Parallel Programming

Shared-Memory Programming with OpenMP

(Part II)

Professor Yi-Ping You (游逸平)
Department of Computer Science
http://www.cs.nctu.edu.tw/~ypyou/

Outline

- Task (OpenMP 3.0)
  - `task` Constructs
  - Task Clauses
- OpenMP Memory Model (`flush`)
- `atomic` (OpenMP 3.1)
- `threadprivate` Data
- Case Study: N-Body Problem

Data vs Task Parallelism

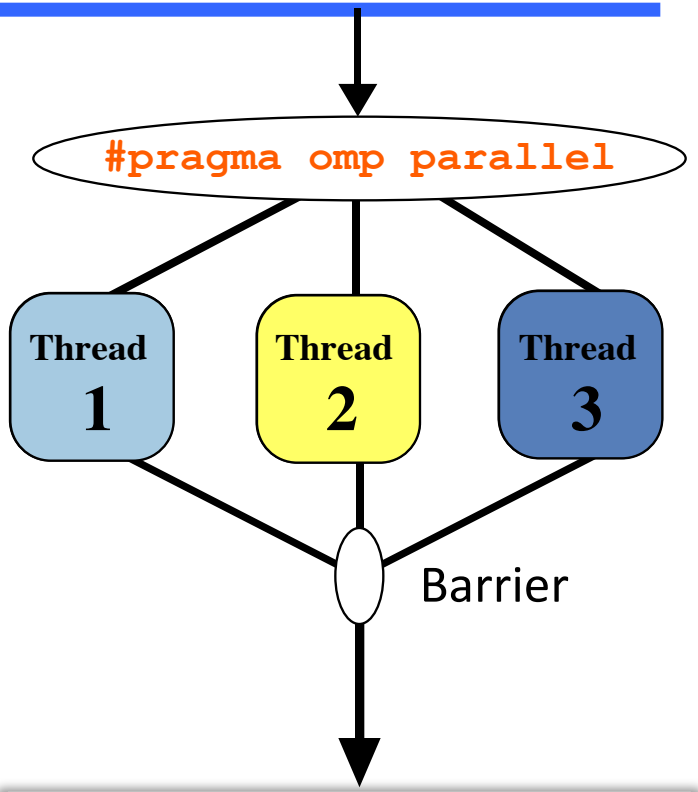
- Data parallelism
  - You have a large amount of data elements and each data element (or possibly a subset of elements) needs to be processed to produce a result
  - When this processing can be done in parallel, we have data parallelism
  - Example:
    - Adding two long arrays of doubles to produce yet another array of doubles
- Task parallelism
  - You have a collection of tasks that need to be completed
  - If these tasks can be performed in parallel you are faced with a task parallel job
  - Examples:
    - Reading the newspaper, drinking coffee, and scratching your back
    - Breathing your lungs, beating of your heart, liver function, controlling the swallowing, etc.

Tasks and OpenMP

- Tasks have been fully integrated into OpenMP
- Key concept: OpenMP always had tasks, we just never called them that
  - Thread encountering `parallel` construct packages up a set of *implicit* tasks, one per thread
  - Team of threads is created
  - Each thread in team is assigned to one of the tasks (and *tied* to it)
  - Barrier holds original master thread until all implicit tasks are finished
- We have simply added a way to create a task explicitly for the team to execute
- Every part of an OpenMP program is part of one task or another!

Parallel Construct – Implicit Task View

- Tasks are created in OpenMP even without an explicit task directive
- Lets look at how tasks are created implicitly for the code snippet below
  - Thread encountering parallel construct packages up a set of implicit tasks
  - Team of threads is created
  - Each thread in team is assigned to one of the tasks (and tied to it)
  - Barrier holds original master thread until all implicit tasks are finished



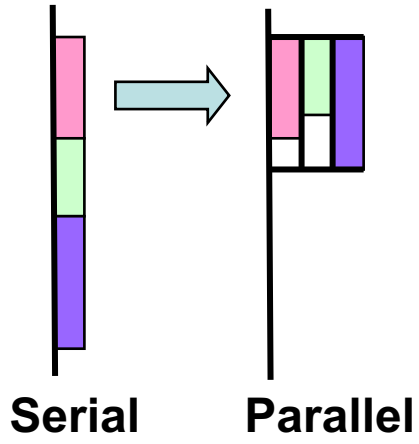

```
#pragma omp parallel
{
   int mydata;
   code...
}
```

OpenMP Tasks (1/2)

- Tasks – Main change for OpenMP 3.0
  - Allows parallelization of irregular problems
    - unbounded loops
    - recursive algorithms
    - producer/consumer
- A task has
  - Code to execute
  - A data environment (it owns its data)
  - An assigned thread that executes the code and uses the data
- Two activities: packaging and execution
  - Each encountering thread packages a new instance of a task (code and data)
  - Some thread in the team executes the task at some later time

OpenMP Tasks (2/2)

- A more flexible way to define units of work
- Tasks are independent units of work
- Tasks are composed of:
  - code to execute
  - data environment (it owns its data)
  - internal control variables (ICV)
- Threads perform the work of each task
- The runtime system decides when tasks are executed
  - Tasks may be deferred
  - Tasks may be executed immediately



Outline

- Task (OpenMP 3.0)
  - `task` Constructs
  - Task Clauses
- OpenMP Memory Model (`flush`)
- `atomic` (OpenMP 3.1)
- `threadprivate` Data
- Case Study: N-Body Problem

`task` Construct

- **`#pragma omp task`** `[clause[[,]clause] ...]`
  structured-block

where clause can be one of:

- `shared (list)`
- `private (list)`
- `firstprivate (list)`
- `default( shared | none )`
- `if (expression)`
- `untied`
- `final (expression)`
- `mergeable`

Discussed later

Example: Parallelizing List Processing

- Given what we've covered about OpenMP, how would you process this loop in Parallel?

```
p = head;
while (p) {
   process(p);
   p = p->next;
}
```

```
for(p=head; p; p=p->next) {
   process(p);
}
```

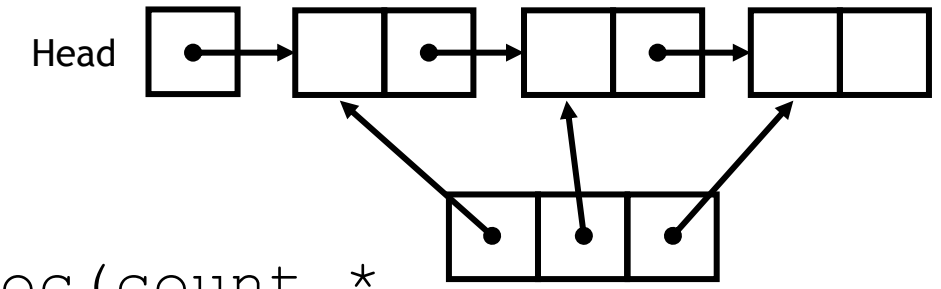
Not a canonical `for` loop

- Remember, the loop worksharing construct only works with loops for which the number of loop iterations can be represented by a closed-form expression at compiler time
  - While loops are not covered

Parallel List Processing with OpenMP 2.5

Count number of items in the linked list

```
while (p != NULL) {
  p = p->next;
  count++;
}
parr = (*node) malloc(count *
       sizeof(struct node));
p = head;
for(i=0; i<count; i++) {
  parr[i] = p;
  p = p->next;
}
#pragma omp parallel for schedule(static,1)
for(i=0; i<count; i++)
  process(parr[i]);
```



Copy pointer to each node into an array

Process nodes in parallel with a for loop

Parallel List Processing Using `task`

```
#pragma omp parallel
{
  #pragma omp single
  {
    node *p = head;
    while (p) {
      #pragma omp task firstprivate(p)
        process(p);
      p = p->next;
    }
  }
}
```

Create a team of threads

One thread executes the single construct

Other threads wait at the implied barrier at the end of the single construct

The "single" thread creates a task with its own value for the pointer `p`

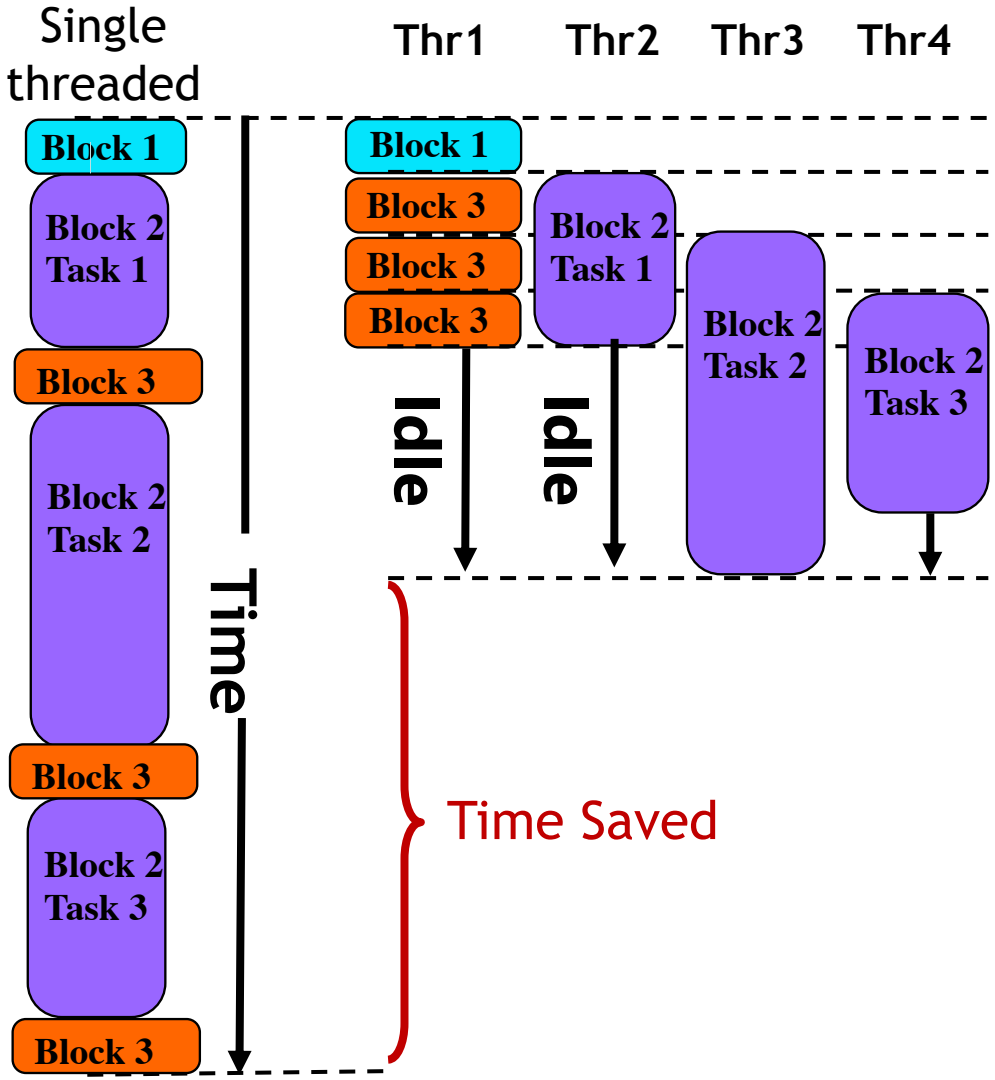
Threads waiting at the barrier execute tasks

Execution moves beyond the barrier once all the tasks are complete

Why are tasks useful?

