10. Cilk and Related Systems

Parallel Programming

Dr Hamidreza Khaleghzadeh
School of Computing
University of Portsmouth

Goals

- Introduce the *Cilk* parallel programming system and its programming model.
- Illustrate with simple examples like *QuickSort*
- Briefly review the related *Fork/Join framework* in Java 1.7, and *Intel Threading Building Blocks*.
- A glance at another recursive algorithm – Barnes Hut.

Cilk

- *Cilk* is a parallel programming language developed at MIT in the mid-1990s by Charles Leiserson and collaborators.
- It is particularly well suited to a *divide-and-conquer* (recursive) style of parallelism.
- Works by repeatedly spawning *lightweight threads* that are *micro-scheduled* by the run-time system of the language.
- The scheduling mechanism built into the language is called *work-stealing*.

Cilk Developments

- The original *Cilk* is a small extension to *ANSI C*.
- In 2006 a version of Cilk was commercialized by a spin-off from MIT called Cilk Arts.
- Acquired by Intel around 2009, and now marketed as *Cilk Plus* – an extension of *C++*.
- Ideas similar to Cilk are now incorporated in other systems – e.g. the *Fork/Join framework* in Java 1.7.

The Fibonacci Series

- Fibonacci sequence probably familiar:

    0, 1, 1, 2, 3, 5, 8, 13, 21, ..., Fib_n, ...
- Each term is the sum of the previous two terms.
- There is a recursive calculation of the *n* th term in this series, beloved of functional programmers (despite or because of it being a spectacularly inefficient algorithm).
- A standard C version of this algorithm is given on the next slide.

Fibonacci Function (C)

```
int fib (int n) {
    if (n<2) {
        return n;
    }
    else {
        int x, y;
        x = fib (n-1);
        y = fib (n-2);
        return (x+y);
    }
}
```

Divide and Conquer

- This recursive pattern, where a calculation is broken into two or more smaller calculations, is often called *divide and conquer*.
- It can often give rise to a natural form of parallelism in which the *sub-tasks are run concurrently*.
- Cilk can be particularly effective for expressing this kind of parallelism.

