COMP 422, Lecture 9: OpenMP 3.0 tasks, Introduction to Intel Thread Building Blocks

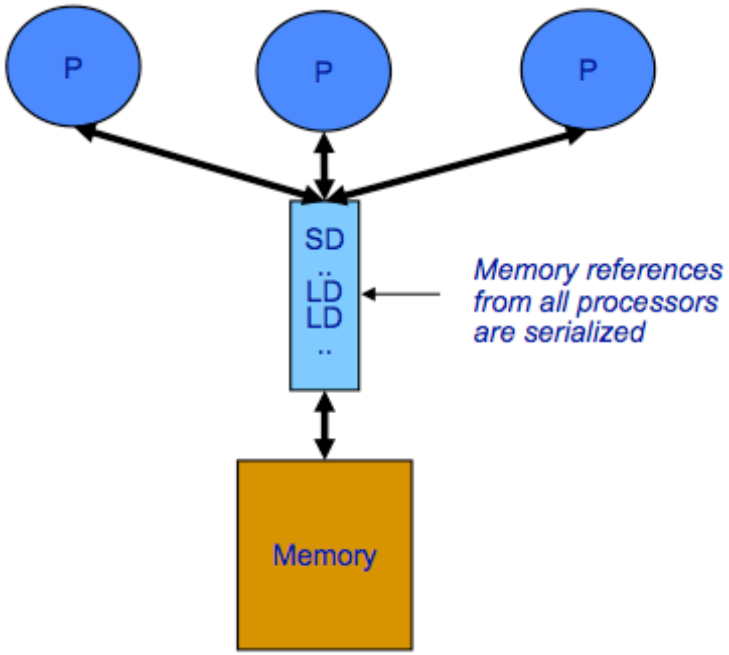
Vivek Sarkar

Department of Computer Science
Rice University

vsarkar@rice.edu

Recap of Lecture 8 (Memory Consistency Models, OpenMP flush)

Sequential Consistency



[Lamport] "*A multiprocessor system is sequentially consistent if the result of any execution is the same as if the operations of all processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by the program*"

Distinguishing Data vs. Synchronization Memory Operations

- Option 1: Annotations at statement level
  - P1
  - data = ON
    A = 23;
    B = 37;
  - synchronization = ON
    Flag = 1;

```
P2
synchronization = ON
    while (Flag != 1) {;}
data = ON
    … = B;
    … = A;
```

- Option 2: Declarations at variable level
  - synch int: Flag
  - data int: A, B
- Option 3: Treat flush/barrier operations as synchronization statements
  - synchronization = ON
  - flush
  - data = ON
  - . . .

Flush Is the Key OpenMP Operation

#pragma omp flush [(list)]

- **Prevents re-ordering of memory accesses across flush**
- **Allows for overlapping computation with communication**
- **A flush** construct with a list applies the flush operation to the items in the list,and does not return until the operation is complete for all specified list items.
  - —If a pointer is present in the list, the pointer itself is flushed, not the object to which the pointer refers
- A **flush** construct without a list, executed on a given thread, operates as if the whole thread-visible data state of the program, as defined by the base language, is flushed.

Implicit flushes

- In barriers
- At entry to and exit from
  - Parallel, parallel worksharing, critical, ordered regions
- At exit from worksharing regions (unless `nowait` is specified)
- In `omp_set_lock`, `omp_set_nest_lock`, `omp_set_nest_lock`, `omp_unset_nest_lock`
- In `omp_test_lock`, `omp_test_nest_lock`, if lock is acquired
- At entry to and exit from `atomic` - flush-set is limited to variable being atomically updated

Question: can threads 1 and 2 enter their critical sections simultaneously in this example?

```
a = b = 0
```

thread 1

```
b = 1
flush(b)
flush(a)
if (a == 0) then
    critical section
end if
```

thread 2

```
a = 1
flush(a)
flush(b)
if (b == 0) then
    critical section
end if
```

