Parallelism (PAR)

Parallel programming principles: Task decomposition

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

Computer Architecture Department
Universitat Politècnica de Catalunya

2014/15-Fall

Outline

Objective: problem decomposition

- From a specification that solves the original problem, find a decomposition of the problem
  - Identify pieces of work (tasks) that can be done concurrently
  - Identify data structures (input, output and/or intermediate) or parts of them that can be managed concurrently

  ... ensuring that the same result is produced
  - Identify dependencies that impose ordering constraints (synchronizations) and data sharing

Objective: problem decomposition

- Having in mind two productivity goals:
  - Performance: maximize concurrency and reduce overheads (maximize potential speedup)
  - Programmability: readability and portability, target architecture independency
- Two usual approaches to problem decomposition
  - Task decomposition (Chapter 4)
  - Data decomposition (Chapter 5)
  - Data-flow decomposition (not covered in this course)

  Either way, you'll look at both tasks and data; difference is in which you look at first, and then the other follows

Task decomposition

- A task is a logically related sequence of operations
  - Corresponds to some logical part of the program
- Two common decompositions
  - Function calls
  - Iterations in a repetitive construct (for, while, ...)
- Task size: granularity
- Task ordering and data sharing constraints to guarantee dependences

Data decomposition

- (Large) Data structures partitioned into parts, usually following their logical organization
- Two common decompositions
  - Array data structures (e.g. rows, columns, blocks, ...)
  - Recursive data structures (e.g. subtrees in a tree, ...)
- The size of data partitions defines the granularity of tasks
- Owner computes rule is used to identify tasks
  - Output data decomposition: output is computed by the process to which the output data is assigned
  - Input data decomposition: all computations that use the input data are performed by the process to which the input is assigned

Data-flow decomposition

- Chains of producer-consumer stages (sequential)
- Data is flowing through this sequence of stages
  - Assembly line is an analogy
  - Instruction pipeline in CPUs
  - Pipes in UNIX:
    $cat\ foobar.c \mid grep\ bar \mid wc$
- Concurrency due to data streaming

Outline

Guidelines for task decomposition

- Efficiency
  - Tasks should have enough work to amortize the cost of creating and managing them
  - Tasks should be sufficiently independent so that managing dependencies doesn't become the bottleneck
- Flexibility in the number and size of tasks generated
  - Task decomposition should not be tied to a specific architecture
  - Fixed tasks vs. parameterized tasks
- Simplicity
  - The code has to remain readable and easy to understand and debug

Task creation in OpenMP (summary)

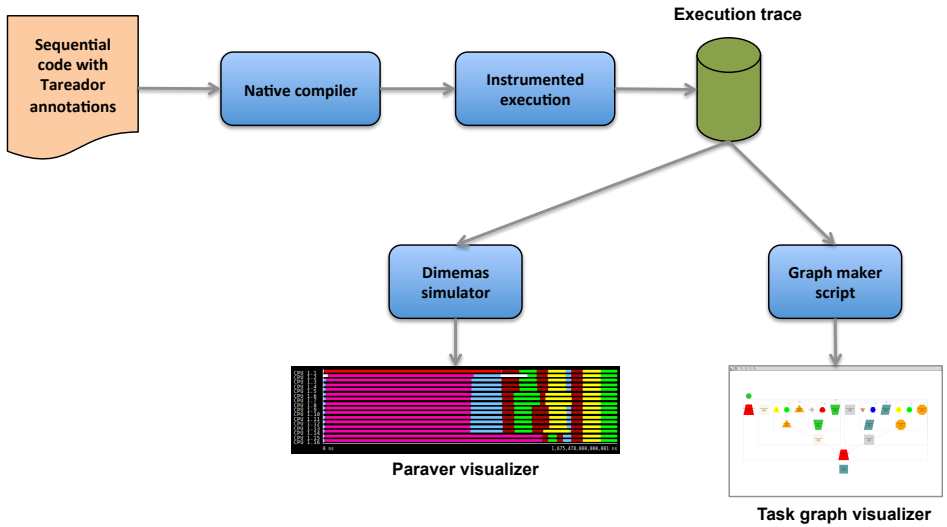
- `#pragma omp parallel`: One **implicit** task is created for each thread in the team (and immediately executed)
  - `#pragma omp for` worksharing: iterations in a loop are distributed among the implicit tasks in the `parallel` region
- `#pragma omp task`: One **explicit** task is created, packaging code and data for (possible) deferred execution
  - Tasks can be nested

Identifying tasks in your sequential program (patterns)

- Linear (or iterative) task generation
  - Tasks found in iterative constructs, such as loops
  - Examples: Pi computation and Mandelbrot set in lab sessions, vector and matrix operations, ...
- Recursive task generation
  - Tasks found when doing a recursive problem decomposition or in divide–and–conquer patterns
  - Two possible task parallelization strategies: Leaf and Tree
  - Examples: multisort and sudoku in lab sessions, Fibonacci and attacking queens problems, ...

