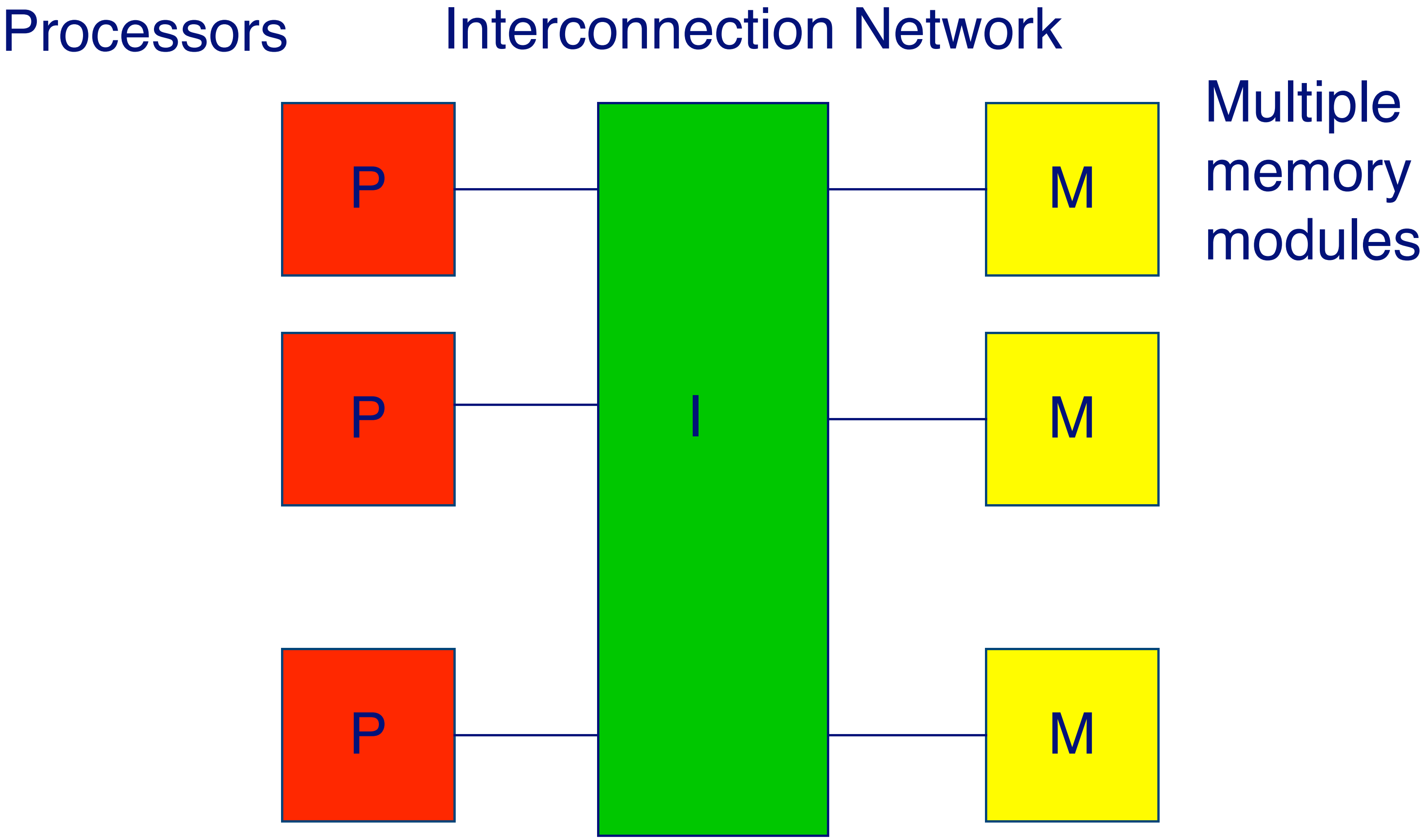
Shared-Memory Parallel Programming

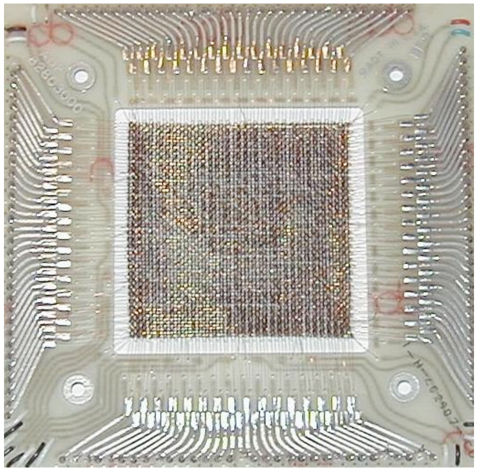
Typical Shared-Memory Architecture (“Dance-hall Configuration”)