- Have potential to parallelize irregular patterns and recursive function calls

```
#pragma omp parallel
{
  #pragma omp single
  {
    node *p = head;
    while (p) {
      #pragma omp task \
          firstprivate(p)
        process(p);
      p = p->next;
    }
  }
}
```



When/Where Are Tasks Complete?

- At thread barriers, explicit or implicit
  - Applies to all tasks generated in the current parallel region up to the barrier
  - Matches user expectation
  - `#pragma omp barrier`
- At task barriers
  - Applies only to tasks generated in the current task
    - i.e., wait until all tasks defined in the current task have completed
  - `#pragma omp taskwait`

Tasks Complete at Thread Barriers

```
#pragma omp parallel
{
  #pragma omp task
  foo();
  #pragma omp barrier
  #pragma omp single
  {
    #pragma omp task
    bar();
  }
}
```

Multiple `foo` tasks created here - one for each

All `foo` tasks guaranteed to be completed here

One `bar` task created here

`bar` task guaranteed to be completed here

Tasks Complete at Task Barriers

- `#pragma omp taskwait` specifies a wait on the completion of child tasks generated since the beginning of the current task

```
#pragma omp task          // Task 1
{  ...
   #pragma omp task       // Task 2
   { do_work1(); }
   #pragma omp task       // Task 3
   {   ...
      #pragma omp task // Task 4
      {   do_work2();
         ...
      }
      #pragma omp taskwait
      ...
   }
}
```

Causes Task 3 to wait until the completion of Task 4

It does not wait for the completion of Task 2 or Task 1

Data Scoping with Tasks: Fibonacci Example (1/2)

```
int main() {
  int n;
  long result;
  ...
  #pragma omp parallel num_threads(NUM_THREADS)
  {
    #pragma omp single nowait
    {
      result = fib(n);
    }
  }
  ...
  return 0;
}
```

Private data is made `firstprivate` by default

`n` is private in both task

```
long fib(int n) {
  long x, y;
  if ( n == 0 || n == 1 ) return(n);
#pragma omp task
  x = fib(n-1);
#pragma omp task
  y = fib(n-2);
#pragma omp taskwait

  return x + y;
}
```

`x` and `y` are private

`x` and `y` are undefined outside the task

Data Scoping with Tasks: Fibonacci Example (2/2)

```
int main() {
  int n;
  long result;
  ...
  #pragma omp parallel num_threads(NUM_THREADS)
  {
    #pragma omp single nowait
    {
      result = fib(n);
    }
  }
  ...
  return 0;
}
```

```
long fib(int n) {
  long x, y;
  if ( n == 0 || n == 1 ) return(n);
#pragma omp task shared(x)
  x = fib(n-1);
#pragma omp task shared(y)
  y = fib(n-2);
#pragma omp taskwait

  return x + y;
}
```

Data Scoping with Tasks: List Traversal Example (1/2)

```
List ml; //my_list
Element *e;
#pragma omp parallel
#pragma omp single
{
  for(e=ml->first; e; e=e->next)
    #pragma omp task
      process(e);
}
```

Possible data race!
Shared variable `e` updated by multiple tasks

Data Scoping with Tasks: List Traversal Example (2/2)

```
List ml; //my_list
Element *e;
#pragma omp parallel
#pragma omp single
{
  for(e=ml->first; e; e=e->next)
    #pragma omp task firstprivate(e)
      process(e);
}
```

Good solution: `e` is `firstprivate`

Notes on Scoping Rules

- The rules for how the default data-sharing attributes of variables are implicitly determined and may not always be obvious
- To avoid any surprises, it is recommended that the programmer explicitly scope all variables that are referenced in a task construct using the data sharing attribute clauses, rather than rely on the OpenMP implicit scoping rules

Sections vs Tasks (1/2)

- The difference between tasks and sections is in the time frame that their code would execute
  - Sections are enclosed within the `sections` construct and (unless the `nowait` clause was specified) threads would not leave it until all sections have been executed
  - Tasks are queued and executed whenever possible at the so-called task scheduling points. The run-time is also allowed to move task between threads, even in the mid of their lifetime. That means that one task might start executing in one thread, then at some scheduling point it might be migrated by the runtime to another thread

Sections vs Tasks (2/2)

- Still tasks and sections are in many ways similar
- For example the following two code fragments achieve essentially the same result:

```
// sections
...
#pragma omp sections
{
  #pragma omp section
  foo();
  #pragma omp section
  bar();
}
...
```

```
// tasks
...
#pragma omp single nowait
{
  #pragma omp task
  foo();
  #pragma omp task
  bar();
}
#pragma omp taskwait
...
```

If there was no `single` construct, each task would get created `num_threads` times, which might not be what one wants

Outline

- Task (OpenMP 3.0)
  - `task` Constructs
  - Task Clauses
- OpenMP Memory Model (`flush`)
- `atomic` (OpenMP 3.1)
- `threadprivate` Data
- Case Study: N-Body Problem

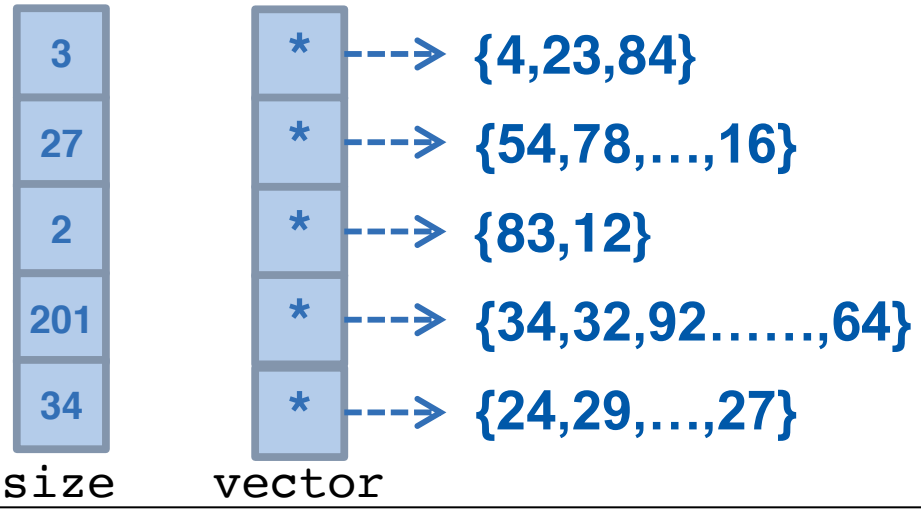
`if` Clause

- `#pragma omp task if (expression)`
- Specifies that the enclosed code section is to be executed in parallel only if the `expression` evaluates to `true`
  - Throttle (cutoff) mechanism to control task creation
  - If expression evaluates to `false`, do not create the task (conceptually)
    - Current task is suspended
    - New task is executed immediately (not deferred) by the encountering thread
      - The data environment is still local to the new task
    - The suspended task is resumed when the generated task is completed

`if` Clause: An Example

- Creating (or not) a task using `if` clause

```
void foo(int *size, int **vector, int N) {
  for (int i=0; i<N; i++) {
  #pragma omp task if ( size[i] > MIN_SIZE )
    compute(vector[i], size[i]);
  }
}
```



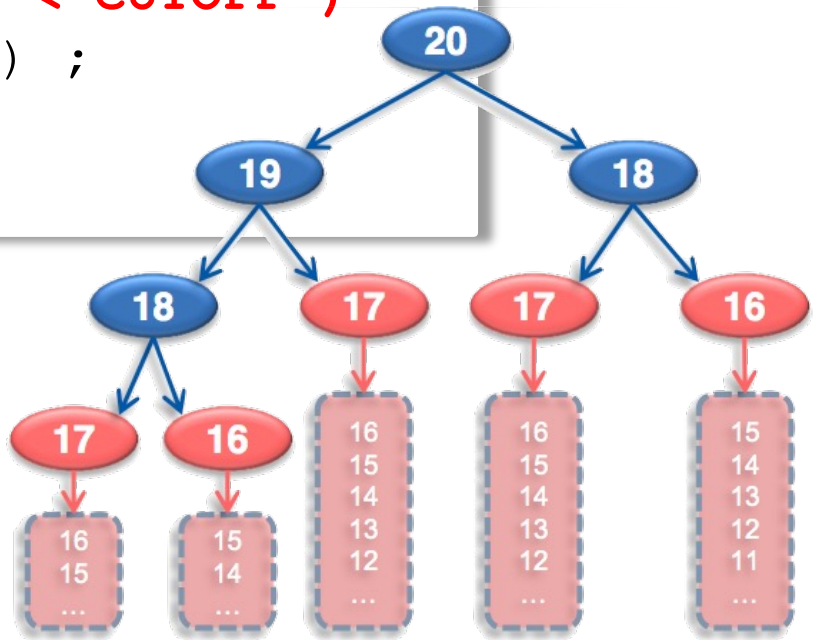
`final` Clause

- `#pragma omp task final (expression)`
- When the `expression` evaluates to `true`, it specifies that the generated task will be a *final* task
  - *final* task
    - A task that forces all of its child tasks to become *final* and *included* tasks
  - *included* task
    - A task for which execution is sequentially included in the generating task region
    - That is, an included task is undeferred and executed immediately by the encountering thread
  - If expression evaluates to `true`, do not create more tasks in the enclosed task context (conceptually)

`final` Clause: An Example

- Using `final` tasks as cutoff mechanism to control granularity

```
void fibonacci ( int n ) {
  if ( n < 2 ) return n ;
  #pragma omp task final ( n < CUTOFF )
  int x = fibonacci ( n - 1 ) ;
  #pragma omp task final ( n < CUTOFF )
  int y = fibonacci ( n - 2 ) ;
  return x + y ;
}
```