Identifying tasks in your sequential program

- In this course we use `Tareador` tool
  - API to define tasks in a program
  - Instrumented execution of annotated sequential code
    - Visualization of task graph, with dynamic task instances and dependences among them
    - Estimation of potential parallelism and speed–up on a number of processors



A very simple example

Parallelize the following sequential code that performs the addition of two vectors:

```
void vector_add(int *A, int *B, int *C, int n) {
   for (int i=0; i< n; i++) C[i] = A[i] + B[i];
}

void main() {
   ....
   vector_add(a, b, c, N);
   ...
}
```

using a) a **linear (iterative) task decomposition** and b) a **recursive divide–and–conquer task decomposition** (for this one you will have to change the original sequential code to make it recursive).

A very simple example (cont.)

Linear (iterative) task decomposition (version 1):

```
void vector_add(int *A, int *B, int *C, int n) {
   for (int i=0; i< n; i++)
       #pragma omp task
       C[i] = A[i] + B[i];
}

void main() {
   ....
   #pragma omp parallel
   #pragma imp single
   vector_add(a, b, c, N);
   ...
}
```

Each explicit task executes a single iteration of the `i` loop. Large task creation overhead, very fine granularity!

A very simple example (cont.)

Linear (iterative) task decomposition (version 2):

```
void vector_add(int *A, int *B, int *C, int n) {
   #pragma omp parallel for schedule(static)
   for (int i=0; i< n; i++) C[i] = A[i] + B[i];
}

void main() {
   ....
   vector_add(a, b, c, N);
   ...
}
```

Each implicit task executes a chunk of contiguous iterations of the `i` loop.

A very simple example (cont.)

Recursive implementation for the previous computation:

```
void vector_add(int *A, int *B, int *C, int n) {
   for (int i=0; i< n; i++) C[i] = A[i] + B[i];
}

#define MIN_SIZE 32

void rec_vector_add(int *A, int *B, int *C, int n) {
   if (n>MIN_SIZE) {
       int n2 = n / 2;
       rec_vector_add(A, B, C, n2);
       rec_vector_add(A+n2, B+n2, C+n2, n-n2);
   }
   else vector_add(A, B, C, n);
}
void main() {
   ....
   rec_vector_add(a, b, c, N);
   ...
}
```

Two possible task parallelizations: leaf and tree

A very simple example (cont.)

Leaf parallelization: a task corresponds with the invocation of `vector_add` once the *sequential* recursive divide–and–conquer stops:

```
void vector_add(int *A, int *B, int *C, int n) {
   for (int i=0; i< n; i++) C[i] = A[i] + B[i];
}

void rec_vector_add(int *A, int *B, int *C, int n) {
   if (n>MIN_SIZE) {
       int n2 = n / 2;
       rec_vector_add(A, B, C, n2);
       rec_vector_add(A+n2, B+n2, C+n2, n-n2);
   }
   else
       #pragma omp task
       vector_add(A, B, C, n);
}

void main() {
   ....
   #pragma omp parallel
   #pragma omp single
   rec_vector_add(a, b, c, N);
   ...
}
```

A very simple example (cont.)

Tree parallelization: a task corresponds with each invocation of `rec_vector_add` during the *parallel* recursive divide–and–conquer execution:

```
void vector_add(int *A, int *B, int *C, int n) {
   for (int i=0; i< n; i++) C[i] = A[i] + B[i];
}

void rec_vector_add(int *A, int *B, int *C, int n) {
   if (n>MIN_SIZE) {
       int n2 = n / 2;
       #pragma omp task
       rec_vector_add(A, B, C, n2);
       #pragma omp task
       rec_vector_add(A+n2, B+n2, C+n2, n-n2);
   }
   else vector_add(A, B, C, n);
}

void main() {
   ....
   #pragma omp parallel
   #pragma omp single
   rec_vector_add(a, b, c, N);
   ...
}
```

A very simple example (cont.)

Assuming $N = 1024$. For each parallelization strategy (Leaf and Tree):

- What is the maximum number of tasks that can be currently active (waiting for execution or running) on the system?
- What is the maximum number of tasks that can be concurrently running on the system?
- How to control the task creation and synchronization overheads?

A very simple example (cont.)

