N], B[N];

Dist_calc0() {
  Send (A[N/2 .. N-1], B[0 .. N-1]);

  for (i=0; i<N/2; i++)
    for (j=0; j<N; j++)
      C[i][j] = distance(A[i], B[j]);

 Recv (C[N/2 .. N-1][0 .. N-1]);
}

/* processor 1 */
node A[N], B[N]; /* duplicate copies */

Dist_calc1() {
  Recv (A[N/2 .. N-1], B[0 .. N-1]);

  for (i=N/2; i<N; i++)
    for (j=0; j<N; j++)
      C[i][j] = distance(A[i], B[j]);

  Send (C[N/2 .. N-1][0 .. N-1]);
}
```
OpenMP Solution Stack

- User
- Application
- OpenMP Program Layer:
  - Directives, Compiler
  - OpenMP Library
  - Environment Variables
- System Layer:
  - Runtime Library
  - OS/System support with Shared memory
OpenMP Programming Model

- **Fork-join model**
  - Thread pool
  - Implicit barrier
  - `#pragma omp`
    - parallel for
    - parallel sections

- Data scoping semantics are somewhat complicated
  - private, shared, copyin, firstprivate, lastprivate, copyprivate, threadprivate, …
  - Implicit rules,…
Parallel Loops and Reductions

- Data level parallelism
  - Parallel loop
    - No communication once parallel region starts (no loop-carried dep.)

```c
for (i=0; i<N; i++) {
    a[i] = b[i] + c[i];
}
```

- Reduction
  - Associative operations

```c
sum = 0;
for (i=0; i<N; i++) {
    sum = sum + a[i];
}
```

```
#pragma omp parallel for
for (i=0; i<N; i++) {
    a[i] = b[i] + c[i];
}
```

```
sum = 0;
#pragma omp parallel for reduction(+:sum)
for (i=0; i<N; i++) {
    sum = sum + a[i];
}
```
Implementation with Thread API

- **Parallel loop**

  ```c
  main() {
    for (i=0; i<nproc; i++)
      thread_create(par_distance);
    for (i=0; i<nproc; i++)
      thread_join();
  }
  void par_distance() {
    tid = thread_id();    n = ceiling(N/nproc);
    s = tid * n;          e = MIN((tid+1)*n, N);
    for (i=s; i<e; i++)
      for (j=0; j<N; j++)
        C[i][j] = distance(A[i], B[j]);
  }
  ```

- **Reduction**

  ```c
  main() {
    for (i=0; i<nproc; i++)
      thread_create(par_sum);
    for (i=0; i<nproc; i++)
      thread_join();
    sum = 0;
    for (i=0; i<nproc; i++)
      sum = sum + local_sum[i];
  }
  void par_sum() {
    tid = thread_id();    n = ceiling(N/nproc);
    s = tid * n;          e = MIN((tid+1)*n, N);
    tmp = 0;
    for (i=s; i<e; i++)
      tmp = tmp + A[i];
    local_sum[tid] = tmp;
  }
  ```
OpenMP Parallel Sections

- Different threads execute different works

```c
#pragma omp parallel sections
{
    #pragma omp section
    A();

    #pragma omp section
    B();
}
/* end of parallel sections */
```
OpenMP Single

- Executed by one thread within a parallel region
  - Any thread can execute the single region
  - Implicit barrier synchronization at the end

```c
#pragma omp parallel
{
    #pragma omp single
    {
        a = 10;
    } /* implicit barrier */

    #pragma omp for
    for (i=0; i<N; i++)
        B[i] = a;

/* end of parallel region */
```
OpenMP Master

- Executed by the master thread
- No implicit barrier
- If a barrier is needed for correctness, must specify one

```c
#pragma omp parallel
{
    #pragma omp master
    {
        a = 10;
    } /* no barrier */
    #pragma omp barrier

