Concurrency Parallelism and Distributed Systems (CPDS)

Dependencies: task ordering and data sharing constraints

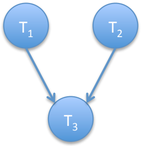
Eduard Ayguadé, Josep-Ramon Herrero and Daniel Jiménez
({eduard,josepr,djimenez}@ac.upc.edu)

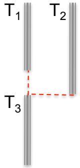
Computer Architecture Department
Universitat Politècnica de Catalunya

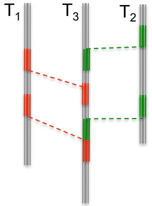
2013/14-Spring

Dependencies

- Constraints in the parallel execution of tasks
  - Task ordering constraints: they force the execution of (groups of) tasks in a required order
  - Data sharing constraints: they force the access to data to fulfil certain properties (write-after-read, exclusive, commutative, ...)







Dependencies (cont.)

- If no constraints are defined during the parallel execution of the tasks, the algorithm is called "embarrassingly" parallel. There are still challenges in this case:
  - Load balancing: how to distribute the work to perform so the load is evenly balanced among tasks (e.g. pi computation vs. Mandelbrot set)
  - Data locality: how to assign work to tasks so that data resides in the memory hierarchy levels close to the processor

Task ordering constraints

- Control flow constraints: the creation of a task depends on the outcome (decision) of one or more previous tasks
- Data flow constraints: the execution of a task can not start until one or more previous tasks have computed some data

Task synchronization

- Task ordering constraints are easily imposed by sequentially composing tasks and/or using global synchronizations.
- In OpenMP (summary):
  - thread barriers (`#pragma omp barrier`): wait for all threads to finish previous work
  - task barriers (`#pragma omp taskwait`): wait for direct task children's work to be done

Fibonnaci series example

Sequential code

```
long fib(long n)
{
    if (n < 2) return n;
    return(fib(n-1) + fib(n-2));
}
void main (int argc, char *argv[])
{
      n = atoi(argv[1]);
      res = fib(n);
      printf("Fibonacci for %d is %d", n, res);
}
```

- The invocations of `fib` for `n-1` and `n-2` can be a task
- We need to guarantee that both instances of `fib` finish before returning the result

Fibonnaci series example (cont.)

OpenMP code using `task` and `taskwait`, with cut-off

```
long fib_parallel(long n, int d)
{
    long x, y;
    if (n < 2) return n;
    if (d<CUTOFF) {
    #pragma omp task shared(x) // firstprivate(n) by default
        x = fib_parallel(n-1, d+1);
    #pragma omp task shared(y) // firstprivate(n) by default
        y = fib_parallel(n-2, d+1);
    #pragma omp taskwait
    } else {
        x = fib_parallel(n-1, d);
        y = fib_parallel(n-2, d);
    }
    return (x+y);
}
void main (int argc, char *argv[])
{
      n = atoi(argv[1]);
#pragma omp parallel
#pragma omp single
      res = fib_parallel(n,0);
      printf("Fibonacci for %d is %d", n, res);
}
```

Task synchronization

- Other more sophisticated mechanisms can be implemented at user level or inside the runtime system to force their execution in the appropriate (dataflow, task generation, ...) order
- Example:
  - Function `foo(i, j)` processes *block(i, j)*
  - Wave-front execution: the execution of `foo(i,j)` depends on `foo(i-1, j)` and `foo(i, j-1)`
  - Processor i computes all blocks *block(i,*)*

```
  for (i=0; i<n i++) {
       for (j=0; j<n;j++) {
          foo(i,j);
        }
   }
```

Task synchronization

- Example (cont.):

```
int blocksfinished[NUMTHREADS];
#pragma omp parallel private(j) num_threads(NUMTHREADS)
{
#pragma omp for
   for (i=0; i<NUMTHREADS; i++) {
        for (j=0; j<n;j++) {
           if (i > 0) {
                while (blocksfinished[i-1]<=j) {
                        #pragma omp flush
                }
           }
           foo(i,j);
           blocksfinished[i]++;
           #pragma omp flush
         }
    }
}
```

  - Pragma `flush` used to enforce memory consistency
  - How should `blocksfinished` be initialized?

Task dependencies in OpenMP v4.0

- The latest release of OpenMP extends the `task` construct to allow the definition of dependencies between sibling tasks (i.e. from the same father)

```
#pragma omp task [depend (in : var_list)]
                 [depend (out : var_list)]
                 [depend (inout : var_list)]
```

  Task dependencies are derived from the dependence type (`in`, `out` or `inout`) and its items in `var_list`. This list may include array sections

Task dependencies in OpenMP v4.0

- The `in` dependence-type: the generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an `out` or `inout` dependence-type list
- The `out` and `inout` dependence-types: the generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an `in`, `out`, or `inout` dependence-type list.