Assuming $N = 1024$. For each parallelization strategy (Leaf and Tree):

- What is the maximum number of tasks that can be currently active (waiting for execution or running) on the system?
- What is the maximum number of tasks that can be concurrently running on the system?
- How to control the task creation and synchronization overheads?

A very simple example with cut-off (Tree)

Tree parallelization with cut-off: the parallel recursive calls are stopped after...

- certain number of recursive calls (static control)
- the size of the vector is too small (static control)
- the number of active tasks is too much (dynamic control)

A very simple example with cut-off (cont.)

```
...
#define CUTOFF 3
void rec_vector_add(int *A, int *B, int *C, int n, int depth) {
   if (n>MIN_SIZE) {
       int n2 = n / 2;
       if (depth < CUTOFF) {
          #pragma omp task
          rec_vector_add(A, B, C, n2, depth+1);
          #pragma omp task
          rec_vector_add(A+n2, B+n2, C+n2, n-n2, depth+1);
       }
       else {
          rec_vector_add(A, B, C, n2, depth);
          rec_vector_add(A+n2, B+n2, C+n2, n-n2, depth);
       }
   }
   else vector_add(A, B, C, n);
}
void main() {
   ....
   #pragma omp parallel
   #pragma omp single
   rec_vector_add(a, b, c, N, 0);
   ...
}
```

A very simple example (cont.)

Assuming $N = 1024$. For the new version:

- What is the maximum number of tasks that can be currently active (waiting for execution or running) on the system?
- What is the maximum number of tasks that can be concurrently running on the system?

`if` clause for `task`: immediate task execution

- If the expression of an `if` clause evaluates to *false*
  - The encountering task is suspended
  - The new task is *executed immediately*
    - with its own data environment
    - different task with respect to synchronization
  - The parent task resumes when the task finishes
  - Allows implementations to *optimize* task creation

`final` clause for `task`: immediate task execution (nested)

- If the expression of a `final` clause evaluates to *true*
  - The generated task and all of its child tasks will be final
  - The execution of a final task is sequentially **included** in the generating task (executed immediately)
- When a `mergeable` clause is present on a task construct, and the generated task is an **included** task, the implementation may generate a merged task instead (i.e. no task and context creation for it).

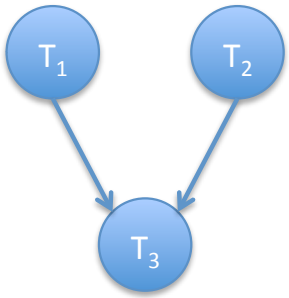
Very simple example with cut-off (rewritten)

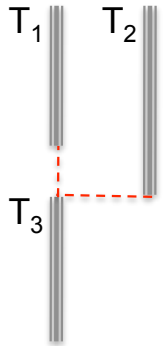
```
...
#define CUTOFF 3
void rec_vector_add(int *A, int *B, int *C, int n, int depth) {
   if (n>MIN_SIZE) {
       int n2 = n / 2;
       #pragma omp task final(depth >= CUTOFF) mergeable
       rec_vector_add(A, B, C, n2, depth+1);
       #pragma omp task final(depth >= CUTOFF) mergeable
       rec_vector_add(A+n2, B+n2, C+n2, n-n2, depth+1);
   }
   else vector_add(A, B, C, n);
}
void main() {
   ....
   #pragma omp parallel
   #pragma omp single
   rec_vector_add(a, b, c, N, 0);
   ...
}
```

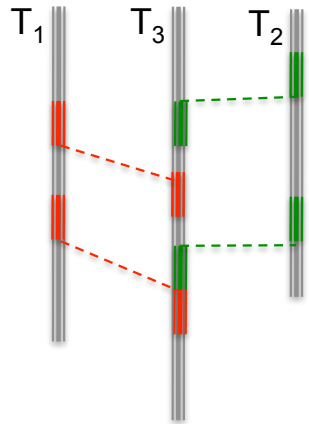
Outline

Dependences

- 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, ...)







Dependences (cont.)