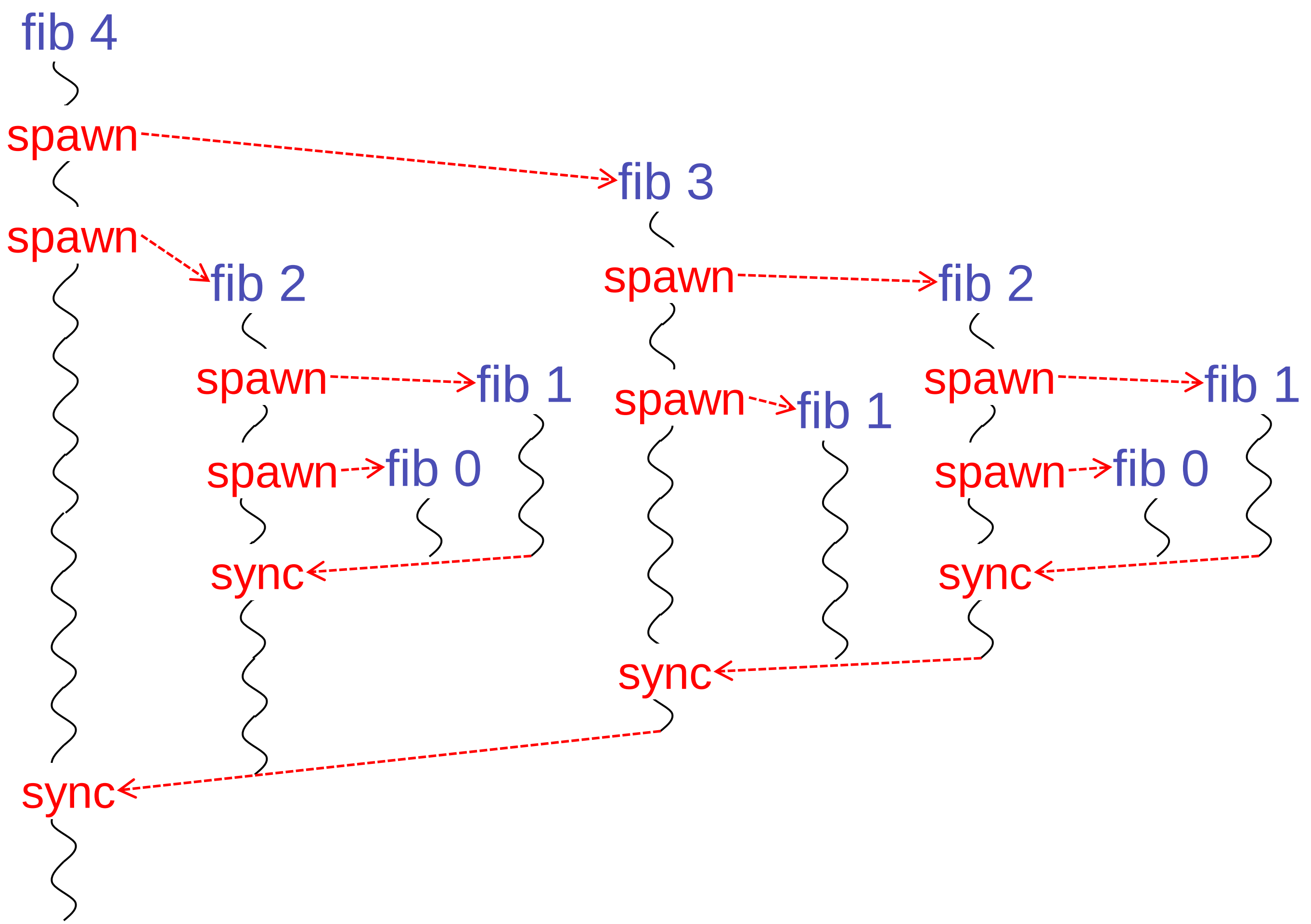
Fibonacci Function (Cilk)

```
cilk int fib (int n) {
    if (n<2) {
        return n;
    }
    else {
        int x, y;
        x = spawn fib (n-1);
        y = spawn fib (n-2);
        sync ;
        return (x+y);
    }
}
```

New Keywords

- The core of Cilk uses just three new keywords: cilk, spawn and sync.
  - Keyword cilk identifies a *cilk procedure* – a parallel version of a C function.
  - Keyword spawn indicates a cilk procedure call be run *in parallel* with the calling thread.
  - Return values *written* by spawned procedures cannot be safely read until a sync statement is executed by the calling procedure - a kind of barrier or join synchronization across threads spawned by this procedure.