Importance of variable list in flush

Incorrect example:

```
a = b = 0
```

thread 1

```
b = 1
flush(b)
flush(a)
if (a == 0) then
   critical section
end if
```

thread 2

```
a = 1
flush(a)
flush(b)
if (b == 0) then
   critical section
end if
```

Correct example:

```
a = b = 0
```

thread 1

```
b = 1
flush(a,b)
if (a == 0) then
   critical section
end if
```

thread 2

```
a = 1
flush(a,b)
if (b == 0) then
   critical section
end if
```

Acknowledgments for today's lecture

- **"Towards OpenMP 3.0", Larry Meadows, HPCC 2007 presentation**
  - **http://www.tlc2.uh.edu/hpcc07/Schedule/speakers/hpcc07_Larry.ppt**
- **OpenMP 3.0 draft specification**
  - **http://www.openmp.org/mp-documents/spec30_draft.pdf**
- **Thread Building Blocks, Arch Robinson, HPCC 2007 tutorial**
  - **http://www.tlc2.uh.edu/hpcc07/Schedule/tbBlocks**
- **Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism",** James Reinders, O'Reily, **First Edition, July 2007**
  - **http://www.oreilly.com/catalog/9780596514808/**

Outline

- Cilk_fence() operation
- OpenMP 3.0 tasks
- Introduction to Thread Building Blocks for C++ programs

Cilk_fence()

- Cilk_fence() ensures that all memory operations of a processor are committed before next instruction is executed.
- Programmers may also synchronize through memory using lock-free protocols, although Cilk is agnostic on consistency model.
  - If a program contains no data races, Cilk effectively supports sequential consistency.
    - **Ordering of memory operations is established by ancestor-descendant relationships or by Cilk_fence() operations**
  - If a program contains data races, Cilk's behavior depends on the consistency model of the underlying hardware.

To aid portability, the `Cilk_fence()` function implements a memory barrier on machines with weak memory models.

Example: Can z = 2?

```
int x = 0, y = 0, z = 0;

cilk void foo()
{
   x = 1;
   if (y==0) z++;
}

cilk void bar()
{
   y = 1;
   if (x==0) z++;
}

cilk int main ()
{
   spawn foo();
   spawn bar();
   sync;
   printf("z = %d\n", z);
   return 0;
}
```

Figure 2.8: An example of the subtleties of memory consistency.

Inserting Cilk_fence() operations

```
int x = 0, y = 0, z = 0;

cilk void foo()
{
   x = 1; Cilk_fence();
   if (y==0) z++;
}

cilk void bar()
{
   y = 1; Cilk_fence();
   if (x==0) z++;
}

cilk int main ()
{
   spawn foo();
   spawn bar();
   sync;
   printf("z = %d\n", z);
   return 0;
}
```

Figure 2.8: An example of the subtleties of memory consistency.

Outline

- Cilk_fence() operation
- OpenMP 3.0 tasks
- Introduction to Thread Building Blocks for C++ programs

General task characteristics in OpenMP 3.0

- 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

Definitions

- *Task construct* – `task` directive plus structured block
- *Task* – the package of code and instructions for allocating data created when a thread encounters a task construct
- *Task region* – the dynamic sequence of instructions produced by the execution of a task by a thread

`task` Construct ~ Cilk's spawn

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

where *clause* can be one of:

`if (`*`expression`***`)`**
`untied`
`shared (`*`list`***`)`**
`private (`*`list`***`)`**
`firstprivate (`*`list`***`)`**
`default( shared | none )`

The `if` clause

- When the `if` clause argument is false
  - —The task is executed immediately by the encountering thread.
  - —The data environment is still local to the new task...
  - —...and it's still a different task with respect to synchronization.