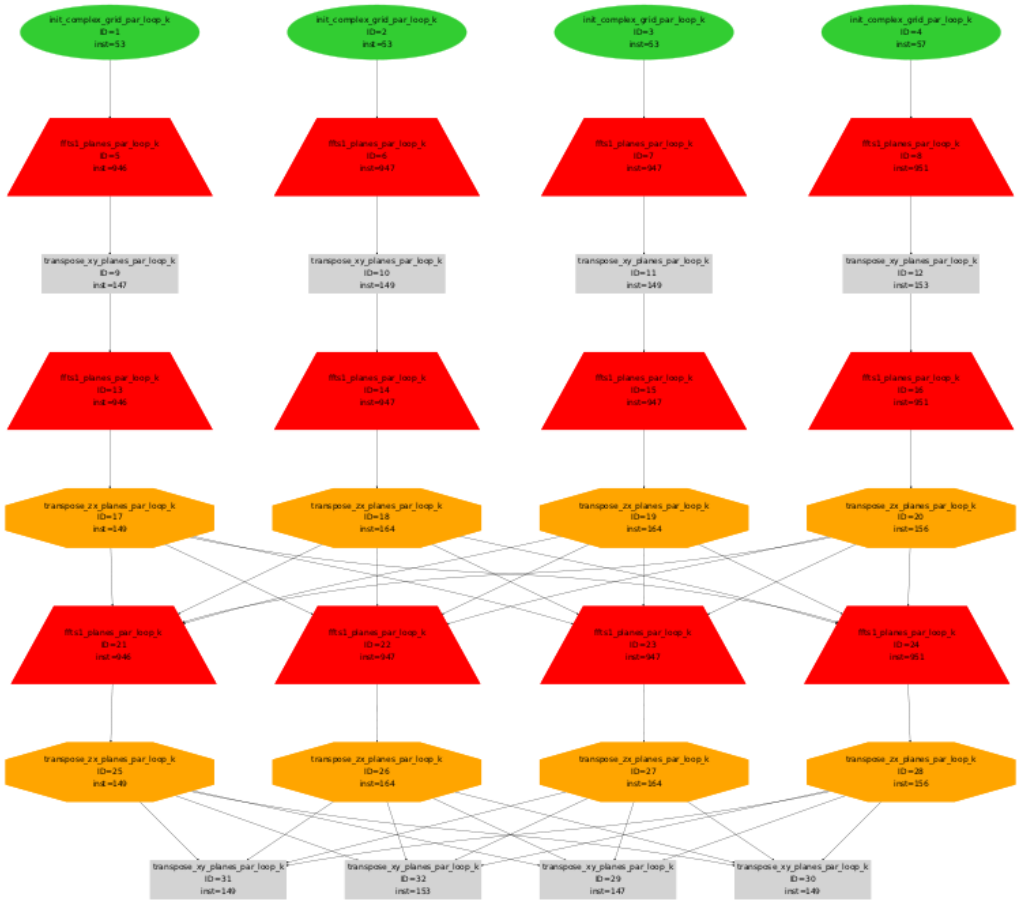
- If no constraints are defined during the parallel execution of the tasks, the algorithm is called "embarrassingly" parallel. There are still challenges in this case:
  - Load balancing: how to distribute the work to perform so the load is evenly balanced among tasks (e.g. pi computation vs. Mandelbrot 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

Data flow constraints: `fft3d` in laboratory session

Tasks: initialization: green; fft: red; transpose: orange and grey



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: wait for the termination of tasks created

- Exercise: In the previous `fft3d`, if each OpenMP thread is assigned the execution of a sequence initialize–fft–transpose–fft, transpose–fft–transpose–transpose, how and when would you synchronize threads to guarantee dependences?

Task synchronization

There are two types of task barriers:

- `taskwait`: Suspends the current task waiting on the completion of **child tasks** of the current task. The taskwait construct is a stand-alone directive.
- `taskgroup`: Suspends the current task at the end of structured block waiting on completion of **child tasks** of the current task **and their descendent** tasks.

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);
}
```

taskwait vs. taskgroup

```
#pragma omp task {}       // T1
#pragma omp task          // T2
{
    #pragma omp task {}   // T3
}
#pragma omp task {}       // T4

#pragma omp taskwait
// Only T1, T2 and T4 are guaranteed to have finished at this point
```