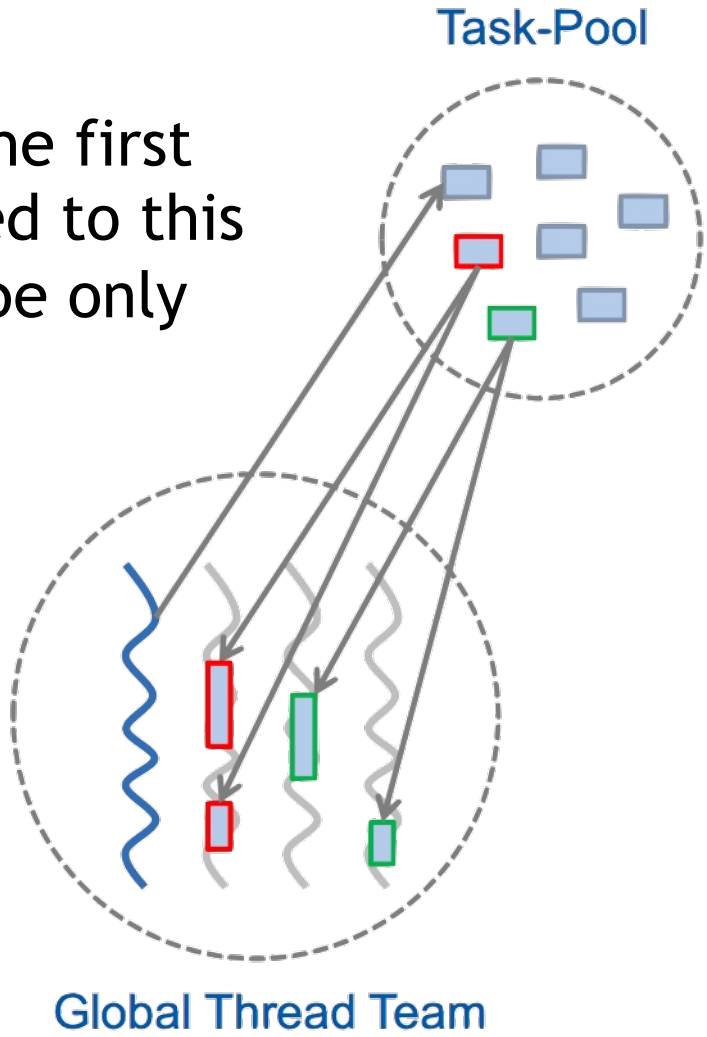


`untied` Clause (1/2)

- `#pragma omp task untied`
- Specifies that the task is never tied to the thread that started its execution. Any thread in the team can resume the task region after a suspension
  - For example, during runtime, the compiler can start the execution of a given task on thread A, break execution, and later resume it on thread B

`untied` Clause (2/2)

- `untied`: Task will not be tied to any thread
- Tied tasks (default)
  - Once the task is executed (for the first time) in one thread it will be tied to this thread, and (if suspended) will be only resumed in this thread
- Untied tasks
  - Can be suspended and resumed in any thread (thread switch)



Task Switching Example

- The thread executing the “for loop”, AKA the generating task, generates many tasks in a short time so...
- The SINGLE generating task will have to suspend for a while when “task pool” fills up
  - Task switching is invoked to start draining the “pool”
  - When “pool” is sufficiently drained – then the single task can being generating more tasks again

```
int exp;  // exp either T or F;
#pragma omp single
{
  for (i=0; i<ONEZILLION; i++) {
    #pragma omp task
    process(item[i]);
  }
}
```

`mergeable` Clause

- `#pragma omp task mergeable`
- Allows the implementation to reuse the environment from another task
- When the generated task is an undeferred task or an included task, it specifies that the implementation may generate a merged task instead

`mergeable` Clause: An Example

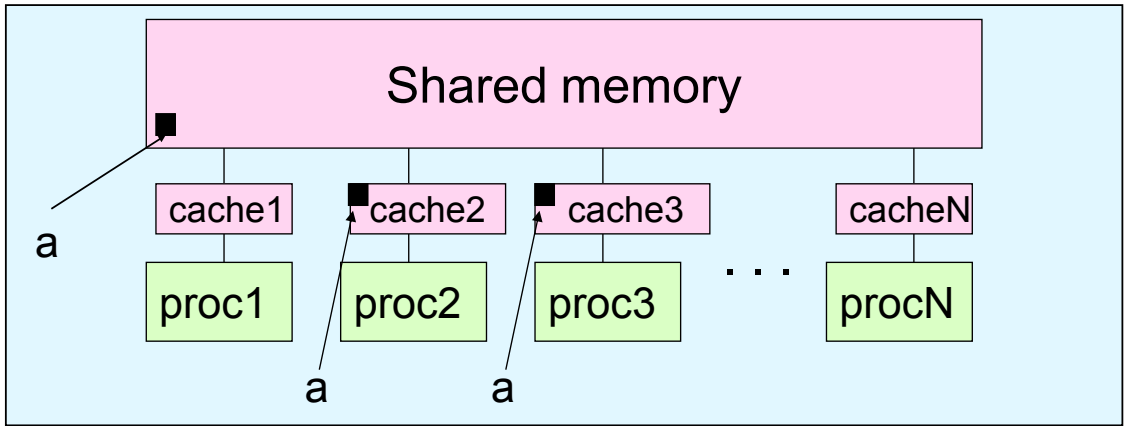
```
void foo() {
  int x = 2;

  #progma omp task mergeable
  {
    x++; // x is by default firstprivate
  }
  #progma omp taskwait
  printf("%d\n", x); // prints 2 or 3
} // Non-deterministic behavior!
```

Outline

- Task (OpenMP 3.0)
  - `task` Constructs
  - Task Clauses
- OpenMP Memory Model (`flush`)
- `atomic` (OpenMP 3.1)
- `threadprivate` Data
- Case Study: N-Body Problem

OpenMP Memory Model

- OpenMP supports a shared memory model
- All threads share an address space ... but what actually see at a given point in time may be complicated:

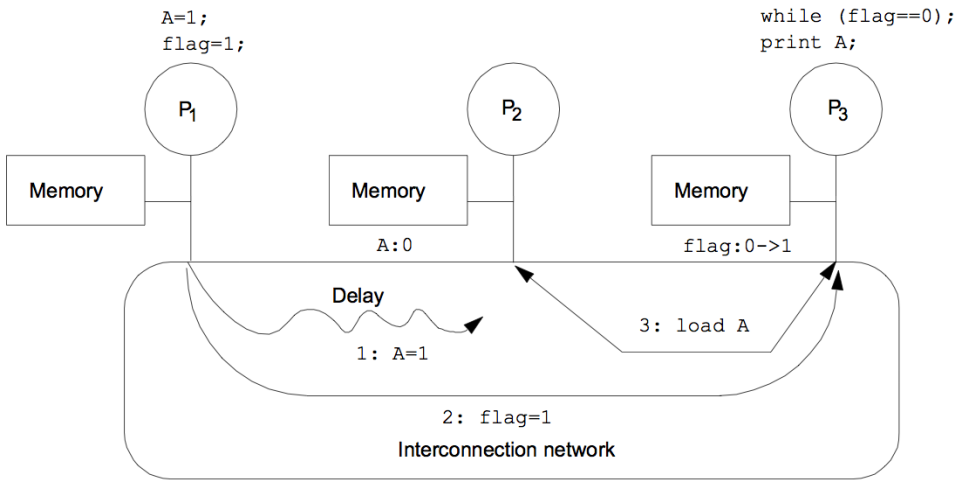


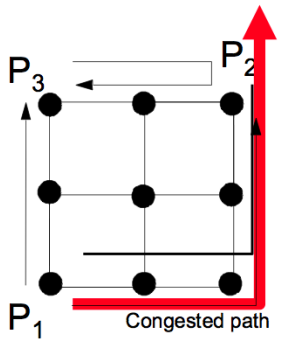
- A memory model is defined in terms of:
  - Coherence: Behavior of the memory system when a single address is accessed by multiple threads
  - Consistency: Orderings of reads, writes, or synchronizations (RWS) with various addresses and by multiple threads
    - The memory consistency model defines when a written value must be seen by a following read instruction made by the other processors

Example: Coherence not Enough

P_1	P_3
/*Assume initial value of A and flag is 0*/	
`A = 1;`	`while (flag == 0);` /*spin idly*/
`flag = 1;`	`print A;`

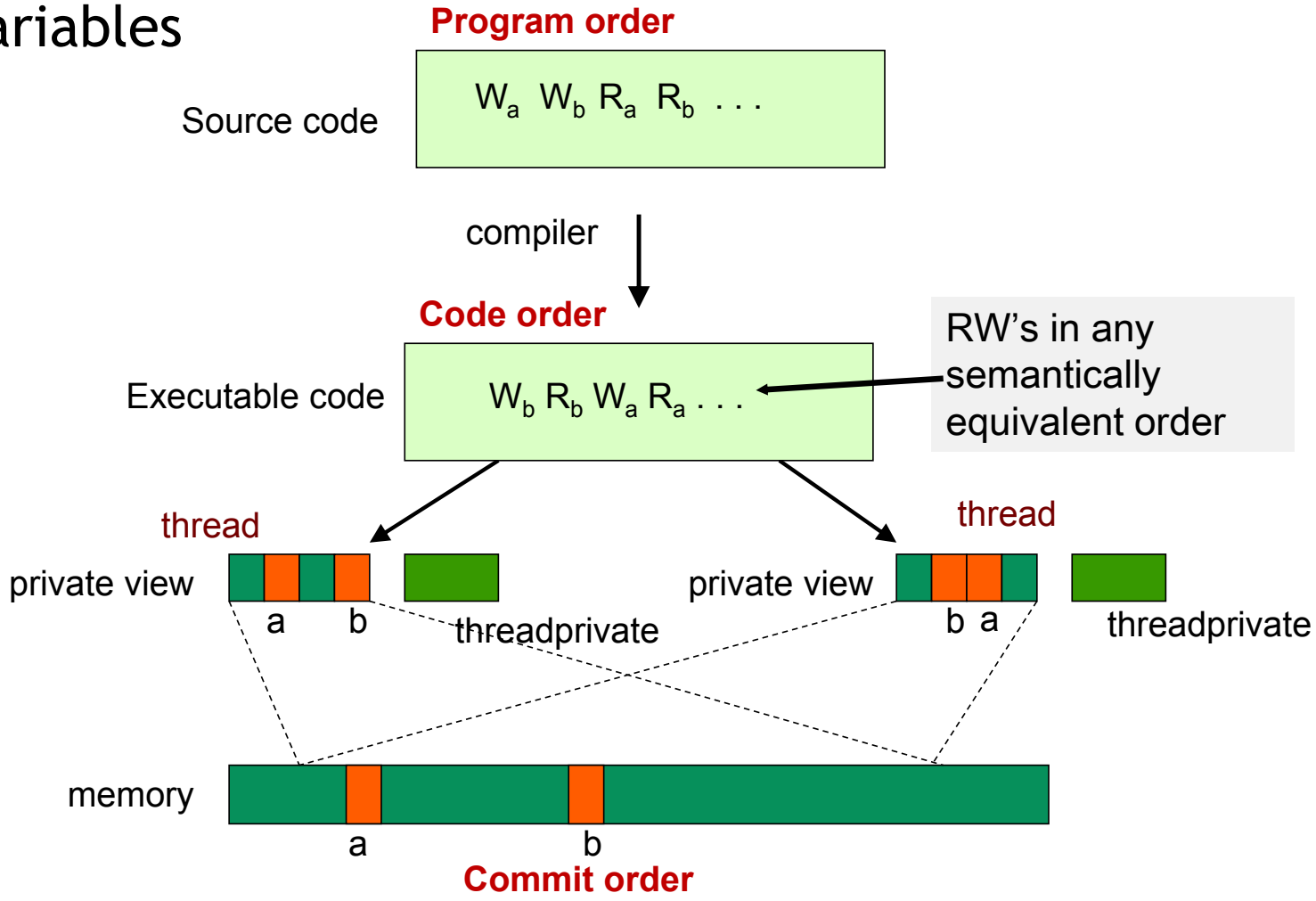
- Expect memory to respect order between accesses to different locations issued by a given process
  - to preserve orders among accesses to same location by different processes
- Intuition not guaranteed by coherence