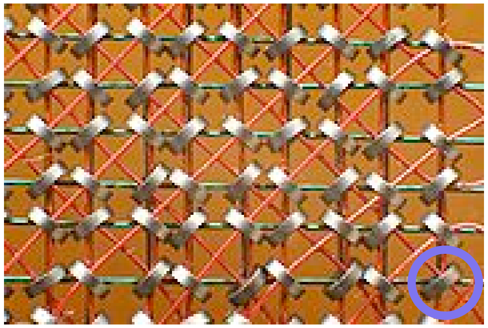
Shared-Memory Architecture also known as

- SMP (Symmetric Multiprocessor) since the view looks the same from all processors.
- UMA (Uniform Memory Access)
- and more recently “Multi-Core”

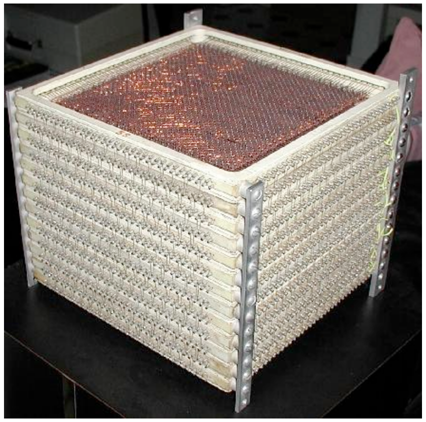
The original “Core”: Core Memory



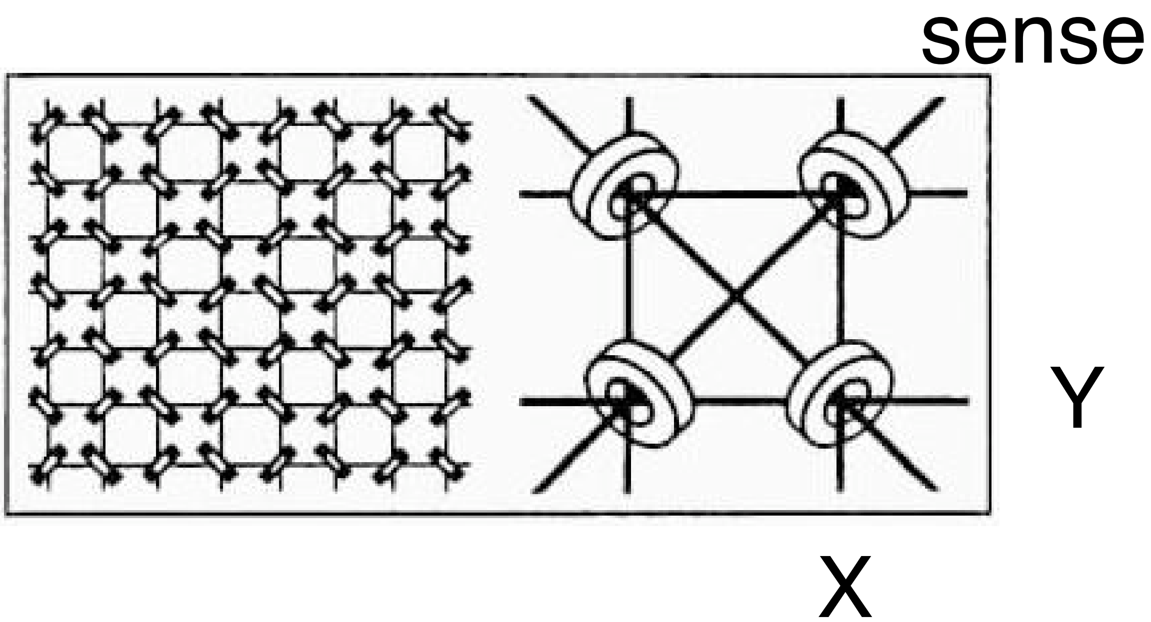
5x5 inches actual size



← 1 Core



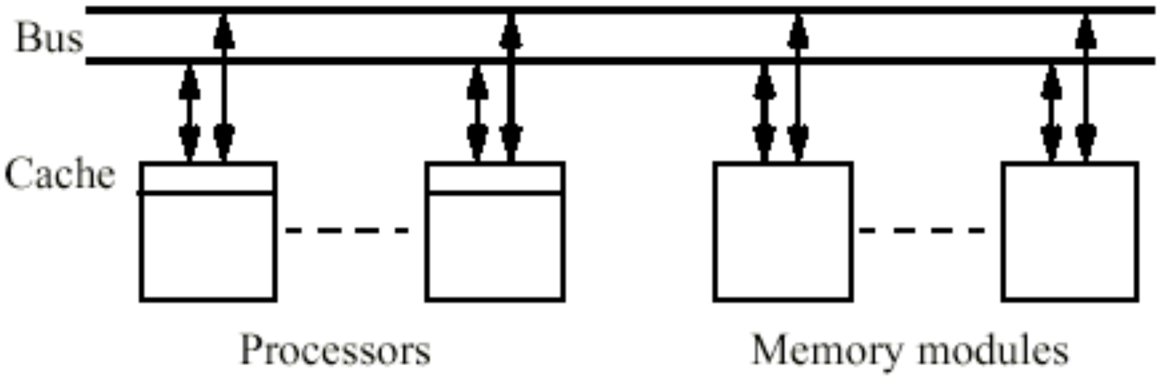
8x8x8 inches actual size



Recall Advantages & Disadvantages of Shared-Memory

- + All data in **one address space**; don’t have to worry about distributing
- – Not scalable, since interconnection network will either
  - saturate or
  - latency will increase

Bus Architecture with Caching



NUMA

- Technically, an architecture with caches is like a Non-Uniform Memory Access machine (local=cache, vs. global=memory).
- However, this is usually reserved for the case where the local/global distinction is **programmed** for explicitly.

Cache Coherency Problem

- Caches keep local-to-processor copies of data in the shared memory
- If a processor modifies cached data, the data in the shared memory is no longer valid
- Worse, copies of the data in other processors' caches is invalid

Concepts for Cache Coherency

- **Invalidation**:
  - Each cache line (group of words) has a validity bit.
  - If a processor writes a word in cache, other processors' caches are checked to see if the corresponding line is present (called "snooping").
  - If the line is present, it is marked invalid.
  - The line will need to be re-fetched from memory if needed.

Concepts for Cache Coherency