```
#pragma omp task {}           // T1
#pragma omp taskgroup
{
    #pragma omp task          // T2
    {
        #pragma omp task {}   // T3
    }
    #pragma omp task {}       // T4
}
// Only T2, T3 and T4 are guaranteed to have finished at this point
```

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 dependences in OpenMP v4.0

- The latest release of OpenMP extends the `task` construct to allow the definition of dependences 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 dependences are derived from the dependence type (`in`, `out` or `inout`) and its items in `var_list`. This list may include array sections

Task dependences 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.

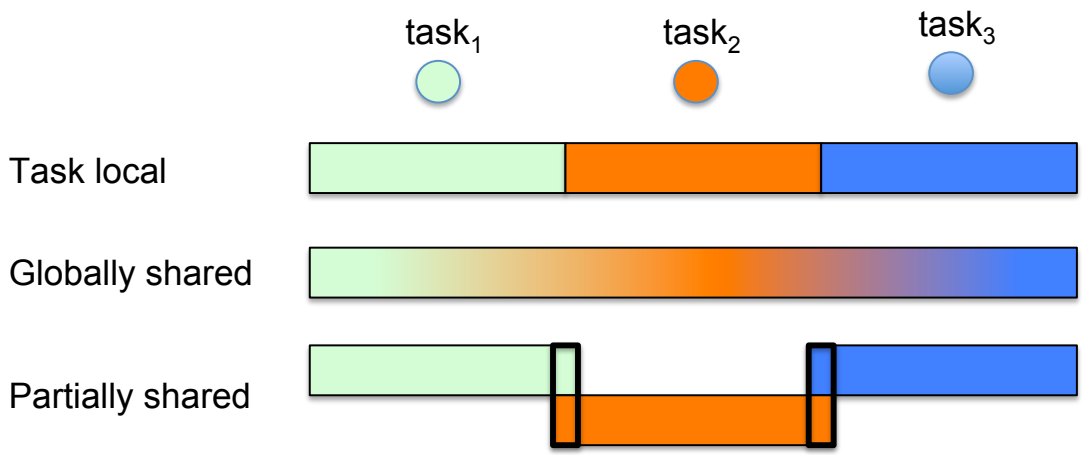
Task dependences in OpenMP v4.0

- Example: wave-front execution

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

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

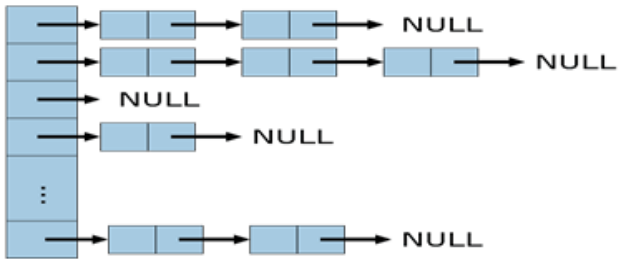
Task interactions: mutual exclusion

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

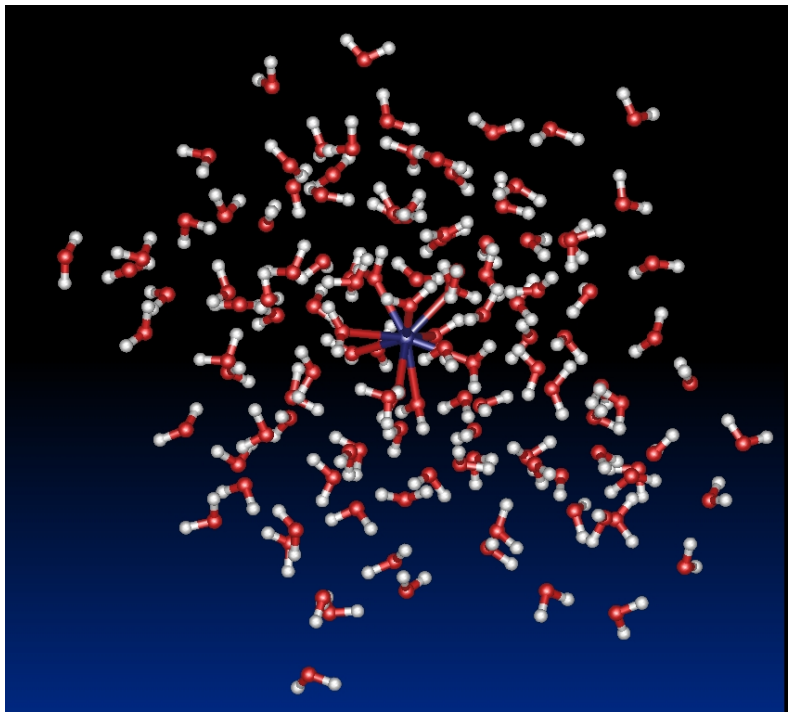
Reducing task interactions

- 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

Outline

Example: molecular dynamics

- Simulate motion in large molecular system. Used for example to understand drug-protein interactions



Example: molecular dynamics (cont.)

- Forces
  - Bonded forces within a molecule (vibration and rotation of chemical bonds between atoms, simple to compute)
  - Non-bonded (long-range) forces between atoms
- Naive algorithm has n^2 interactions to compute non–bonded forces: not feasible
  - Use cutoff method: only consider forces from neighbors that are 'close enough'
  - Sequential code optimized to reduce the number of floating point operations
    - $f_{i,j} = -f_{j,i}$

Example: molecular dynamics (cont.)

Sequential code:

```
Int const N // number of atoms

Array of Real :: atoms (3, N) // 3D coordinates
Array of Real :: forces (3, N) // force in each dimension
// Other data structures for velocity, acceleration and other physical properties
Array of List :: neighbors(N) // atoms in cutoff volume

loop over time steps {
        vibrational_forces (N, atoms, forces)
        rotational_forces (N, atoms, forces)
        every x iterations {
                    neighbor_list (N, atoms, neighbors)
                    }
        non_bonded_forces (N, atoms, neighbors, forces)
        update_atom_positions (N, atoms, forces)
        physical_properties ( . . . Lots of stuff . . . )
} // end of timestep loop
```

Example: molecular dynamics (cont.)

Key data structures:

- An array of atom coordinates, one element per atom
- An array of lists, one per atom, each defining the neighborhood of atoms considered to be 'close'
- An array of forces on atoms, one element per atom
- Other arrays with physical properties, velocity and acceleration, one element per atom (not shown in previous pseudo-codes)

Example: molecular dynamics (cont.)

Bonded forces (vibrational and rotation) are computed as follows:

```
function vibrational_forces (N, atoms, Forces)
    Int N // number of atoms
    Array of Real :: atoms (3, N) // 3D coordinates
    Array of Real :: Forces (3, N) // force in each dimension
    loop [i] over atoms
             Forces(1,i) = vibrational_force(atoms(1, i))
             Forces(2,i) = vibrational_force(atoms(2, i))
             Forces(3,i) = vibrational_force(atoms(3, i))
    end loop [i]
end function bonded_forces
```

```
function rotational_forces (N, atoms, Forces)
    Int N // number of atoms
    Array of Real :: atoms (3, N) // 3D coordinates
    Array of Real :: Forces (3, N) // force in each dimension
    loop [i] over atoms
             Forces(1,i) += rotational_force(atoms(1, i))
             Forces(2,i) += rotational_force(atoms(2, i))
             Forces(3,i) += rotational_force(atoms(3, i))
    end loop [i]
end function bonded_forces
```

Example: molecular dynamics (cont.)

Non–bonded forces are computed as follows:

```
function non_bonded_forces (N, atoms, neighbors, Forces)
    Int N // number of atoms
    Array of Real :: atoms (3, N) // 3D coordinates
    Array of Real :: forces (3, N) // force in each dimension
    Array of List :: neighbors(N) // atoms in cutoff volume
    real : : forceX, forceY, forceZ
    loop [i] over atoms
        loop [j] over neighbors(i)
            forceX = non_bond_force(atoms(1, i) , atoms(1, j) )
            forceY = non_bond_force(atoms(2, i) , atoms(2, j) )
            forceZ = non_bond_force(atoms(3, i) , atoms(3, j) )
            Forces(1, i) += forceX; Forces(1, j) -= forceX;
            Forces(2, i) += forceY; Forces(2, j) -= forceY;
            Forces(3, i) += forceZ; Forces(3, j) -= forceZ;
        end loop [j]
    end loop [i]
end function non_bonded_forces
```

Example: molecular dynamics (cont.)

Computation of the list of neighbors every certain iterations of the timestep loop (assumption: atoms move slowly)

```
function neighbor_list (N, ID, cutoff, atoms, neighbors)
    Int N // number of atoms
    Real cutoff // radius of sphere defining neighborhood
    Array of Real :: atoms (3, N) // 3D coordinates
    Array of List :: neighbors(N) // atoms in cutoff volume
    real : : dist_squ

    initialize_lists (N, neighbors)
    loop [i] over atoms
        loop [j] over atoms greater than i
            dist_squ = square(atom(1,i) - atom(i, j) ) +
                       square(atom(2, i) - atom(2, j ) ) +
                       square(atom(3, i) - atom(3, j ) )
            if (dist_squ < (cutoff * cutoff))
                       add_to_list (j, neighbors(i))
            end if
        end loop [j]
    end loop [i]
end function neighbor_list
```

Example: molecular dynamics (cont.)

The new position for each atom based on computed forces is found as follows:

```
function update_atom_positions (N, atoms, Forces)
    Int N // number of atoms
    Array of Real :: atoms (3, N) // 3D coordinates
    Array of Real :: Forces (3, N) // force in each dimension
    loop [i] over atoms
             atoms(1, i)+= update_position(atoms(1, i), Forces(1,i))
             atoms(2, i)+= update_position(atoms(2, i), Forces(2,i))
             atoms(3, i)+= update_position(atoms(3, i), Forces(3,i))
    end loop [i]
end function update_atom_positions
```

Example: molecular dynamics (accesses to data structures)

- The atomic coordinates
  - Read-only access by the neighbor list, bonded and non-bonded force functions
  - Read-write access for the position update function
- The force array
  - Read-only access by position update function
  - Accumulate access by bonded and non-bonded functions
- The neighbor list
  - Read-only access by non-bonded force function
  - Generated by the neighbor list function

Molecular dynamics: task decomposition

Granularity 1: Each iteration of the timestep loop

- Sequential!

Granularity 2: Each function invocation inside the timestep loop (coarse-grain tasks)

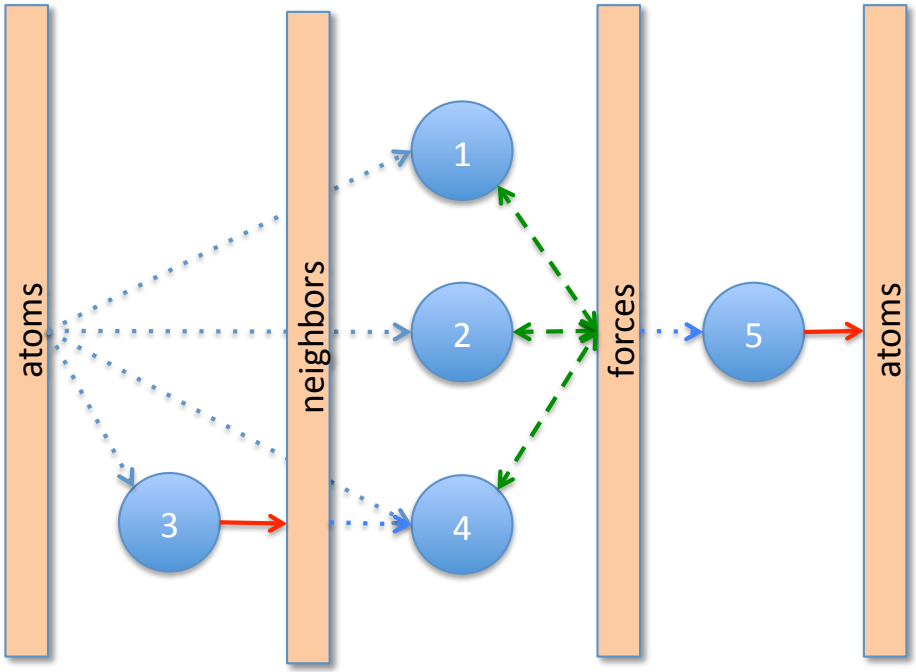
- Small number of (dependent or independent?) tasks ...

Molecular dynamics: data dependencies

- Tasks are dependent: dataflow (dependences) serialize execution of tasks
  - Tasks `neighbors_list` prior to task `non_bonded_forces`
  - Task `update_atom_positions` can not execute until vector `force` has been updated
- Tasks interact, need to protect interaction
  - Tasks computing atom i read and write the same element of `force` array, accumulating partial results
  - Potentially, a task can interact with any other task (i.e. task computing atom i has contribution to its neighbor j)

Molecular dynamics (coarse-grained tasks)

- Each function call is a task



Nodes
1 vibrational_forces
2 rotational_forces
3 list_neighbors
4 non_bonded_forces
5 update_atom_positions

Edges
read only
read-write
accumulate (reduce)

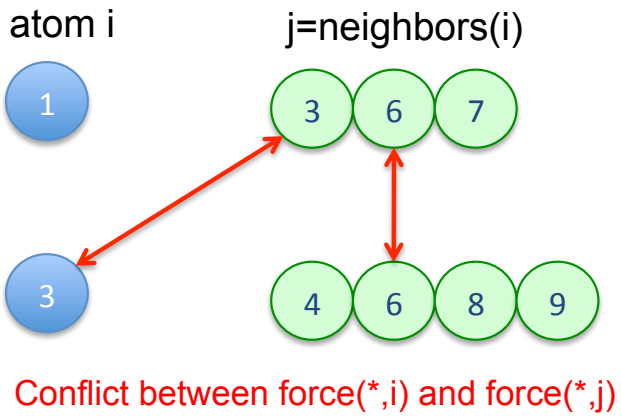
Molecular dynamics: task decomposition (cont.)

Granularity 3: Each iteration on atoms in each function is a task (fine-grain tasks)

- All tasks in `vibrational/rotational_forces` are independent: each one computes a different element in `force`
- All tasks in `update_atom_positions` are independent: each one computes a different element in `atoms`
- All tasks in `neighbors_list` are independent: each one computes a different list in `neighbors`

Molecular dynamics: task decomposition (cont.)

- However, tasks i and j in `non_bonded_forces` may accumulate on the same element in `force`



Note: the granularity of tasks can be increased by simply executing several iterations in each task