Reordering Memory Operations

- Optimizations by compilers and hardware execution models (e.g. out-of-order-execution) reorder operations to variables



Sequential Consistency

- In a multi-processor, ops (R, W, S) are sequentially consistent if
  - Commit order == program order in each thread
    - program order == code order == commit order
  - Same overall order seen on all threads
- Problems: Current hardware does not directly support sequential consistency
  - Write buffers break sequential consistency on orders of Writes (W)
  - Size of (R, W) words may be smaller than objects so individual (R,W) ops can overlap (e.g., 64 bit variables on a 32 bit architecture)
  - Synchronization operations (S) to impose sequential consistency add a great deal of overhead

Solution: Relaxed Consistency

- Memory ops must be divided into "data" ops and "synch" ops
  - Data ops (reads & writes) are not ordered w.r.t. each other
  - Data ops **are** ordered w.r.t. synch ops and synch ops are ordered w.r.t. each other
  - Weak consistency guarantees
    - S→W, S→R , R→S, W→S, S→S
- The Synchronization operation relevant to this discussion is `flush`

OpenMP's flush is analogous to a fence in other shared memory API's

`flush` Construct

- `#program omp flush [(list)]`
- Defines a sequence point at which a thread is guaranteed to see a consistent view of memory with respect to the "flush set"
- The flush set is:
  - "all thread visible variables" for a flush construct without an argument list
  - a list of variables when the "`flush(list)`" construct is used
- The action of `flush` is to guarantee that:
  - All R,W ops that overlap the flush set and occur prior to the flush complete before the flush executes
  - All R,W ops that overlap the flush set and occur after the flush don't execute until after the flush
  - Flushes with overlapping flush sets can not be reordered

`flush` Is the Key OpenMP Operation

- Prevents re-ordering of accesses
- Provides a guarantee that memory references are complete
- Provides the mechanism for moving data between threads
- Allows for overlapping computation with communication

Note: the `flush` operation does not actually synchronize different threads. It just ensures that a thread's values are made consistent with main memory and available to other threads.

`flush` Example

- Private view allows hiding memory latency

"`a`" can be committed to memory as soon as here

or as late as here

```
a = . . .;

<other computation>

#pragma omp flush(a)
```

Re-ordering Example

```
a = ...;   //(1)
b = ...;   //(2)
c = ...;   //(3)

#pragma omp flush(c)     //(4)
#pragma omp flush(a,b)   //(5)

. . . a . . . b . . .; //(6)
. . . c . . .;         //(7)
```

- (1) and (2) may not be moved after (5)
- (6) may not be moved before (5)
- (4) and (5) may be interchanged at will

Moving Data between Threads

- To move the value of a shared `var` from thread A to thread B, do the following in exactly this order:
  - Write `var` on thread A
  - Flush `var` on thread A
  - Flush `var` on thread B
  - Read `var` on thread B

Implicit `flush`es

- A flush operation is implied by OpenMP synchronizations, e.g.
  - at entry/exit of `parallel` regions
  - at implicit and explicit barriers
  - at entry/exit of `critical` regions
  - whenever a lock is set or unset
  - ...
  - but not at entry to worksharing regions or entry/exit of `master` regions

Example: Pair Wise Synchronization (Producer-Consumer Sync.)

- OpenMP lacks synchronization constructs that work between pairs of threads
- When this is needed you have to build it yourself
- Pair wise synchronization
  - Use a shared flag variable
  - Reader spins waiting for the new flag value
  - Use flushes to force updates to and from memory

Example: Producer-Consumer Pattern

Thread 0

```
a = foo();
flag = 1;
```

Thread 1

```
while (!flag);
b = a;
```

- This is incorrect code
- The compiler and/or hardware may re-order the reads/writes to `a` and `flag`, or `flag` may be held in a register
- OpenMP has a `#pragma omp flush` directive which specifies an explicit flush operation
  - can be used to make the above example work
  - … but its use is difficult and prone to subtle bugs

Example: Producer-Consumer Problem (1/2)

- Parallelize a producer consumer program
  - One thread produces values that another thread consumes
  - Often used with a stream of produced values to implement "pipeline parallelism"
  - The key is to implement pairwise synchronization between threads

```
int main() {
  double *A, sum, runtime;
  int flag = 0;
  A = (double *)malloc(N*sizeof(double));
  runtime = omp_get_wtime();
  fill_rand(N, A); // Producer: fill an array of data
  sum = Sum_array(N, A); // Consumer: sum the array
  runtime = omp_get_wtime() - runtime;
  printf(" In %lf secs, The sum is %lf\n",runtime,sum);
}
```

Example: Producer-Consumer Problem (2/2)

```
int main() {
  double *A, sum, runtime;
  int numthreads, flag = 0;
  A = (double *)malloc(N*sizeof(double));
  #pragma omp parallel sections
  {
    #pragma omp section
    {
      fill_rand(N, A);
      #pragma omp flush
      flag = 1;
      #pragma omp flush (flag)
    }
    #pragma omp section
    {
      #pragma omp flush (flag)
      while (flag == 0){
        #pragma omp flush (flag)
      }
      #pragma omp flush
      sum = Sum_array(N, A);
    }
  }
}
```

Use `flag` to signal when the "produced" value is ready

Flush forces refresh to memory. Guarantees that the other thread sees the new value of `A`

Flush needed on both "reader" and "writer" sides of the communication

Notice you must put the flush inside the while loop to make sure the updated `flag` variable is seen

Data Races and `flush`

- The program in the previous slide works possibly everywhere
- But technically, it has a race on the variable `flag` and a compiler is free to break this program
- Later when we explore atomics in more details, we'll talk about how to fix this

Atomics and Synchronization Flags (1/2)

```
int main() {
  double *A, sum, runtime;
  int numthreads, flag = 0;
  A = (double *)malloc(N*sizeof(double));
  #pragma omp parallel sections
  {
    #pragma omp section
    {
      fill_rand(N, A);
      #pragma omp flush
      flag = 1;
      #pragma omp flush (flag)
    }
    #pragma omp section
    {
      #pragma omp flush (flag)
      while (flag == 0){
        #pragma omp flush (flag)
      }
      #pragma omp flush
      sum = Sum_array(N, A);
    }
  }
}
```

- This program only works since we don't really care about the value of flag
  - All we care is that the flag no longer equals zero
- Why is there a problem communicating the actual value of flag?
- Doesn't the flush assure the flag value is cleanly communicated?

Atomics and Synchronization Flags (2/2)

```
int main() {
  double *A, sum, runtime;
  int numthreads, flag = 0;
  A = (double *)malloc(N*sizeof(double));
  #pragma omp parallel sections
  {
    #pragma omp section
    {
      fill_rand(N, A);
      #pragma omp flush
      flag = 1;
      #pragma omp flush (flag)
    }
    #pragma omp section
    {
      #pragma omp flush (flag)
      while (flag == 0){
        #pragma omp flush (flag)
      }
      #pragma omp flush
      sum = Sum_array(N, A);
    }
  }
}
```

- If `flag` straddles word boundaries or is a data type that consists of multiple words, it is possible for the read to load a partial result
- We need the ability to manage updates to memory locations atomically

Outline

- Task (OpenMP 3.0)
  - `task` Constructs
  - Task Clauses
- OpenMP Memory Model (`flush`)
- `atomic` (OpenMP 3.1)
- `threadprivate` Data
- Case Study: N-Body Problem

Recall: `atomic` Construct

Synchronization: `atomic` (Basic Form)

- Atomic provides mutual exclusion but only applies to the update of a memory location (the update of `X` in the following example)
- The statement inside the atomic must be one of the following forms:
  - `x binop= expr`
  - `x++`
  - `++x`
  - `x--`
  - `--x`
- `x` is an lvalue of scalar type and binop is a non-overloaded built-in operator

```
#pragma omp parallel
{
  double tmp, B;
  B = DOIT();
  tmp = big_ugly(B);
#pragma omp atomic
  X += tmp;
}
```

Additional forms of atomic were added in OpenMP 3.1

- The original OpenMP atomic was too restrictive
  - For example it didn't include a simple atomic store

The OpenMP 3.1 atomics (1/2)

- Atomic was expanded to cover the full range of common scenarios where you need to protect a memory operation so it occurs atomically:
  - `#pragma omp atomic [read | write | update | capture]`
- Atomic can protect loads
  - `#pragma omp atomic read`

    `v = x;`
- Atomic can protect stores
  - `#pragma omp atomic write`

    `x = expr;`
- Atomic can protect updates to a storage location (this is the default behavior … i.e. when you don't provide a clause)
  - `#pragma omp atomic update`

    `x++;` or `++x;` or `x--;` or `--x;` or

    `x binop= expr;` or `x = x binop expr;`

This is the original OpenMP atomic

The OpenMP 3.1 atomics (2/2)

- Atomic can protect the assignment of a value (its capture) AND an associated update operation:
  - `#pragma omp atomic capture`
    `statement or structured block`
- Where the statement is one of the following forms:
  - `v=x++;  v=++x;  v=x--;  v=--x;  v=x binop expr;`
- Where the structured block is one of the following forms:
  - `{v=x; x binop= expr;}`
    `{v=x; x = x binop expr;}`
    `{v=x; x++;}`
    `{++x; v=x;}`
    `{v=x; x--;}`
    `{--x; v=x;}`
    `{x binop= expr; v=x;}`
    `{x = x binop expr; v=x;}`
    `{v=x; ++x;}`
    `{x++; v=x;}`
    `{v=x; --x;}`
    `{x--; v=x;}`

Atomics and Synchronization Flags

```
int main() {
  double *A, sum, runtime;   int numthreads, flag = 0, flg_tmp;
  A = (double *)malloc(N*sizeof(double));
  #pragma omp parallel sections
  {
    #pragma omp section
    {
      fill_rand(N, A);
      #pragma omp flush
      #pragma omp atomic write
      flag = 1;
      #pragma omp flush (flag)
    }
    #pragma omp section
    {
      while (1){
        #pragma omp flush (flag)
        #pragma omp atomic read
        flg_tmp = flag;
        if (flg_tmp==1) break;
      }
      #pragma omp flush
      sum = Sum_array(N, A);
    }
  }
}
```

This program is truly race free

The reads and writes of flag are protected so the two threads can not conflict

Outline

- Task (OpenMP 3.0)
  - `task` Constructs
  - Task Clauses
- OpenMP Memory Model (`flush`)
- `atomic` (OpenMP 3.1)
- `threadprivate` Data
- Case Study: N-Body Problem