- The alternative to snooping is to **broadcast** the written word to all processors.
- **Broadcast** is expensive because it uses bandwidth even if the processor does not have the line cached.
- **Directory-based** system is another approach: The directory knows where all copies are; sends updates selectively.

Write-Through, Write-Back, etc.

- **Write-through**: When a new value is written to a cached word, the value is immediately written to memory as well.
- **Write-back**: When a new value is written to a cached word, the value is not written to memory until the cache line is replaced with some other set of words.
- **No-Write**: Only reads are cached

Tradeoffs?

- **Write-through**: Memory is always up-to-date.
- **Write-back**: Less traffic writing stuff to memory (that might not be used between writes).
- **No-Write**: Typically most accesses are reads, so this achieves performance with simplicity.

MESI States for Cache Lines

- **Exclusive Modified** (M): not shared by other caches and contains modified information, i.e. main memory does not contain the current value.
- **Exclusive Unmodified** (E): not shared and was not modified.
- **Shared Unmodified** (S): unmodified and present in other caches.
- **Invalid** (I): invalid as other caches or main memory contain a modified version.

Effect on Cache Miss for Line

- Prior to reading new data:
  - **Exclusive Modified** (M): Data must be written (if not already written back)
  - **Exclusive Unmodified** (E): NOOP
  - **Shared Unmodified** (S): NOOP
  - **Invalid** (I): NOOP

Write Buffers

- To avoid blocking the processor while a write-back is taking place, a write-buffer can be employed.