- It's a user directed optimization
  - —when the cost of deferring the task is too great compared to the cost of executing the task code
  - —to control cache and memory affinity

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**

- **At task barriers**
  - **—applies only to child tasks generated in the current task, not to "descendants"**
  - `#pragma omp taskwait`
    - taskwait is like Cilk's sync

Example – parallel pointer chasing using tasks

```
#pragma omp parallel
{
  #pragma omp single private(p)
   {
    p = listhead ;
    while (p) {
       #pragma omp task
                process (p)
       p=next (p) ;
     }
   }
}
```

Spawn call to process(p)

Implicit taskwait

Example – parallel pointer chasing on multiple lists using tasks (nested parallelism)

```
#pragma omp parallel
{
   #pragma omp for private(p)
   for ( int i =0; i <numlists ; i++) {
       p = listheads [ i ] ;
       while (p ) {
       #pragma omp task
           process (p)
       p=next (p ) ;
       }
   }
}
```

Example: postorder tree traversal

```
void postorder(node *p) {
    if (p->left)
      #pragma omp task
        postorder(p->left);
    if (p->right)
      #pragma omp task
        postorder(p->right);
#pragma omp taskwait // wait for child tasks
  process(p->data);
}
```

- **Parent task suspended until children tasks complete**

Task switching

- **Certain constructs have task scheduling points at defined locations within them**
- **When a thread encounters a task scheduling point, it is allowed to suspend the current task and execute another (called *task switching*)**
- **It can then return to the original task and resume**

Task switching example

```
#pragma omp single
{
  for (i=0; i<ONEZILLION; i++)
    #pragma omp task
      process(item[i]);
}
```

- Too many tasks generated in an eye-blink
- Generating task will have to suspend for a while
- With task switching, the executing thread can:
  - —execute an already generated task (draining the "*task pool*")
  - —dive into the encountered task (could be very cache-friendly)

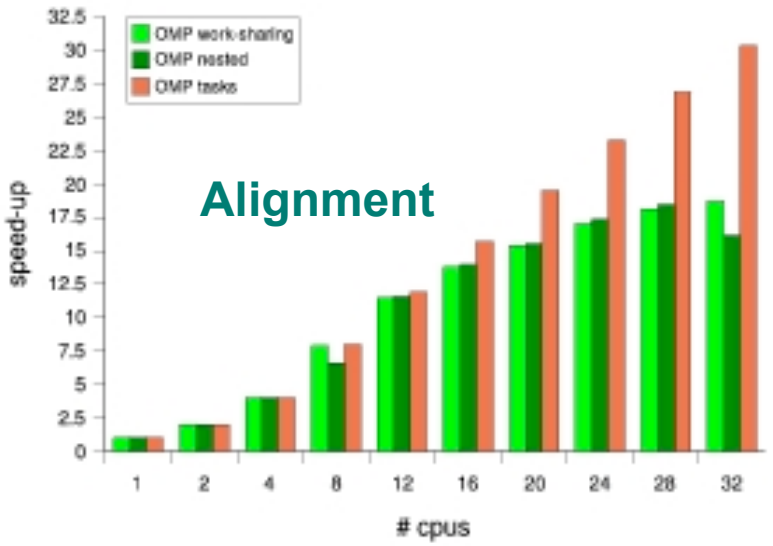
Thread switching

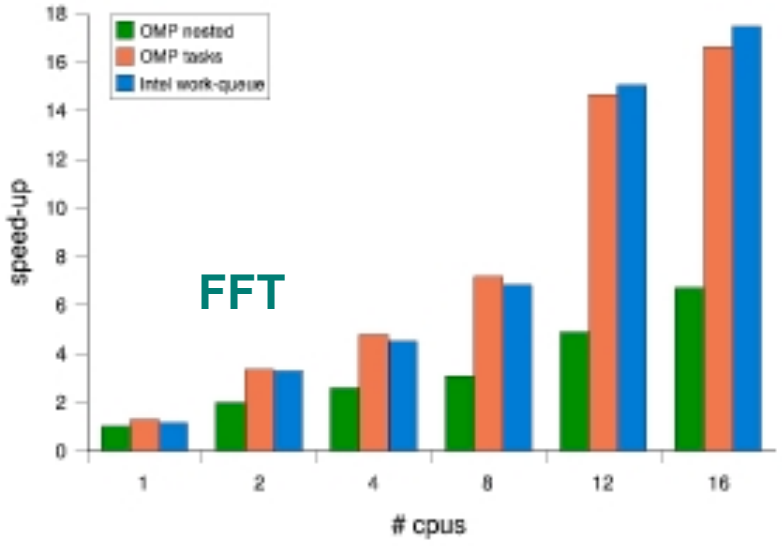
```
#pragma omp single
{
  #pragma omp task      untied
    for (i=0; i<ONEZILLION; i++)
      #pragma omp task
        process(item[i]);
}
```