Data Sharing: `threadprivate`

- Preserves global scope for per-thread storage
- Legal for name-space-scope and file-scope
- Different from making them `private`
  - With `private`, global variables are masked
  - `threadprivate` preserves global scope within each thread
- `threadprivate` variables can be initialized using `copyin` or at time of definition (using language-defined initialization capabilities)

```
void foo(int var) {
  printf("%d\n", var);
}
int main() {
  int var;
  #pragma omp parallel private(var)
    foo(var);
  return 0;
}
```

```
int var;
#pragma omp threadprivate(var)
void foo() {
  printf("%d\n", var);
}
int main() {
  #pragma omp parallel
    foo();
  return 0;
}
```

A `threadprive` Example

- Use `threadprivate` to create a counter for each thread

```
int A = 0;
#pragma omp threadprivate(A)

int main()
{
   #pragma omp parallel
   do_domething_to(A);
   ...
}
```

Data Copying : copyin

- You initialize threadprivate data using a copyin clause

```
int A = 0;
#pragma omp threadprivate(A)

int main()
{
   A = 1;
   #pragma omp parallel copyin(A)
   do_domething_to(A);
   ...
}
```

Data Copying : copyprivate

- Used with a `single` region to broadcast values of privates from one member of a team to the rest of the team

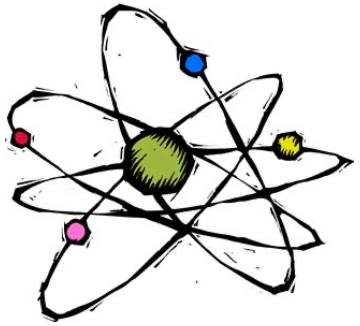
```
#include <omp.h>
void input_parameters(int, int);
void do_work(int, int);

void main() {
  int Nsize, choice;

  #pragma omp parallel private (Nsize, choice)
  {
    #pragma omp single copyprivate (Nsize, choice)
    input_parameters(Nsize, choice);

    do_work(Nsize, choice);
  }
}
```

Outline

- Task (OpenMP 3.0)
  - `task` Constructs
  - Task Clauses
- OpenMP Memory Model (`flush`)
- `atomic` (OpenMP 3.1)
- `threadprivate` Data
- Case Study: N-Body Problem

The N-Body Problem

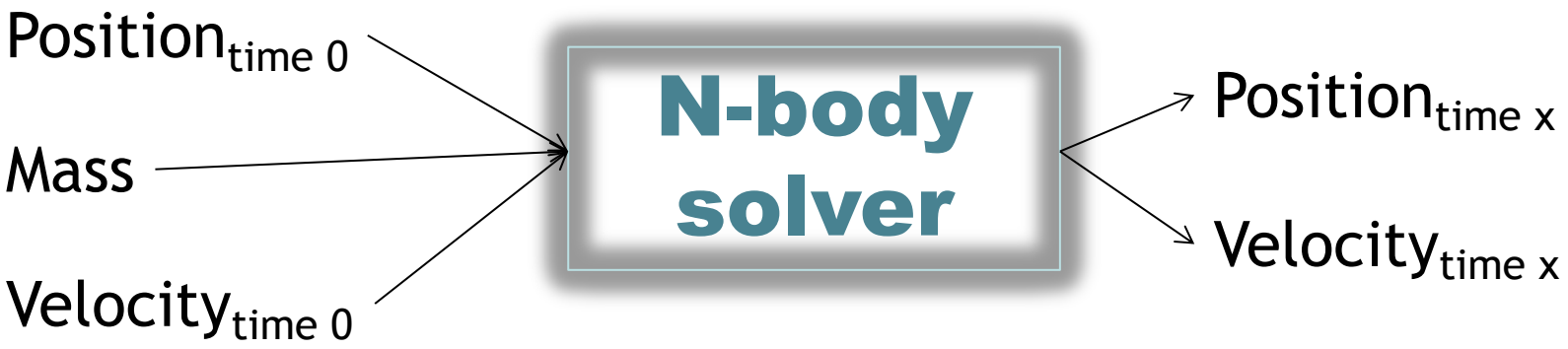
- Find the positions and velocities of a collection of interacting particles over a period of time
  - A collection of stars
  - A collection of molecules or atoms
  - …



- An n-body solver is a program that finds the solution to an n-body problem by simulating the behavior of the particles

Simulating Motion of Planets

- Determine the positions and velocities:
  - Newton's second law of motion
  - Newton's law of universal gravitation



The Force on Particle q exerted by k

- Suppose there are two particles q and k
  - q has position $\boldsymbol{s}_q(t)$ at time t
  - k has position $\boldsymbol{s}_k(t)$ at time t
- The force on q exerted by k is

$$\mathbf{f}_{qk}(t) = -\frac{Gm_q m_k}{\left|\mathbf{s}_q(t) - \mathbf{s}_k(t)\right|^3}\left[\mathbf{s}_q(t) - \mathbf{s}_k(t)\right]$$

- G is the gravitational constant

$$6.673 \times 10^{-11}\mathrm{m}^3/(\mathrm{kg}\cdot\mathrm{s}^2)$$

The Total Force on Particle q

- The total force on q is

$$\mathbf{F}_q(t) = \sum_{\substack{k=0 \\ k \neq q}}^{n-1} \mathbf{f}_{qk} = -Gm_q \sum_{\substack{k=0 \\ k \neq q}}^{n-1} \frac{m_k}{\left|\mathbf{s}_q(t) - \mathbf{s}_k(t)\right|^3} \left[\mathbf{s}_q(t) - \mathbf{s}_k(t)\right]$$

- $\mathbf{F}_q(t) = m_q \mathbf{a}_q(t) = m_q \mathbf{s}_q''(t)$ $\qquad$ $\mathbf{v}_q(t) = \mathbf{s}_q'(t)$
  - The acceleration of an object is given by the second derivative of its position
  - Newton's second law of motion
- The acceleration of q is

$$\mathbf{s}_q''(t) = -G \sum_{\substack{j=0 \\ j \neq q}}^{n-1} \frac{m_j}{\left|\mathbf{s}_q(t) - \mathbf{s}_j(t)\right|^3} \left[\mathbf{s}_q(t) - \mathbf{s}_j(t)\right]$$

Serial Pseudo-Code (A Basic Algorithm)

```
Get input data;
for each timestep {
   if (timestep output) Print positions and velocities of
       particles;
   for each particle q
      Compute total force on q;
   for each particle q
      Compute position and velocity of q;
}
Print positions and velocities of particles;
```

```
for each particle q {
   for each particle k != q {
      x_diff = pos[q][X] - pos[k][X];
      y_diff = pos[q][Y] - pos[k][Y];
      dist = sqrt(x_diff*x_diff + y_diff*y_diff);
      dist_cubed = dist*dist*dist;
      forces[q][X] -= G*masses[q]*masses[k]/dist_cubed * x_diff;
      forces[q][Y] -= G*masses[q]*masses[k]/dist_cubed * y_diff;
   }
}
```

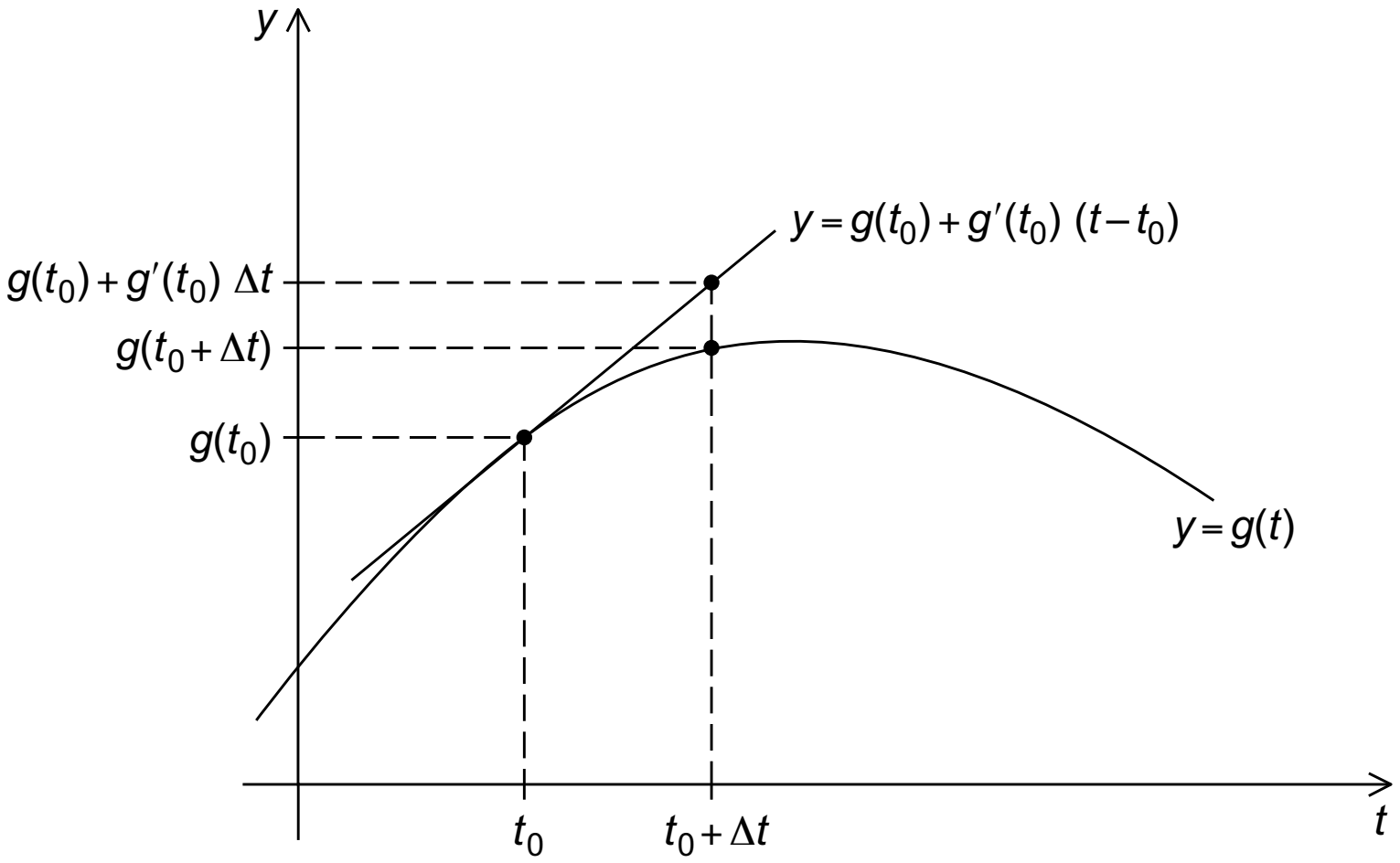
A Reduced Algorithm

- Newton's third law of motion
  - For every action there is an equal and opposite reaction
  - If the force on q due to k is f_{qk}, then the force on k due to q is $-f_{qk}$

```
for each particle q
   forces[q] = 0;
for each particle q {
   for each particle k > q {
      x_diff = pos[q][X] – pos[k][X];
      y_diff = pos[q][Y] – pos[k][Y];
      dist = sqrt(x_diff*x_diff + y_diff*y_diff);
      dist_cubed = dist*dist*dist;
      force_qk[X] = G*masses[q]*masses[k]/dist_cubed * x_diff;
      force_qk[Y] = G*masses[q]*masses[k]/dist_cubed * y_diff

      forces[q][X] += force_qk[X];
      forces[q][Y] += force_qk[Y];
      forces[k][X] –= force_qk[X];
      forces[k][Y] –= force_qk[Y];
   }
}
```