Thread Creation in fib



Proliferation of Threads

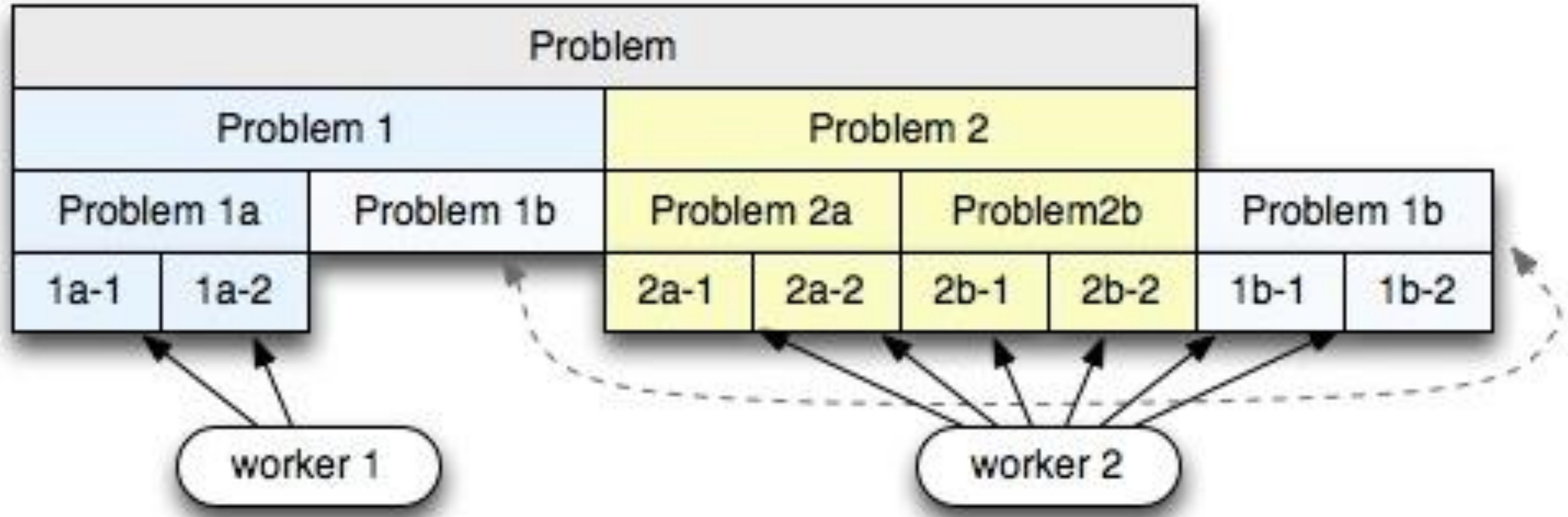
- Very rapidly have *many* (in this case exponentially many) lightweight threads.
- But Cilk micro-scheduling is *very efficient*, and overheads of spawn/sync operations is small.
  - At least an order of magnitude less than if these were, say POSIX threads or Java threads†
- Number of underlying POSIX-style threads typically equal to number of cores
  - execution of the lightweight Cilk threads shared between these.

†For the similar TBB (see later) thread vs task overhead ratio is 18 for Linux, 100 for Windows, according to TBB User Guide from Intel.

Work Stealing

- Each core maintains a *queue* (specifically a double-ended queue, or *deque*) of pending cilk procedures ("threads") awaiting execution.
- When one core runs out of work, it randomly chooses one of the other cores, and *steals* a pending procedure from the top of the "victim's" work deque.
  - "Near the top" implies this is a task spawned early in the recursion, which will probably spawn further threads as the "thief" processes it.
  - Minimizes synchronization between cores.

Visualization of Work Stealing†



†Image from
http://www.igvita.com/2012/02/29/work-stealing-and-recursive-partitioning-with-fork-join/

A More Interesting Example

- In the *QuickSort* algorithm, an array or section of an array is sorted by first *partitioning* it so that elements on the *left* side of the array are *less than* some randomly chosen *pivot value*, and those on the *right* side are *greater than* the pivot.
- The two sides of the partition are then independently sorted by a recursive call.
- A C version of the core QuickSort algorithm is given on the next slide.