- Care must be taken that memory fetches don’t occur, meanwhile, from lines that are also in the write buffer on their way back to memory.

Semantics

- Cache coherency protocols, etc. try to preserve a semantics of memory access.
- Typically they want single loads and stores to look “atomic” or “indivisible” so that the programming model is as near as possible to a theoretical MIMD ideal.

Locking, Critical Sections

- It is often necessary to have atomicity at larger grains than single reads and writes.
- Example: attaining exclusive access to some critical data structure.

Two types of locking

- **Busy-waiting**: processor keeps “spinning” while waiting for processor holding the lock to unlock.
- **Non-busy-waiting**: processor blocks, turning over itself to a different process, until the lock is unlocked.
  - Typically entails a *little* busy-waiting just to access the ready-queue and waiting list of processors.

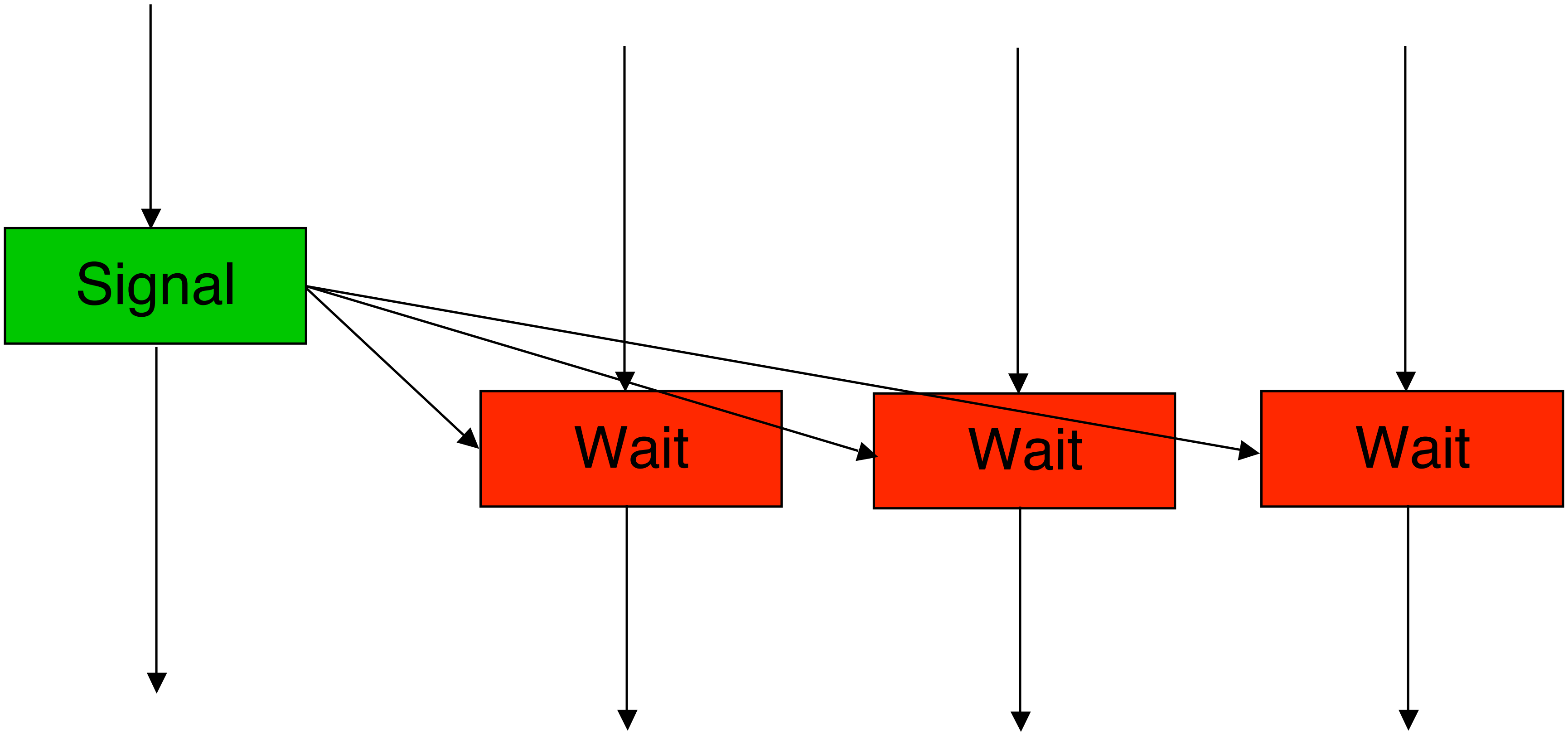
Synchronization in General

- Locking is a form of synchronization
- There are also varieties of locking:
  - Exclusive only
  - Shared-read, Exclusive-write
  - etc.
- Other forms include “signal synchronization”: one process waiting for another, as if the latter were writing data need by the former.