Using the Tangent Line to Approximate a Function

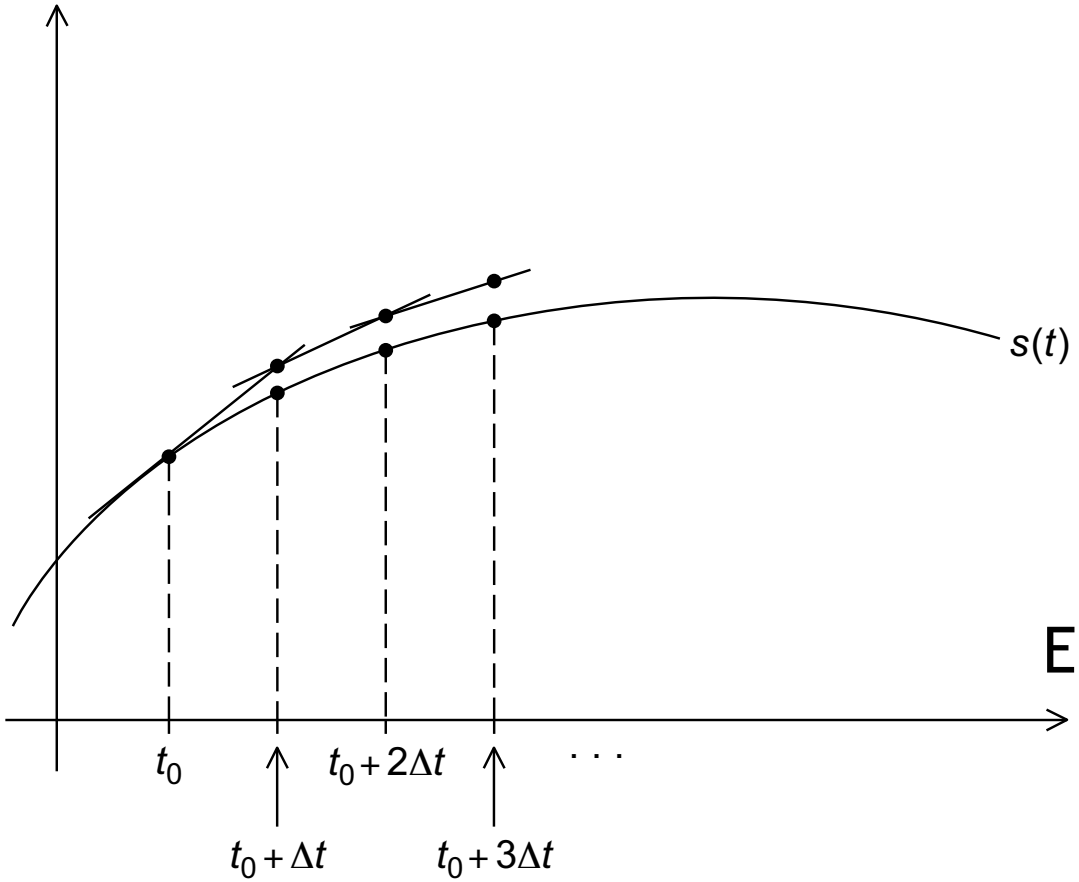
- $g(t+\Delta t) \approx g(t_0) + g'(t_0)(t+\Delta t - t) = g(t_0) + \Delta t g'(t_0)$



Approximating New Position and Velocity

$$\mathbf{s}_q(\Delta t) \approx \mathbf{s}_q(0) + \Delta t\,\mathbf{s}'_q(0) = \mathbf{s}_q(0) + \Delta t\,\mathbf{v}_q(0),$$

$$\mathbf{v}_q(\Delta t) \approx \mathbf{v}_q(0) + \Delta t\,\mathbf{v}'_q(0) = \mathbf{v}_q(0) + \Delta t\,\mathbf{a}_q(0) = \mathbf{v}_q(0) + \Delta t\frac{1}{m_q}\mathbf{F}_q(0)$$



Euler's method

The Code for Computing Position and Velocity

```
for each particle q {
   for each particle k != q {
      x_diff = pos[q][X] – pos[k][X];
      y_diff = pos[q][Y] – pos[k][Y];
      dist = sqrt(x_diff*x_diff + y_diff*y_diff);
      dist_cubed = dist*dist*dist;
      forces[q][X] –= G*masses[q]*masses[k]/dist_cubed * x_diff;
      forces[q][Y] –= G*masses[q]*masses[k]/dist_cubed * y_diff;
   }
}
```

```
pos[q][X] += delta_t*vel[q][X];
pos[q][Y] += delta_t*vel[q][Y];
vel[q][X] += delta_t/masses[q]*forces[q][X];
vel[q][Y] += delta_t/masses[q]*forces[q][Y];
```

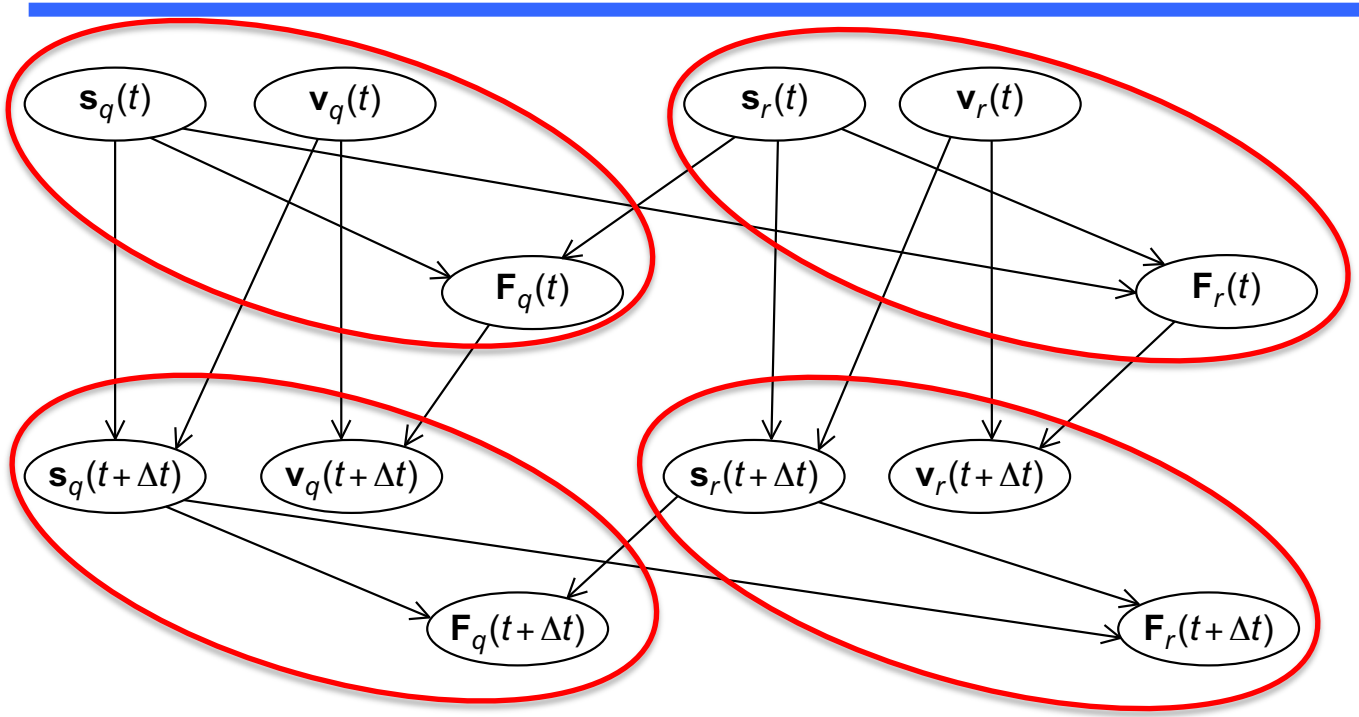
Serial N-Body Simulation (Basic Algorithm)

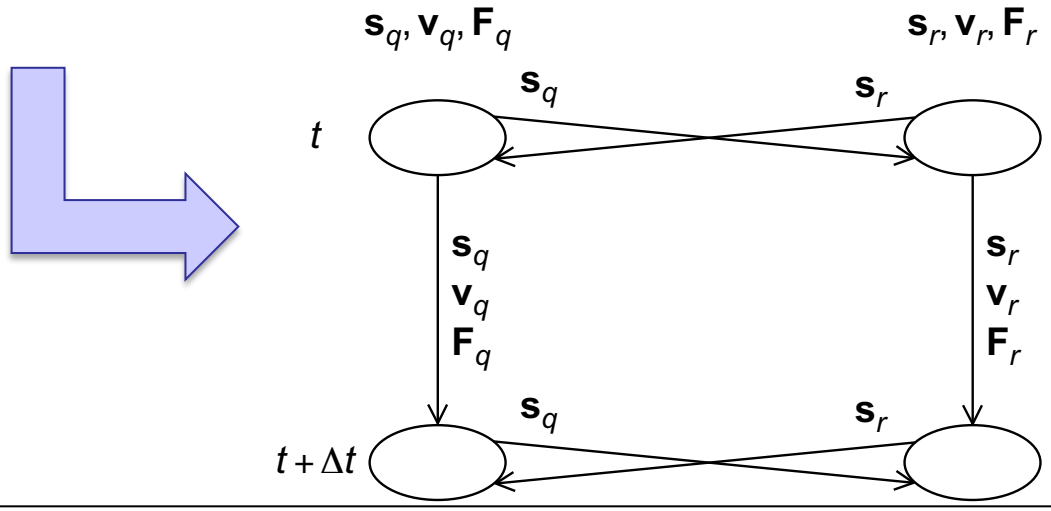
```
#define DIM 2   /* Two-dimensional system */
#define X 0     /* x-coordinate subscript */
#define Y 1     /* y-coordinate subscript */

const double G = 6.673e-11;
/* Gravitational constant. */
/* Units are m^3/(kg*s^2)  */

typedef double vect_t[DIM];
/* Vector type for position, etc. */

struct particle_s {
   double m;  /* Mass     */
   vect_t s;  /* Position */
   vect_t v;  /* Velocity */
};
```

Serial N-Body Simulation (Basic Algorithm)