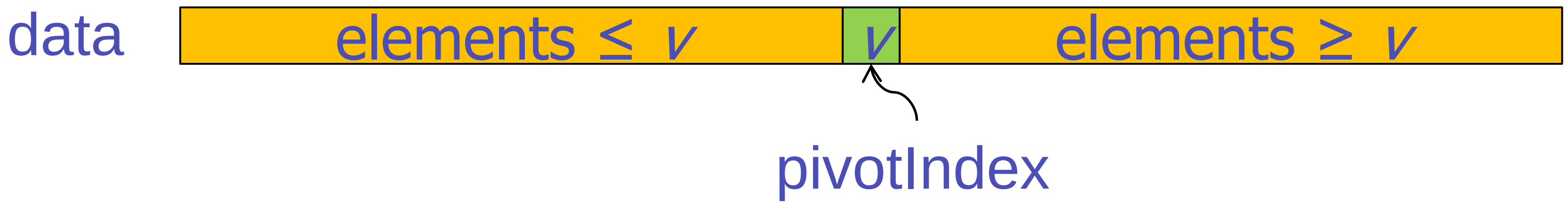
QuickSort in C

```
void quicksort(int data [], int first, int last) {
    if(last > first) {
        int pivotIndex = last ;
        pivotIndex = partition(data, first, last, pivotIndex) ;
        quicksort(data, first, pivotIndex - 1) ;
        quicksort(data, pivotIndex + 1, last) ;
    }
}
```

Behaviour of partition()

- Before call to partition:

- After call to partition:



Notes

- Both quicksort and partition functions operate on a subset of the data array determined by the index values first and last.
- The partition function can be implemented in various ways using $O(N)$ comparisons and swaps of pairs of elements.
  - One particular algorithm is given at http://en.wikipedia.org/wiki/Quicksort
- Note the *divide and conquer* nature of this very widely used algorithm.

QuickSort in Cilk

```
cilk void quicksort(int data [], int first, int last) {
    if(last > first) {
        int pivotIndex = last ;
        pivotIndex = partition(data, first, last, pivotIndex) ;
        spawn quicksort(data, first, pivotIndex - 1) ;
        spawn quicksort(data, pivotIndex + 1, last) ;
        sync ;
    }
}
```

Notes

- Simply change quicksort to a cilk procedure, and spawn it rather than calling it (remember to sync!)
- The most intricate part of the algorithm – the in-place partition function - hasn't changed at all. It remains an ordinary C function.
- For optimal results, parallel divide and conquer usually sets some *threshold problem size* below which it reverts to sequential computation.
  - I haven't done this here – we spawn all the way down to the *base case* of the recursion.

QuickSort Performance

- C version:

```
$ gcc -O2 qsort.c -o qsort
$ ./qsort
Completed in 1376 milliseconds
```

- Cilk version:

```
$ cilkc -O2 qsort.cilk -o qsort
$ ./qsort
Completed in 1547 milliseconds
$ ./qsort --nproc 2
Completed in 865 milliseconds
```

Notes on Timings

- Timings on my (dual core) laptop, for $N = 10{,}000{,}000$
- Overhead of converting C to Cilk here only around 12% - could probably be reduced by “thresholding” as described earlier.
- Speedup of Cilk version on two cores close to 1.8.
- -O2 is a recommended optimization level (without it C and Cilk both 2-3 times slower)

Postscript

- Sadly last MIT release of original Cilk was in 2007, and, while still available on their web site, no longer seems to be maintained.
- *Cilk Plus* keywords (cilk_spawn, cilk_sync) still available in Intel C++ compiler, but deprecated around 2017.
  - Intel now recommended to use OpenMP pragmas task and taskwait to replace these keywords. See:

https://software.intel.com/en-us/articles/migrate-your-application-to-use-openmp-or-intelr-tbb-instead-of-intelr-ciktm-plus

- Possible rebirth at MIP?: http://cilk.mit.edu/

Work Stealing in Java 1.7

- Java 1.7 introduced an implementation of fine-grained “threads” similar to that in Cilk, with its Fork/Join Framework.
- Basic ideas proposed by Doug Lea around 2000, and explicitly inspired by Cilk.