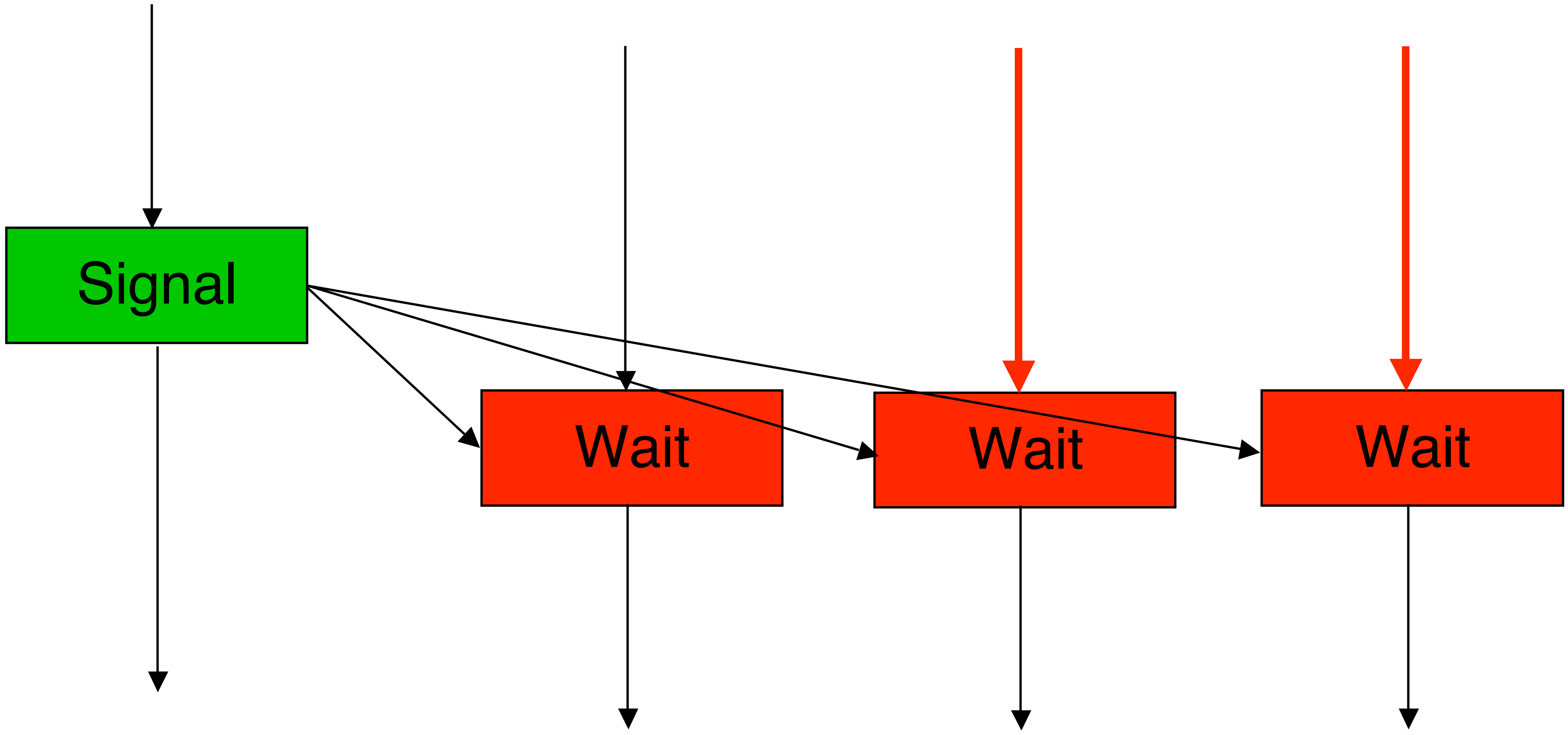
Signal Synchronization

- Different from mutual exclusion: asymmetric
- One-to-one: For each posting of an event, there is one wake-up.
- “Avalanche”: For a single posting of an event, there is an arbitrary number of wakeups (all processes on the queue wakeup).
- Both have their uses.