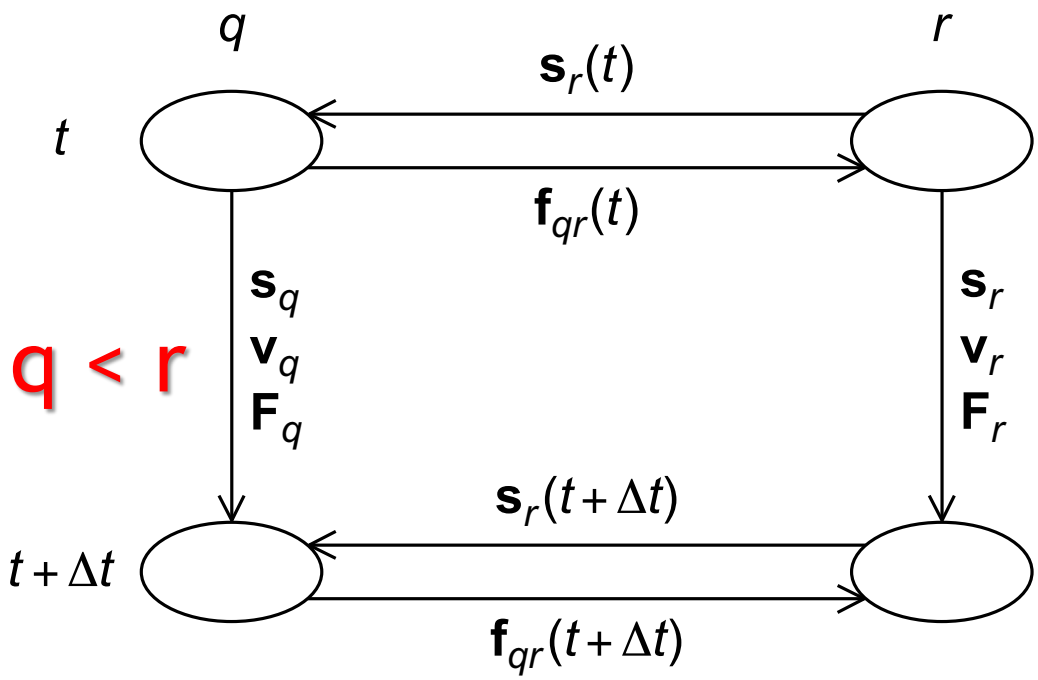
```
#include <string.h>
...
int main(int argc, char* argv[]) {
   int n;            /* Number of particles        */
   int n_steps;      /* Number of timesteps        */
   int step;         /* Current step               */
   int part;         /* Current particle           */
   vect_t* forces;   /* Forces on each particle    */
   ...
   forces = malloc(n*sizeof(vect_t));
   for (step = 1; step <= n_steps; step++) {
      ...
      /* Assign 0 to each element of the
         forces array */
      memset(forces, 0, n*sizeof(vect_t));
      for (part = 0; part < n; part++)
         Compute_force(part, forces, curr, n);
      ...
   }
}
```

Communications Among Tasks in the Basic N-Body Solver





Communications Among Agglomerated Tasks in the Reduced N-Body Solver

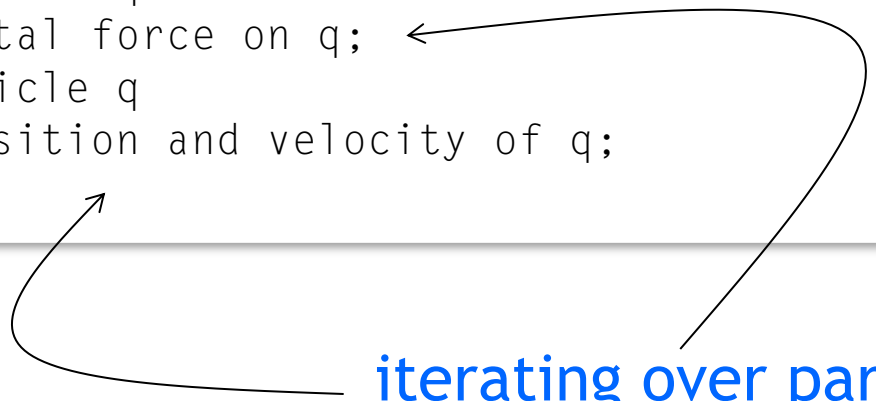


Computing the total force on q in the reduced algorithm

```
for each particle k > q {
   x_diff = pos[q][X] - pos[k][X];
   y_diff = pos[q][Y] - pos[k][Y];
   dist = sqrt(x_diff*x_diff + y_diff*y_diff);
   dist_cubed = dist*dist*dist;
   force_qk[X] = G*masses[q]*masses[k]/dist_cubed * x_diff;
   force_qk[Y] = G*masses[q]*masses[k]/dist_cubed * y_diff;

   forces[q][X] += force_qk[X];
   forces[q][Y] += force_qk[Y];
   forces[k][X] -= force_qk[X];
   forces[k][Y] -= force_qk[Y];
}
```

Serial Pseudo-Code

```
for each timestep {
  if (timestep output) Print positions and velocities of particles;
  for each particle q
     Compute total force on q;
  for each particle q
     Compute position and velocity of q;
}
```



- In principle, parallelizing the two inner for loops will map tasks/particles to cores

First Attempt to Parallelize It

```
    for each timestep {
       if (timestep output) Print positions and velocities of
           particles;
#      pragma omp parallel for
       for each particle q
          Compute total force on q;
#      pragma omp parallel for
       for each particle q
          Compute position and velocity of q;
    }
```

- Let's check for race conditions caused by loop-carried dependences

The Two Loops

```
#  pragma omp parallel for
   for each particle q {
      forces[q][X] = forces[q][Y] = 0;
      for each particle k != q {
         x_diff = pos[q][X] – pos[k][X];
         y_diff = pos[q][Y] – pos[k][Y];
         dist = sqrt(x_diff*x_diff + y_diff*y_diff);
         dist_cubed = dist*dist*dist;
         forces[q][X] –= G*masses[q]*masses[k]/dist_cubed * x_diff;
         forces[q][Y] –= G*masses[q]*masses[k]/dist_cubed * y_diff;
      }
   }
```

```
#  pragma omp parallel for
   for each particle q {
      pos[q][X] += delta_t*vel[q][X];
      pos[q][Y] += delta_t*vel[q][Y];
      vel[q][X] += delta_t/masses[q]*forces[q][X];
      vel[q][Y] += delta_t/masses[q]*forces[q][Y];
   }
```

Repeated forking and joining of threads

The same team of threads will be used in both inner loops and for every iteration of the outer loop

```
#  pragma omp parallel
   for each timestep {
      if (timestep output) Print positions and velocities of
          particles;
#     pragma omp for
      for each particle q
         Compute total force on q;
#     pragma omp for
      for each particle q
         Compute position and velocity of q;
   }
```

But every thread will print all the positions and velocities

Adding the `single` Directive

```
#  pragma omp parallel
   for each timestep {
      if (timestep output) {
#        pragma omp single
         Print positions and velocities of particles;
      }
#     pragma omp for
      for each particle q
         Compute total force on q;
#     pragma omp for
      for each particle q
         Compute position and velocity of q;
   }
```

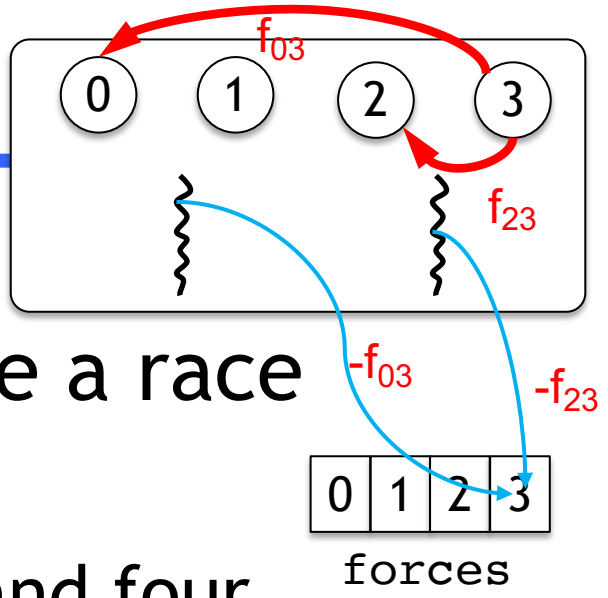
Parallelizing the Reduced Solver Using OpenMP

```
#  pragma omp parallel
   for each timestep {
      if (timestep output) {
#        pragma omp single
         Print positions and velocities of particles;
      }
#     pragma omp for
      for each particle q
         forces[q] = 0.0;
#     pragma omp for
      for each particle q
         Compute total force on q;
#     pragma omp for
      for each particle q
         Compute position and velocity of q;
   }
```

Computing the total force on q in the reduced algorithm

```
for each particle k > q {
   x_diff = pos[q][X] - pos[k][X];
   y_diff = pos[q][Y] - pos[k][Y];
   dist = sqrt(x_diff*x_diff + y_diff*y_diff);
   dist_cubed = dist*dist*dist;
   force_qk[X] = G*masses[q]*masses[k]/dist_cubed * x_diff;
   force_qk[Y] = G*masses[q]*masses[k]/dist_cubed * y_diff;

   forces[q][X] += force_qk[X];
   forces[q][Y] += force_qk[Y];
   forces[k][X] -= force_qk[X];
   forces[k][Y] -= force_qk[Y];
}
```

Problems



- $\mathbf{F}_3 = -\mathbf{f}_{03} - \mathbf{f}_{13} - \mathbf{f}_{23}$
- Updates to `forces[3]` create a race condition
  - Suppose we have two threads and four particles
  - Thread 0 would compute $-\mathbf{f}_{03}-\mathbf{f}_{13}$, while thread 1 would compute $-\mathbf{f}_{23}$
- In fact, this is the case in general
  - Updates to the elements of the forces array introduce race conditions into the code

First Solution Attempt

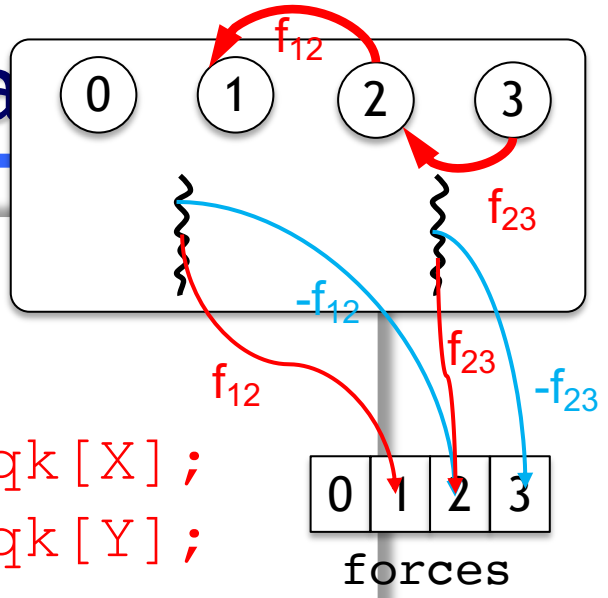
before all the updates to forces

```
#  pragma omp critical
   {
      forces[q][X] += force_qk[X];
      forces[q][Y] += force_qk[Y];
      forces[k][X] -= force_qk[X];
      forces[k][Y] -= force_qk[Y];
   }
```

Access to the forces array will be effectively serialized!!!