Task dependencies in OpenMP v4.0

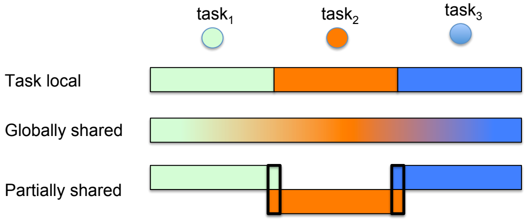
- Example: wave-front execution

```
#pragma omp parallel private(j)
{
   for (i=0; i<n i++) {
        for (j=0; j<n;j++) {
           #pragma omp task depend(in : block[i-1][j], block[i][j-1])
                            depend(out: block[i][j])
           foo(i,j);
         }
    }
}
```

- These new extensions are not yet available in current compilers (specification released July 2013), only in prototype implementations (OmpSs @ BSC/UPC, which includes additional features not in the standard)

Data sharing constraints: patterns

- Task-local data: variables (or chunks of them) only used by single task
- Globally shared data: variables (or chunks of them) not associated with any particular task
- Partially shared data: variables (or chunks of them) shared among smaller groups of tasks (e.g. boundary elements or halos as in Gauss-Seidel)



Task interactions

- Task interactions are needed to guarantee proper access to shared data, without adding too much overhead
  - With shared memory architectures, all processors have access to all data, but must use synchronization to prevent 'race conditions'
  - With distributed memory architectures, each processor has its own data, so race conditions are not possible, but must use communication to (in effect) share data (next chapter)
- Basic approach: first identify what data is shared, second figure out how it is accessed, and finally add the appropriate interactions to guarantee correctness

Task interactions: mutual exclusion

- Critical section: sequence of statements in a task that may conflict with a sequence of statements in another task, creating a possible data race
- Two sequences of statements conflict if both access the same data and at least one of them modifies the data
- Mutual exclusion: mechanism to ensure that only one task at a time executes the code within the critical section
- Locking anomalies:
  - Deadlock (correctness)
  - Contention (performance): minimize the number of mutual exclusions and the size of the critical section protected

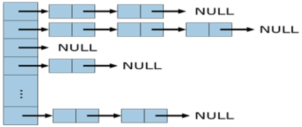
Task interactions: mutual exclusion

In OpenMP two alternatives for mutual exclusion: critical and low-level synchronization functions

- `critical (name)` pragma: if a thread arrives at the top of the critical-section block while another thread is processing the block, it waits until the prior thread exits
  - A critical section in OpenMP implies a memory `flush` on entry to and on exit from it (memory consistency)
  - `name` is an identifier that can be used to support disjoint sets of critical sections

Example: hash table

- Inserting elements in a hash table, defined as a collection of linked lists



```
for (i = 0; i < elements; i++) {
    index = hash(element[i]);
    insert_element (element[i], index);
}
```

- Updates to the list in any particular slot must be protected to prevent a race condition

```
#pragma omp parallel for private (index)
for (i = 0; i < elements; i++) {
    index = hash(element[i]);
    #pragma omp critical
    insert_element (element[i], index);
}
```

Task interactions: mutual exclusion

- Low-level synchronization functions that provide a lock capability

```
void omp_init_lock(omp_lock_t *lock)
void omp_destroy_lock(omp_lock_t *lock)

void omp_set_lock(omp_lock_t *lock)
void omp_unset_lock(omp_lock_t *lock)

int omp_test_lock(omp_lock_t *lock)
```

  - The lock functions guarantee that the lock variable itself is consistently updated between threads, ...
  - ... but do not imply a `flush` of other variables
  - Nested locks are possible

Example: hash table

- Associate a lock variable with each slot in the hash table, protecting the chain of elements in an slot

```
/* hash_lock declared as type omp_lock_t */
omp_lock_t hash_lock[HASH_TABLE_SIZE];

/* locks initialed in function main */
for (i = 0; i < HASH_TABLE_SIZE; i++)
    omp_init_lock(&hash_lock[i]);

#pragma omp parallel for private (index)
for (i = 0; i < elements; i++) {
    index = hash(element[i]);
    omp_set_lock (&hash_lock[index]);
    insert_element (element[i], index);
    #pragma omp flush
    omp_unset_lock (&hash_lock[index]);
}

/* locks destroyed in function main */
for (i = 0; i < HASH_TABLE_SIZE; i++)
    omp_destroy_lock(&hash_lock[i]);
```

- Threads may be inserting elements into the hash table in parallel, as long as these elements hash to different slots

Reducing task interactions

- Group tasks that have strong data interactions. This also has an effect in increasing task granularity
- Replicate computations into local structures to avoid data sharing
- Separable dependencies (reductions): the dependencies between tasks can be managed by replicating key data structures and then accumulating results into these local structures. The tasks then execute according to the embarrassingly parallel pattern and the local replicated data structures are combined into the final global result

Concurrency Parallelism and Distributed Systems (CPDS)

Dependencies: task ordering and data sharing constraints

Eduard Ayguadé, Josep-Ramon Herrero and Daniel Jiménez
({eduard,josepr,djimenez}@ac.upc.edu)

Computer Architecture Department
Universitat Politècnica de Catalunya

2013/14-Spring