    #pragma omp for
    for (i=0; i<N; i++)
        B[i] = a;
}
/* end of parallel region */
```
Load Imbalance

- **OpenMP schedule**
  - `#pragma omp parallel for schedule (type [, chunk])`
  - *type* := static, dynamic, guided, runtime
  - *chunk* := positive integer
  - **static**
    - Divide iterations by *chunk* (near equal in size by default)
    - Statically assign threads in a round-robin fashion
  - **dynamic**
    - Divide iterations by *chunk* (1 by default)
    - Dynamically assign a chunk to an idle thread (master/worker)
  - **guided**
    - Chunk size is reduced in an exponentially decreasing manner
    - Dynamically assign a chunk to an idle thread (master/worker)
    - Minimum chunk size is specified by *chunk* (1 by default)
  - **runtime**
    - Determined at runtime with OMP_SCHEDULE environment variable
Load Imbalance

- **OpenMP schedules**
  - **Static schedule on iteration space**
    - 0
    - $\frac{1}{4}N$
    - $\frac{1}{2}N$
    - $\frac{3}{4}N$
    - N-1
  - **Dynamic schedule on iteration space (master/worker)**
  - **Guided schedule on iteration space (master/worker)**
OpenMP Programming Model

- Task model (OpenMP 3.0 – released, May 2008)
  - Task creation and join
  - Can handle
    - Unbounded loops
    - Recursive algorithms
    - Producer/consumer

- \#pragma omp
  - task
  - taskwait

- (NOTE) parallel sections use fork-join model
  - Not suitable for above mentioned jobs
OpenMP Examples

- Task level parallelism
  ```c
  void traverse (NODE *p) {
    if (p->left)    
      traverse(p->left);
    if (p->right)   
      traverse(p->right);
    process(p);
  }
  ```

- Post-order visit?
- Individual join?
- Join all the descendant tasks?
  - Join all the task created so far
  - Taskgroup is needed (Not defined in OpenMP 3.0)
Implementation with Thread API

- Easier implementation, compared to data parallelism
- Threads can be joined individually

```c
traverse(NODE *p) {
    if (p->left)
        tid[0] = thread_create(traverse, p->left);
    if (p->right)
        tid[1] = thread_create(traverse, p->right);

    for (i=0; i<2; i++) thread_join(tid[i]); /* post-order visit */

    process(p);
}
```
Synchronization

- Correct order of thread execution
  - Different order may result in incorrect results

- Mutual Exclusion
  - Avoid race condition
  - Correct updates for shared objects

- Event Synchronization
  - Coordinate multiple thread execution
  - Barrier/join after parallel computation
  - Wait event notification
Mutual Exclusion

- **Thread API**

  ```c
  main() {
    for (i=0; i<nproc; i++)
      thread_create(par_dequeue);
    for (i=0; i<nproc; i++)
      thread_join();
  }
  void par_dequeue() {
    tid = thread_id();   n = ceiling(N/nproc);
    s = tid * n;         e = MIN((tid+1)*n, N);
    for (i=s; i<e; i++) {
      if (x[i] == cond) {
        thread_mutex_lock(queue_lock);
        dequeue(x[i]);
        thread_mutex_unlock(queue_lock);
      }
    }
  }
  ```

- **OpenMP**

  - Exclusive accesses among the team of threads
  - ```pragma omp```
    - ```critical        structured_block```
    - ```atomic          statement```

  ```c
  main() {
    #pragma omp parallel for
    for (i=0; i<N; i++) {
      if (x[i] == cond)
        #pragma omp critical
        dequeue(x[i]);
    }
  }
  ```
Event Synchronization

- **Thread API**
  - Manipulated control among individual threads
  - Condition variable
    - `pthread_cond_wait()`
    - `pthread_cond_signal()`
  - Barrier can be implemented with
    - Mutex lock’ed counter
    - Condition variable wait/signal

- **OpenMP**
  - **Barrier synchronization**
    - Wait until all the threads in a team reach to a point
  - `#pragma omp barrier`

```c
main() {
  #pragma omp parallel
  sub();
}
sub() {
  work1();
  #pragma omp barrier
  work2();
}
```
Other Components of OpenMP

- Runtime library
  - `omp_get_num_threads();` `omp_set_num_threads();`
  - `omp_get_thread_num();`
  - `omp_get_thread_schedule();` `omp_set_thread_schedule();`
  - `omp_init_lock();` `omp_set_lock();` `omp_unset_lock();` ...
  - ...

- Environment variables
  - `OMP_SCHEDULE`, `OMP_NUM_THREADS`
  - `OMP_DYNAMIC`, `OMP_NESTED`
  - `OMP_STACKSIZE`, `OMP_THREAD_LIMIT`
  - ...

21
OpenMP Programming Practice

- OpenMP
  - Start with a *parallelizable* algorithm
  - Implement serially, *mostly ignoring*
    - Data races
    - Synchronization
    - Threading syntax
  - Test & Debug
  - Annotation with *directives* for parallelization & synchronization
  - Test & Debug

- Ideal way
  - Start with some algorithm
  - Implement serially, *ignoring*
    - Data races
    - Synchronization
    - Threading syntax
  - Test & Debug
  - *Auto-magically* parallelize
Summary

- **OpenMP is:**
  - An API that may be used to explicitly direct **multi-threaded, shared memory parallelism**
  - Portable
    - C/C++ and Fortran support
    - Implemented on most Unix variants and Windows
  - Standardized
    - Major computer HW and SW vendors jointly defines (OpenMP.org)

- **OpenMP does NOT:**
  - Support distributed memory systems (**but Cluster OpenMP does**)
  - Automatically parallelize
  - Have data distribution controls
  - Guarantee efficiency, freedom from data races, …