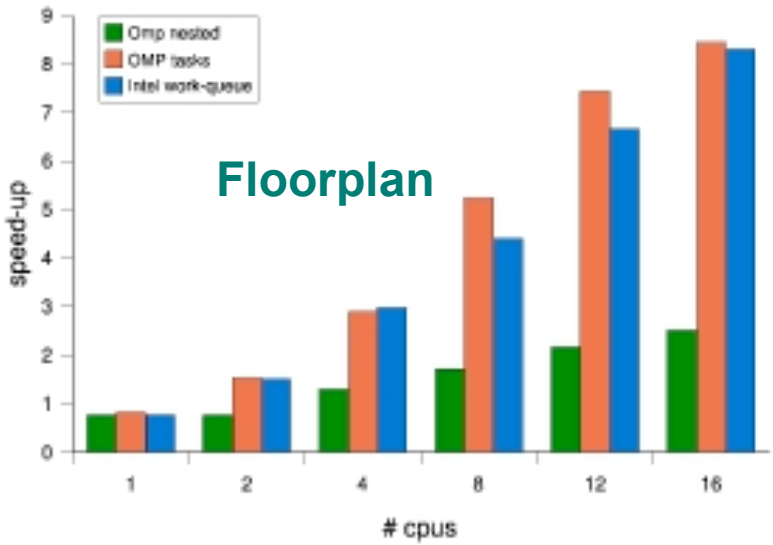
- Eventually, too many tasks are generated
- Generating task is suspended and executing thread switches to a long and boring task
- Other threads get rid of all already generated tasks, and start starving…

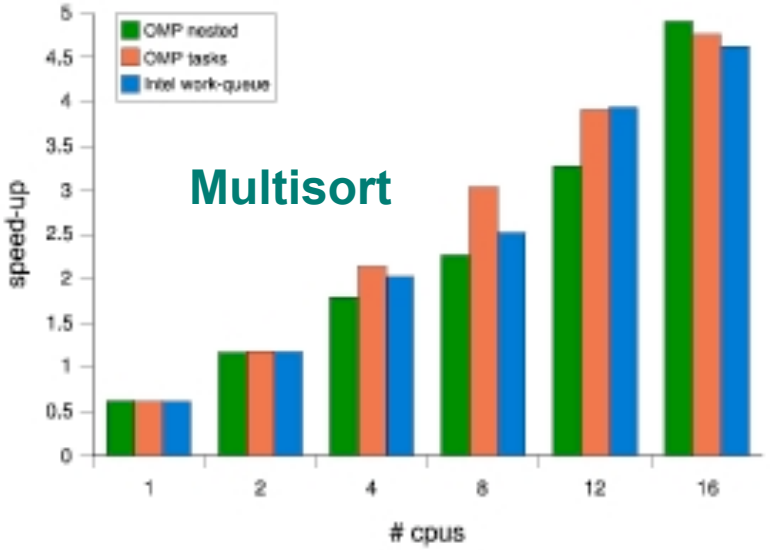
- With thread switching, the generating task can be resumed by a different thread, and starvation is over
- Too strange to be the default: the programmer is responsible!