QuickSort using Fork/Join

```
public class ForkJoinQuickSort extends RecursiveAction {
    ...
    int first, last ;
    ForkJoinQuickSort(int first, int last) {
        this.first = first ;
        this.last  = last  ;
    }
    protected void compute() {
        if(last > first) {
            int pivotIndex = last ;
            pivotIndex = partition(data, first, last, pivotIndex) ;
            invokeAll(new ForkJoinQuickSort(first, pivotIndex - 1),
                        new ForkJoinQuickSort(pivotIndex + 1, last)) ;
        }
    }
}
```

Notes

- The main method does this:

```
ForkJoinQuickSort mainTask =
          new ForkJoinQuickSort(0, N-1) ;
ForkJoinPool mainPool = new ForkJoinPool(P) ;

mainPool.invoke(mainTask) ;
```

- compute method similar to run method on Thread.
- My implementation assumed data is a static field of the class ForkJoinQuickSort.

Java Fork/Join Performance

- Timings on my laptop...
- With *P* = 1:

  $ java ForkJoinQuickSort

  Completed in 2818 milliseconds
- With *P* = 2:

  $ java ForkJoinQuickSort

  Completed in 1544 milliseconds
- Speed up on 2 cores about 1.8 again.
- Slower than Cilk, but not optimized – thresholds may be more important here?

Intel Threading Building Blocks

- Intel *Threading Building Blocks* contain some ideas on task parallelism similar to Cilk, etc.
- A C++ template library, rather than C language extensions or Java classes.
- Task spawning features quite similar to Java 7 F/J, but lower-level
  - Note TBB predates release of Java F/J Framework by several years.
- Users encouraged, rather, to use higher level, more “declarative” templates from TBB, like parallel_for, parallel_reduce, ...

```
class FibTask: public tbb::task {
public:
    const long n;
    long* const sum;
    FibTask( long n_, long* sum_ ) : n(n_), sum(sum_) {}
    task* execute() {
        if(n < 2) {
            *sum = n ;
        }
        else {
            long x, y;
            FibTask& a = *new( allocate_child() ) FibTask(n-1,&x);
            FibTask& b = *new( allocate_child() ) FibTask(n-2,&y);
            // Set ref_count to 'two children plus one for the wait'.
            set_ref_count(3);
            spawn( b );
            spawn_and_wait_for_all(a);
            *sum = x+y;
        }
        return NULL;
    }
};
```

Task Parallel Fibonacci in TBB

- Sample on previous slide adapted from TBB User Guide.
- Main program might do:

```
long sum;
FibTask& a = *new(tbb::task::allocate_root()) FibTask(N, &sum);
tbb::task::spawn_root_and_wait(a) ;
printf("sum = %ld\n", sum) ;
```

- Not all TBB is so gruesome, but it does require some expertise in C++!

Special Topic:

BARNES-HUT N-BODY CODE

N-Body Codes

- *N-Body codes* are codes that simulate the movement of "*N*" bodies (*N* usually large) under some law of mutual attraction or repulsion.
- Examples include the "Galaxy" simulation and the molecular dynamics simulation posted as development project ideas in the week 8 lab script.
- Where the force law has a long range nature – e.g. the inverse square law of gravity – the Barnes-Hut method may be effective.