Molecular dynamics: task decomposition (cont.)

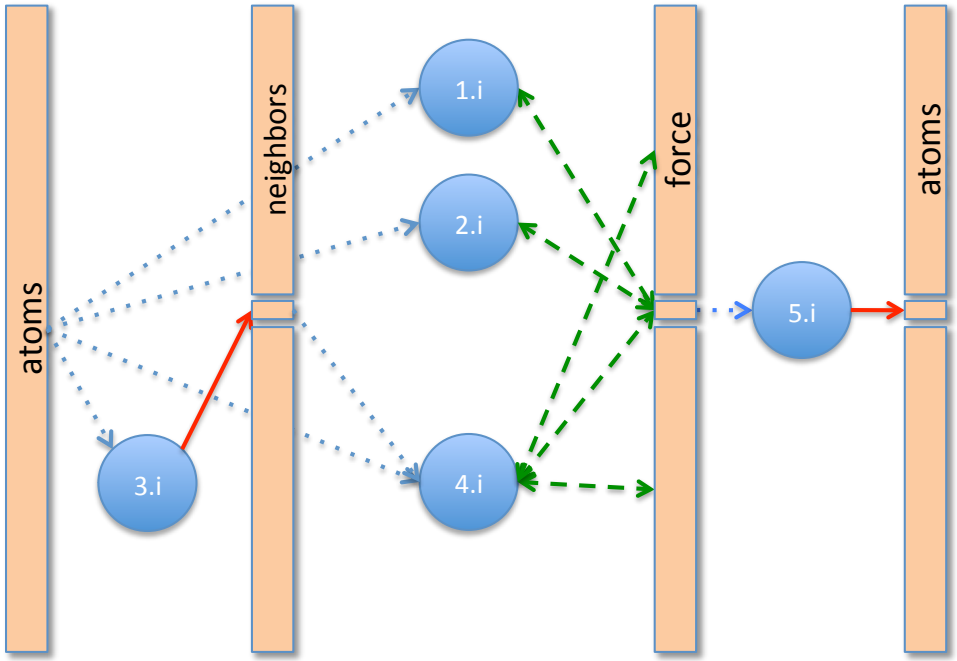
Two observations:

- Potential load balancing problem since the number of neighbors of an atom may vary greatly from one atom to another
- Loop j iterates for all those atoms with index larger than i: reduction in $1/2$ the number of floating point operations during the computation of forces. Is this going to have an impact (negative) later in how tasks interact?[1]

[1] That sequential optimization implies synchronization on the parallel force computation

Example: molecular dynamics (fine-grained tasks)

- Each iteration on atoms in each function is a task



Nodes
1.i iteration *i* vibrational_forces
2.i iteration *i* rotational_forces
3.i iteration *I* list_neighbors
4.i iteration *i* non_bonded_forces
5.i iteration *i* update_atom_positions

Edges
read only
read-write
accumulate (reduce)

Fine-grained molecular dynamics (enforcing dependences)

Option 1:

- Ensure exclusive access to memory (e.g. `atomic` or `critical` in OpenMP) during the read-compute-write sequence in each task in `vibrational_forces`, `rotational_forces` and `non_bonded_forces`
- Global synchronization before starting tasks in `update_atom_positions`
- Idem after the execution of tasks in `update_atom_positions`

Fine-grained molecular dynamics (enforcing dependences)

Option 2: if number of atoms is sufficiently large

- Global synchronization (e.g. `taskwait` in OpenMP) between tasks in `vibrational_forces` and `rotational_forces`
- Idem between tasks in `rotational_forces` and `non_bonded_forces`
- Ensure exclusive access to memory during the read-compute-write sequence in each task in `non_bonded_forces`
- Global synchronization between tasks in `non_bonded_forces` and `update_atom_positions`
- Idem after the execution of tasks in `update_atom_positions`

Fine-grained molecular dynamics (enforcing dependences)

Option 3:

- If the sequential optimization in `non_bonded_forces` to reduce the number of floating-point operations is removed (i.e. $force_{i,j}$ and $force_{j,i}$ are redundantly computed) then the exclusive access to memory is not needed

Option 4:

- The exclusive access is not needed if we restructure the code so that iteration i in `vibrational_forces`, `rotational_forces` and `non_bonded_forces` are fused in a single task. Probably this also improves memory locality.
- Global synchronization before starting and after finishing all tasks in `update_atom_positions` is still necessary

Parallelism (PAR)

Parallel programming principles: Task decomposition

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

Computer Architecture Department
Universitat Politècnica de Catalunya

2014/15-Fall