Performance Results 1



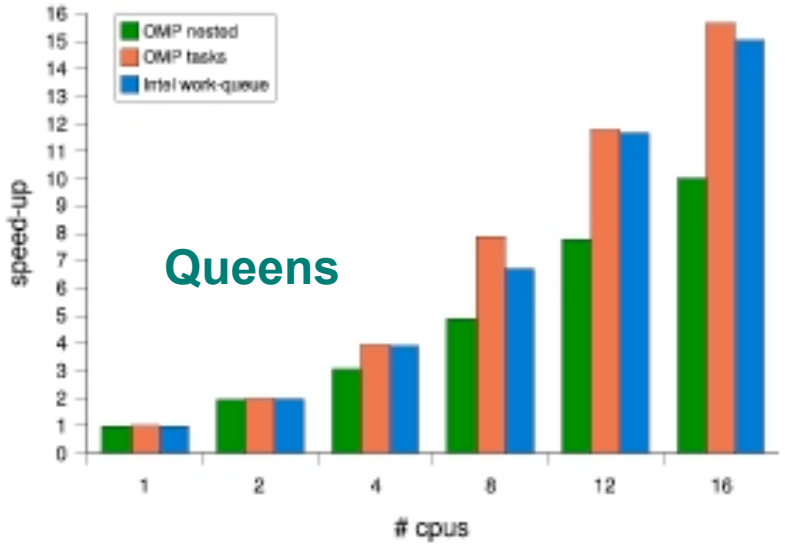


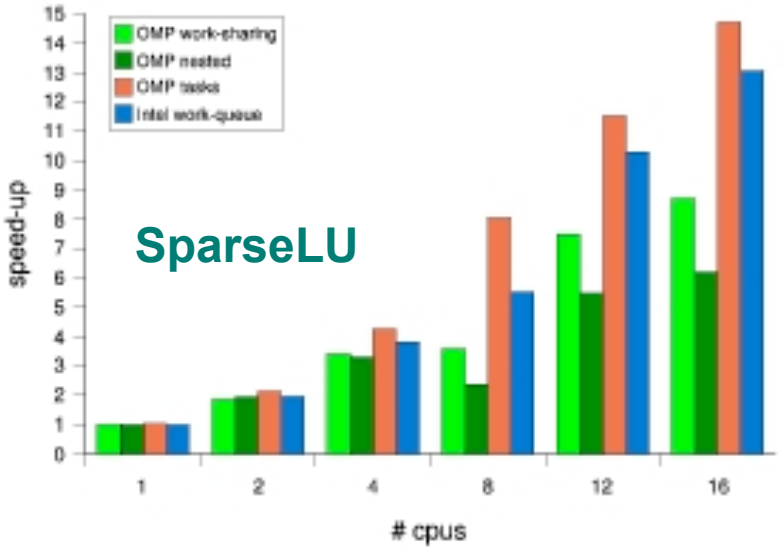


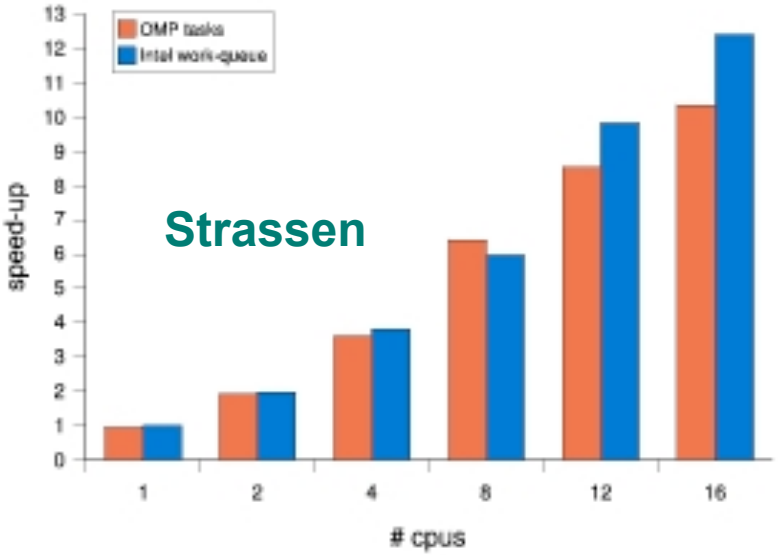


All tests run on SGI Altix 4700 with 128 processors

Performance Results 2







All tests run on SGI Altix 4700 with 128 processors

Summary: Tasks and OpenMP

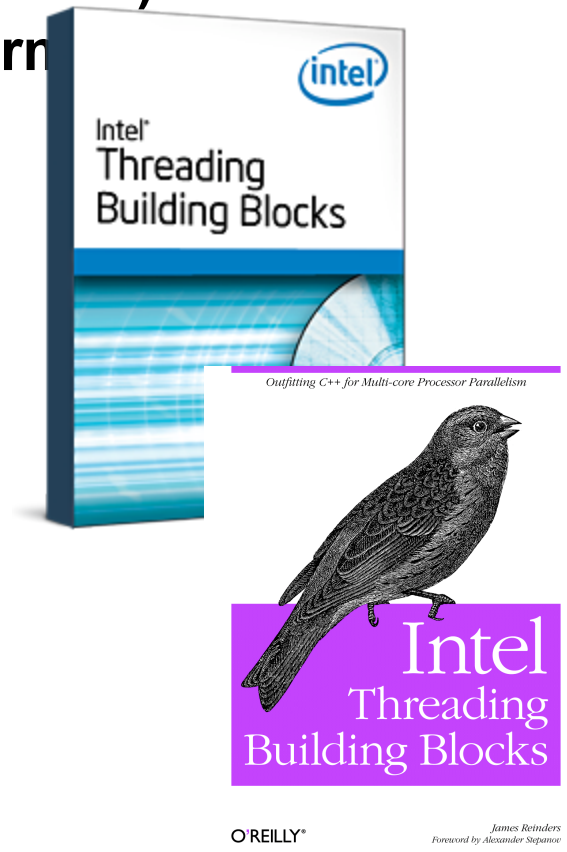
- Tasks have been fully integrated into OpenMP
- Key concept: OpenMP has always had tasks, we just never called them that.
  - Thread encountering `parallel` construct packages up a set of *implicit* tasks, one per thread.
  - Team of threads is created.
  - Each thread in team is assigned to one of 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!

Outline

- Cilk_fence() operation
- OpenMP 3.0 tasks
- Introduction to Thread Building Blocks for C++ programs

Thread Building Blocks (TBB) Overview

- **Intel® Threading Building Blocks (Intel® TBB) is a C++ library that simplifies threading for perform**



•Move the level at which you program from threads to tasks

•Let the run-time library worry about how many threads to use, scheduling, cache etc.

•Committed to:
- compiler independence
- processor independence
- OS independence

•GPL license allows use on many platforms; commercial license allows use in products