Complexity of Naïve Algorithm

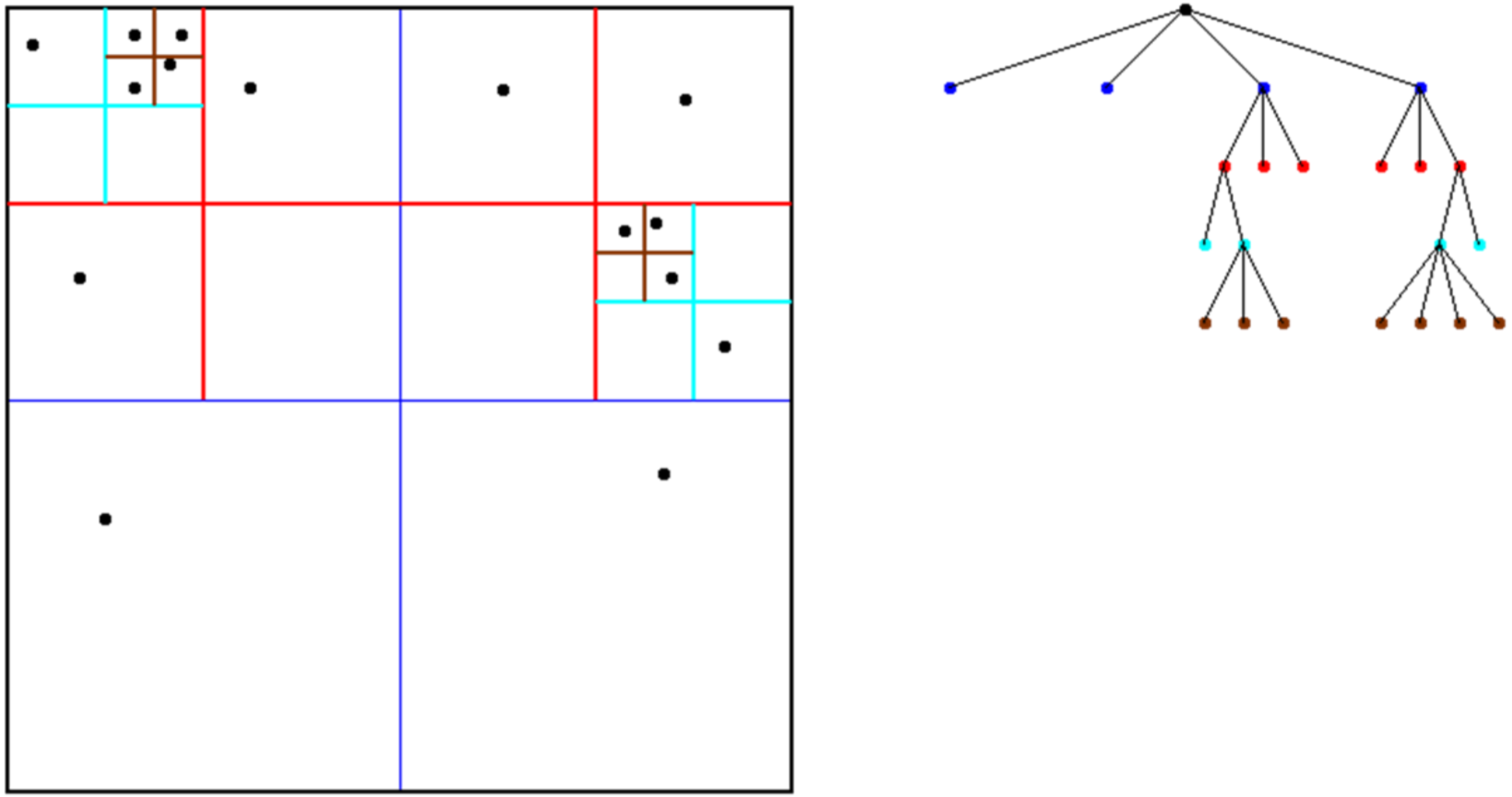
- The "naïve" algorithm for calculating force on all particles involves N force law contributions on each of N particles – complexity N^2.
- In Barnes-Hut we group together particles exerting a force on i th particle.
- If group is far from particle i, approximate force exerted by group as a whole as force exerted by a *single large mass* at the *centre of gravity* of the group.

Grouping: the Barnes Hut Tree

- First divide cubical region of 3d space into $2^3 = 8$ regions, halving each dimension.
- For every sub-region that contains any particles, divide again recursively to make an *oct-tree*, until "leaf" regions have at most one particle.