What About Naming Critica

```
#pragma omp critical first
{
    forces[q][X] += force_qk[X];
    forces[q][Y] += force_qk[Y];
}
#pragma omp critical second
{
    forces[k][X] += force_qk[X];
    forces[k][Y] += force_qk[Y];
}
```



The two critical sections can be executed simultaneously

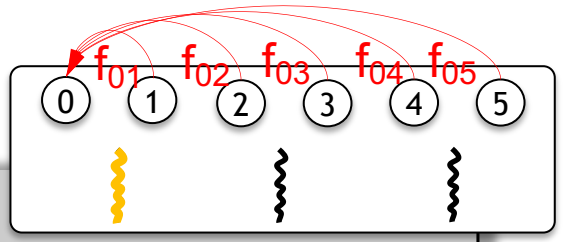
Second Solution Attempt

```
omp_set_lock(&locks[q]);
forces[q][X] += force_qk[X];
forces[q][Y] += force_qk[Y];
omp_unset_lock(&locks[q]);

omp_set_lock(&locks[k]);
forces[k][X] -= force_qk[X];
forces[k][Y] -= force_qk[Y];
omp_unset_lock(&locks[k]);
```

Use one lock for each particle

Force Computations for Reduced Algorithm with Block Partition

- Suppose we have three threads and six particles



		Thread		
Thread	**Particle**	**0**	**1**	**2**
0	0	$\mathbf{f}_{01}+\mathbf{f}_{02}+\mathbf{f}_{03}+\mathbf{f}_{04}+\mathbf{f}_{05}$	0	0
	1	$-\mathbf{f}_{01}+\mathbf{f}_{12}+\mathbf{f}_{13}+\mathbf{f}_{14}+\mathbf{f}_{15}$	0	0
1	2	$-\mathbf{f}_{02}-\mathbf{f}_{12}$	$\mathbf{f}_{23}+\mathbf{f}_{24}+\mathbf{f}_{25}$	0
	3	$-\mathbf{f}_{03}-\mathbf{f}_{13}$	$-\mathbf{f}_{23}+\mathbf{f}_{34}+\mathbf{f}_{35}$	0
2	4	$-\mathbf{f}_{04}-\mathbf{f}_{14}$	$-\mathbf{f}_{24}-\mathbf{f}_{34}$	$\mathbf{f}_{45}$
	5	$-\mathbf{f}_{05}-\mathbf{f}_{15}$	$-\mathbf{f}_{25}-\mathbf{f}_{35}$	$-\mathbf{f}_{45}$

Block partition: `schedule(static, total_iterations/thread_count)`

Force Computations for Reduced Algorithm with Cyclic Partition

Thread	Particle	Thread		
		0	1	2
0	0	$\mathbf{f}_{01}+\mathbf{f}_{02}+\mathbf{f}_{03}+\mathbf{f}_{04}+\mathbf{f}_{05}$	0	0
1	1	$-\mathbf{f}_{01}$	$\mathbf{f}_{12}+\mathbf{f}_{13}+\mathbf{f}_{14}+\mathbf{f}_{15}$	0
2	2	$-\mathbf{f}_{02}$	$-\mathbf{f}_{12}$	$\mathbf{f}_{23}+\mathbf{f}_{24}+\mathbf{f}_{25}$
0	3	$-\mathbf{f}_{03}+\mathbf{f}_{34}+\mathbf{f}_{35}$	$-\mathbf{f}_{13}$	$-\mathbf{f}_{23}$
1	4	$-\mathbf{f}_{04}-\mathbf{f}_{34}$	$-\mathbf{f}_{14}+\mathbf{f}_{45}$	$-\mathbf{f}_{24}$
2	5	$-\mathbf{f}_{05}-\mathbf{f}_{35}$	$-\mathbf{f}_{15}-\mathbf{f}_{45}$	$-\mathbf{f}_{25}$

Cyclic partition: `schedule(static, 1)`

Workload distribution matters!

Revised Algorithm for Force Computations

```
#  pragma omp for
   for each particle q {
      force_qk[X] = force_qk[Y] = 0;
      for each particle k > q {
         x_diff = pos[q][X] – pos[k][X];
         y_diff = pos[q][Y] – pos[k][Y];
         dist = sqrt(x_diff*x_diff + y_diff*y_diff);
         dist_cubed = dist*dist*dist;
         force_qk[X] = G*masses[q]*masses[k]/dist_cubed * x_diff;
         force_qk[Y] = G*masses[q]*masses[k]/dist_cubed * y_diff;

         loc_forces[my_rank][q][X] += force_qk[X];
         loc_forces[my_rank][q][Y] += force_qk[Y];
         loc_forces[my_rank][k][X] –= force_qk[X];
         loc_forces[my_rank][k][Y] –= force_qk[Y];
      }
   }
```

`loc_forces` is a shared array, but each thread accesses its own elements

```
#  pragma omp for
   for (q = 0; q < n; q++) {
      forces[q][X] = forces[q][Y] = 0;
      for (thread = 0; thread < thread_count; thread++) {
         forces[q][X] += loc_forces[thread][q][X];
         forces[q][Y] += loc_forces[thread][q][Y];
      }
   }
```

Evaluation Results

- 400 particles for 1000 timesteps

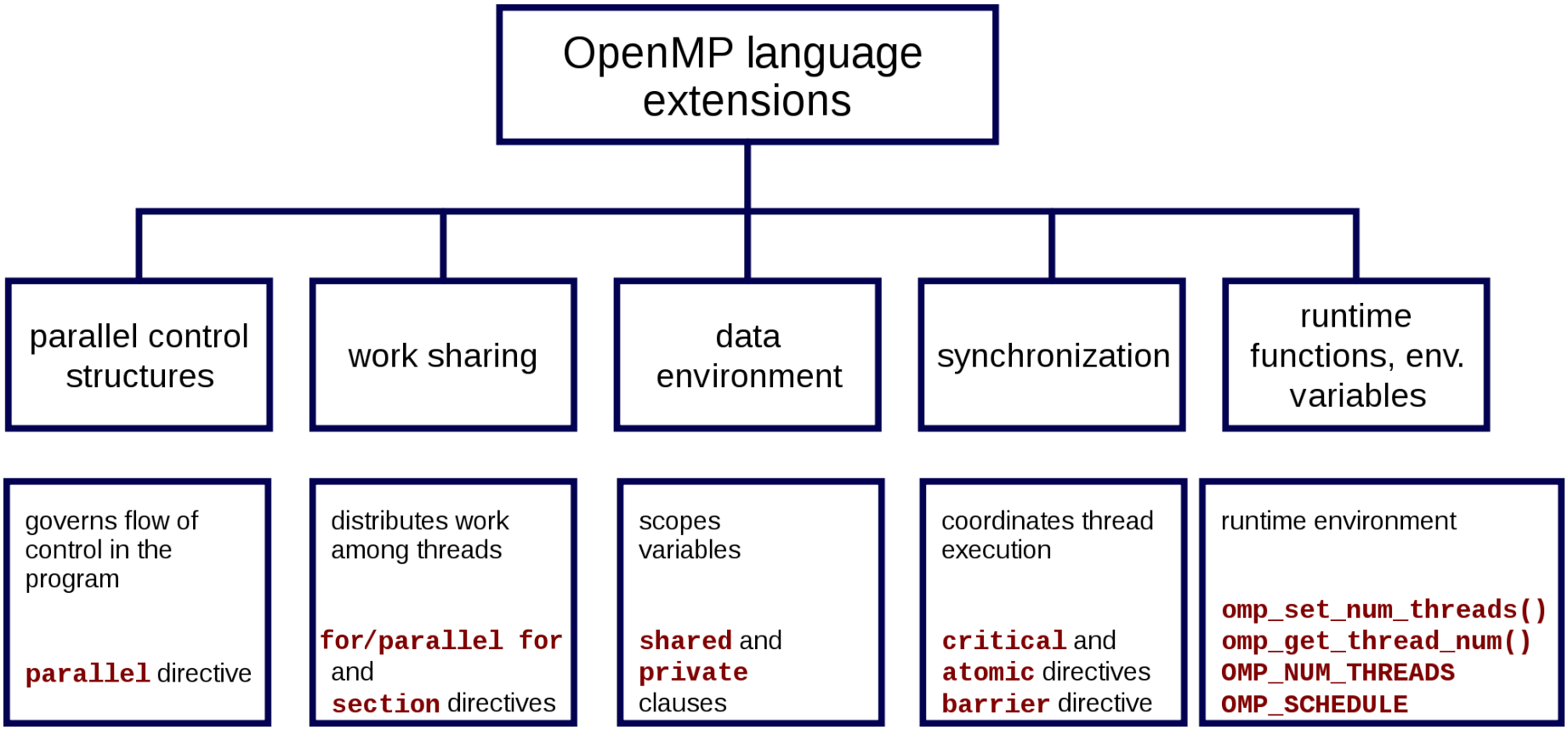
Table 6.3 Run-Times of the *n*-Body Solvers Parallelized with OpenMP (times are in seconds)

Threads	Basic	Reduced Default Sched	Reduced Forces Cyclic	Reduced All Cyclic
1	7.71	3.90	3.90	3.90
2	3.87	2.94	1.98	2.01
4	1.95	1.73	1.01	1.08
8	0.99	0.95	0.54	0.61

OpenMP Summary

- We have covered most of OpenMP
  - Enough so you can start writing real parallel applications with OpenMP
- `#pragma omp parallel`
- `#pragma omp for`
- `#pragma omp critical`
- `#pragma omp atomic`
- `#pragma omp barrier`
- `#pragma omp single`
- `#pragma omp section`
- `#pragma omp sections`
- `#pragma omp flush`
- Data environment clauses
  - `private (variable_list)`
  - `firstprivate (variable_list)`
  - `lastprivate (variable_list)`
  - `reduction(+:variable_list)`
- Tasks (remember ... private data is made `firstprivate` by default)
  - `#pragma omp task`
  - `#pragma omp taskwait`
- `#pragma threadprivate(variable_list)`

OpenMP: The 30,000 Feet Perspective



References

- Tim Mattson, Michael Wrinn, and Mark Bull, “A “Hands-on” Introduction to OpenMP”
- Tim Mattson, “Shared Memory Programming with OpenMP - Basics”, 2012 Short Course on Parallel Programming, Berkeley
- Tim Mattson, “More about OpenMP- New Features” , 2012 Short Course on Parallel Programming, Berkeley
- Intel Software College, “Programming with OpenMP”
- Gabriel Mateescu, “Writing and tuning OpenMP programs on distributed shared memory platforms”
- UCB CS267 Course, “Applications of Parallel Computers”
- Jay Hoeflinger and Bronis de Supinski, “The OpenMP Memory Model”, IWOMP 2005.