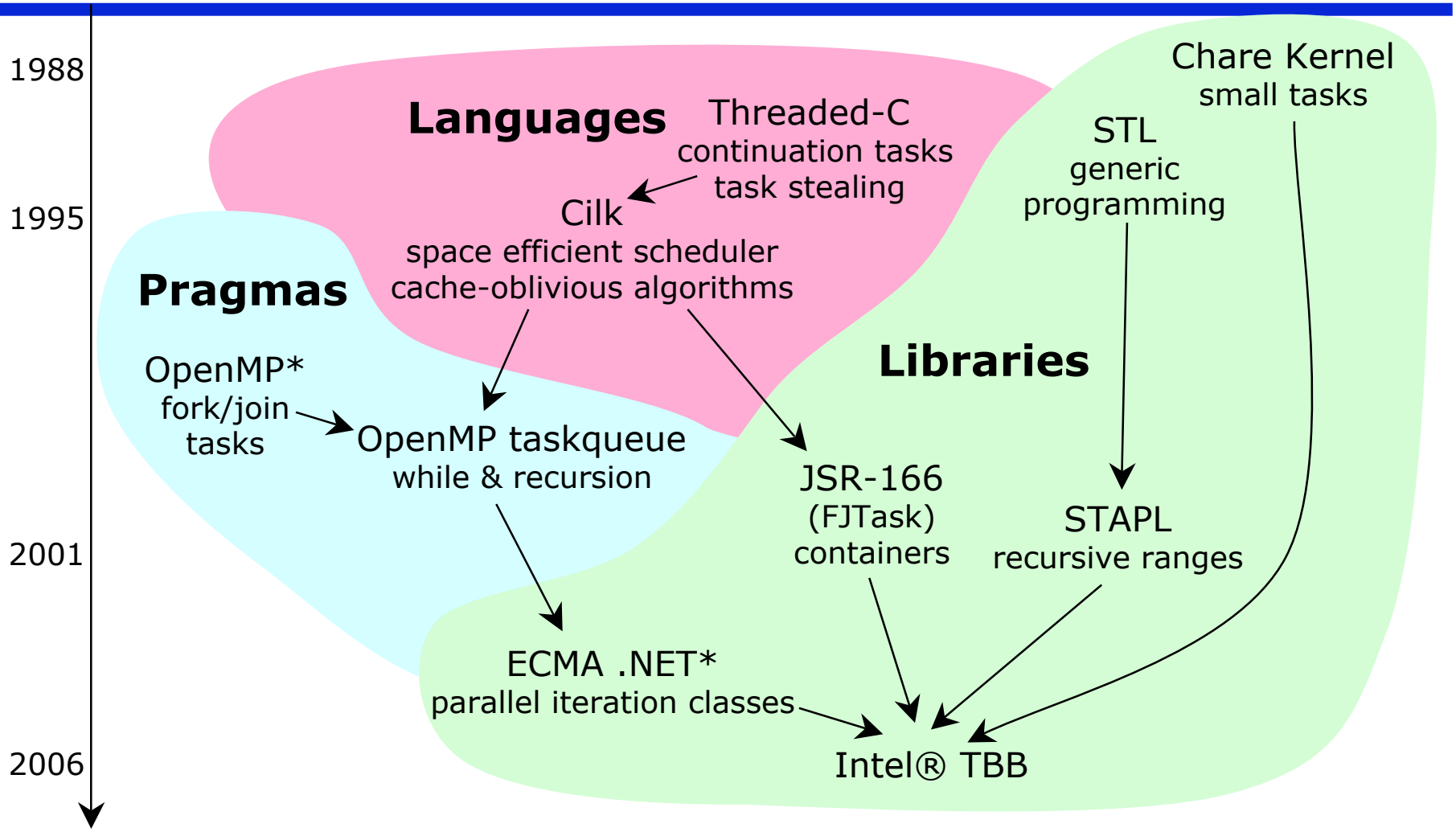
Three Approaches to Parallelism

- **New language**
  - —**Cilk, NESL, Fortress, X10, Chapel, …**
  - —**Clean, conceptually simple**
  - —But **very difficult to get widespread acceptance**
- **Language extensions / pragmas**
  - —**OpenMP, HPF**
  - —**Easier to get acceptance**
  - —But **still require a special compiler or pre-processor**
- **Library**
  - —**POOMA, Hood, MPI, ...**
  - —**Works in existing environment, no new compiler needed**
  - —But **Somewhat awkward**
    - – **Syntactic boilerplate**
    - – **Cannot rely on advanced compiler transforms for performance**

Generic Programming

- Best known example is C++ Standard Template Library (STL)
- Enables distribution of broadly-useful high-quality algorithms and data structures
- Write best possible algorithm with fewest constraints
  - —Do not force particular data structure on user
  - —Classic example: STL std::sort
- Instantiate algorithm to specific situation
  - —C++ template instantiation, partial specialization, and inlining make resulting code efficient