Two Dimensional Example

Adaptive quadtree where no square contains more than 1 particle

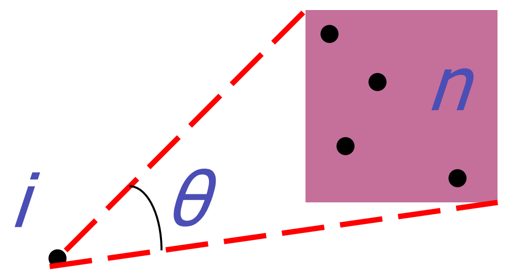


- Picture borrowed from

www.cs.berkeley.edu/~demmel/cs267/lecture26/lecture26.html

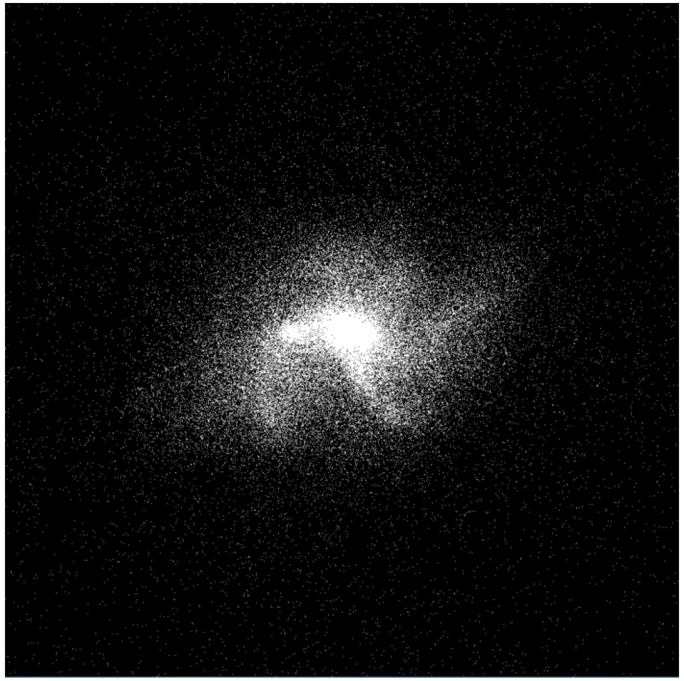
Barnes Hut Force Computation

- Force on a particle *i* – start with root node *n*:
  - if node *n* is "distant from" *i*, just add contribution to force on *i* from centre of mass of *n* – no need to visit children of *n*;
  - if node *n* is "close to" *i*, visit children of *n* and recurse.
- Hinges on definition of *distant from* and*close to*.
  - Basic idea is that a node representing some region of space is *distant* from a particle *i* if the angle it subtends is smaller than a threshold *opening angle*:



Complexity

- On average, number of nodes "opened" to compute force on i is $O(\log N)$, as opposed to visiting $O(N)$ particles in naïve algorithm.
- Huge win when $N \approx 10^{10}$, for example (c.f. week 1).



Barnes-Hut version of my "Galaxy" program, with 100,000 stars.

Scope for Recursive Parallelization?

- *Idea*: can we exploit the kind of recursive parallelization approaches discussed in this lecture to parallelize Barnes-Hut?
- I will make a sequential Java version of B-H available on Moodle.
- The core calcForce() method on the Node class has a recursive structure similar to other examples discussed in this lecture – maybe can use Java Fork/Join or similar to parallelize?

Summary

- Reviewed Cilk, one of the more elegant and efficient systems for shared memory parallel programming in C.
- Especially suited for *divide and conquer parallelism* (while OpenMP is arguably suitable for *data parallelism*).
- With emergence of multicore, Cilk has been commercialized, and its ideas continue to influence other products and systems.
- Also briefly reviewed products influenced by Cilk – Intel TTB, and Java Fork/Join.

Further Reading

- *Cilk 5.4.6 Reference Manual*, Supercomputing Technologies Group, MIT Laboratory for Computer Science, http://supertech.csail.mit.edu/cilk/
- Doug Lea, *A Java Fork/Join Framework*, ACM 2000 Java Grande Conference, http://gee.cs.oswego.edu/dl/papers/fj.pdf
- JDK 1.7 Fork/Join Framework, http://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html