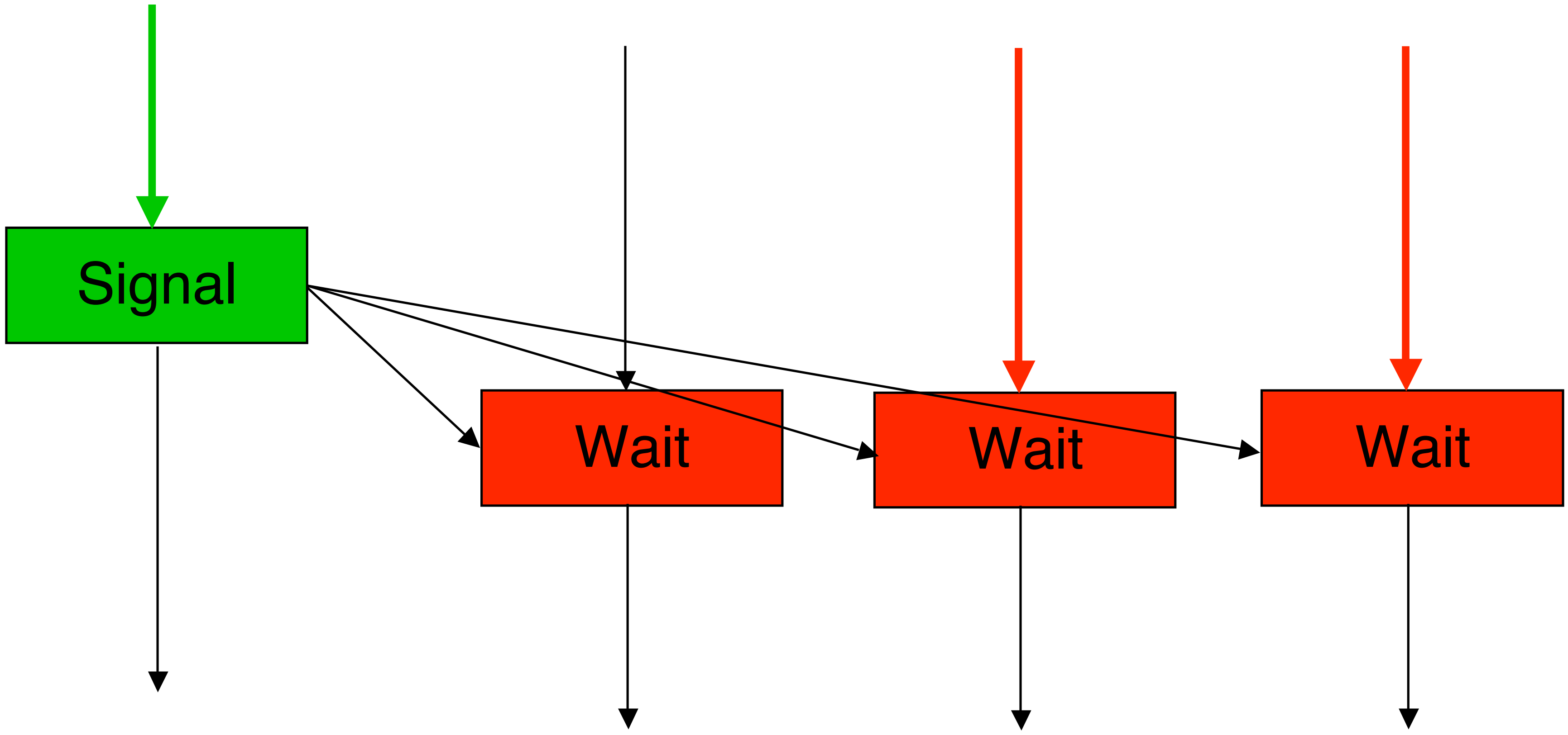
1-1 Signal Synchronization