Family Tree



*Other names and brands may be claimed as the property of others

Scalability

- Ideally you want Performance $\propto$ Number of hardware threads
- Impediments to scalability
  - Any code which executes once for each thread (e.g. a loop starting threads)
  - Coding for a fixed number of threads (can't exploit extra hardware; oversubscribes less hardware)
  - Contention for shared data (locks cause serialization)
- TBB approach
  - Create tasks recursively (for a tree this is logarithmic in number of tasks)
  - Deal with tasks not threads. Let the runtime (which knows about the hardware on which it is running) deal with threads.
  - Try to use partial ordering of tasks to avoid the need for locks.

    Provide efficient atomic operations and locks if you really need them.

Key Features of Intel® Threading Building Blocks

- You specify *task patterns* instead of threads (focus on the work, not the workers)
  - Library maps user-defined logical tasks onto physical threads, efficiently using cache and balancing load
  - Full support for *nested parallelism*
- Targets threading for *robust performance*
  - Designed to provide portable scalable performance for computationally intense portions of shrink-wrapped applications.
- *Compatible* with other threading packages
  - Designed for CPU bound computation, not I/O bound or real-time.
  - Library can be used in concert with other threading packages such as native threads and OpenMP.
- Emphasizes *scalable, data parallel* programming
  - Solutions based on functional decomposition usually do not scale.

TBB Components

Generic Parallel Algorithms
parallel_for
parallel_reduce
pipeline
parallel_sort
parallel_while
parallel_scan

Concurrent Containers
concurrent_hash_map
concurrent_queue
concurrent_vector

Task scheduler

Low-Level Synchronization Primitives
atomic
mutex
spin_mutex
queuing_mutex
spin_rw_mutex
queuing_rw_mutex

Memory Allocation
cache_aligned_allocator
scalable_allocator

Timing
tick_count

Serial Example

```
static void SerialApplyFoo( float a[], size_t n ) {
    for( size_t i=0; i!=n; ++i )
        Foo(a[i]);
}
```

Will parallelize by dividing iteration space of i into chunks

Parallel Version

red = provided by TBB

```
class ApplyFoo {
    float *const my_a;
public:
    ApplyFoo( float *a ) : my_a(a) {}
    void operator()( const blocked_range<size_t>& range ) const {
        float *a = my_a;
        for( int i= range.begin(); i!=range.end(); ++i )
            Foo(a[i]);
    }
};

void ParallelApplyFoo(float a[], size_t n ) {
    parallel_for( blocked_range<int>( 0, n ),
                  ApplyFoo(a),
                  auto_partitioner() );
}
```

Task

Pattern

Iteration space

Automatic grain size

```
template <typename Range, typename Body, typename Partitioner>
void parallel_for(const Range& range,
                  const Body& body,
                  const Partitioner& partitioner);
```

Requirements for Body

Body::Body(const Body&)	**Copy constructor**
Body::~Body()	**Destructor**
void Body::operator() (Range& *subrange*) const	**Apply the body to *subrange*.**

- **parallel_for schedules tasks to operate in parallel on subranges of the original, using available threads so that:**
  - **—Loads are balanced across the available processors**
  - **—Available cache is used efficiently**
  - **—Adding more processors improves performance of existing code (without recompilation!)**

Range is Generic

- Requirements for parallel_for Range

R::R (const R&)	Copy constructor
R::~R()	Destructor
bool R::empty() const	True if range is empty
bool R::is_divisible() const	True if range can be partitioned
R::R (R& r, split)	Split r into two subranges

- Library provides blocked_range and blocked_range2d
- You can define your own ranges
- Partitioner calls splitting constructor to spread tasks over range
- Puzzle: Write parallel quicksort using parallel_for, without recursion! (One solution is in the TBB book)

How this works on `blocked_range2d`