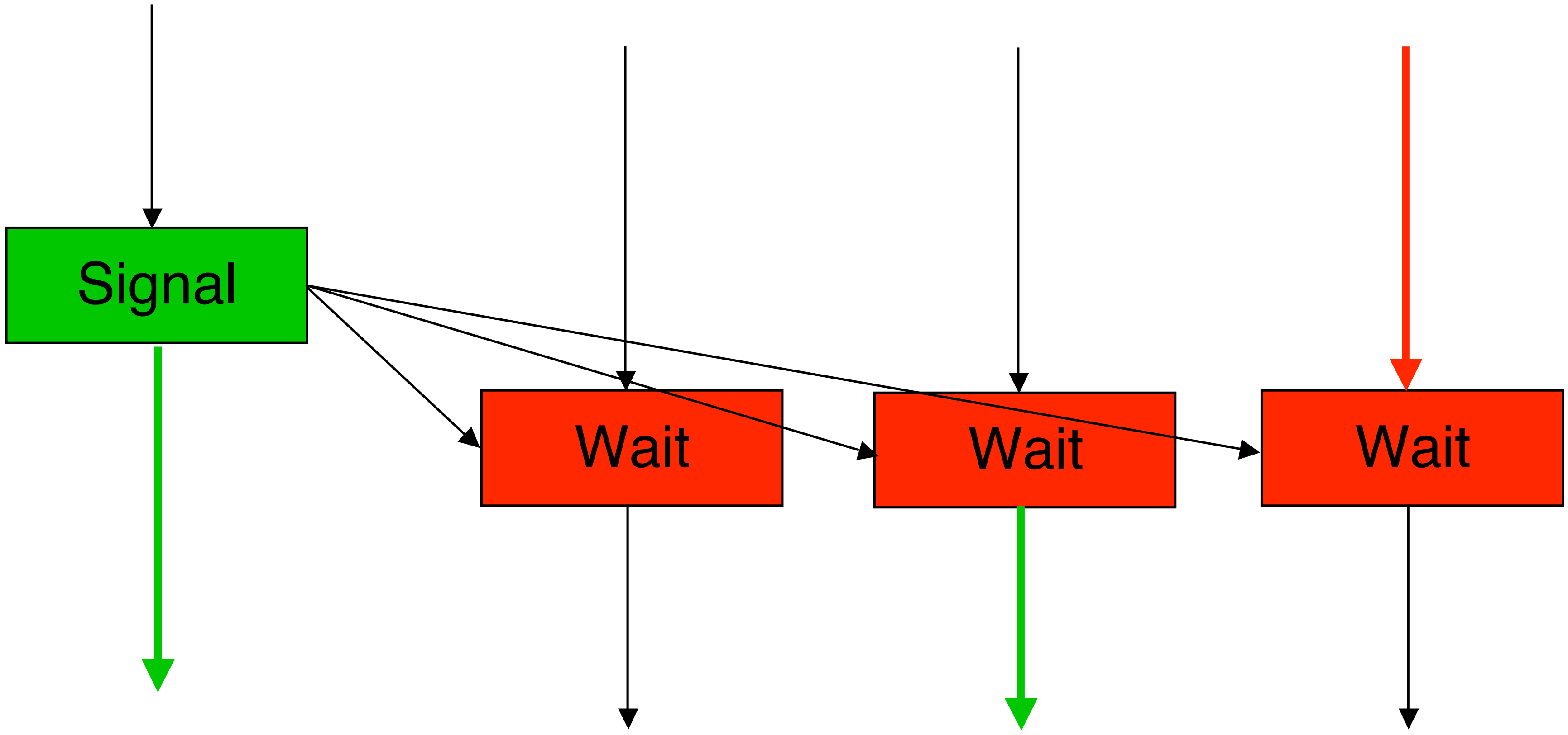
1-1 Signal Synchronization



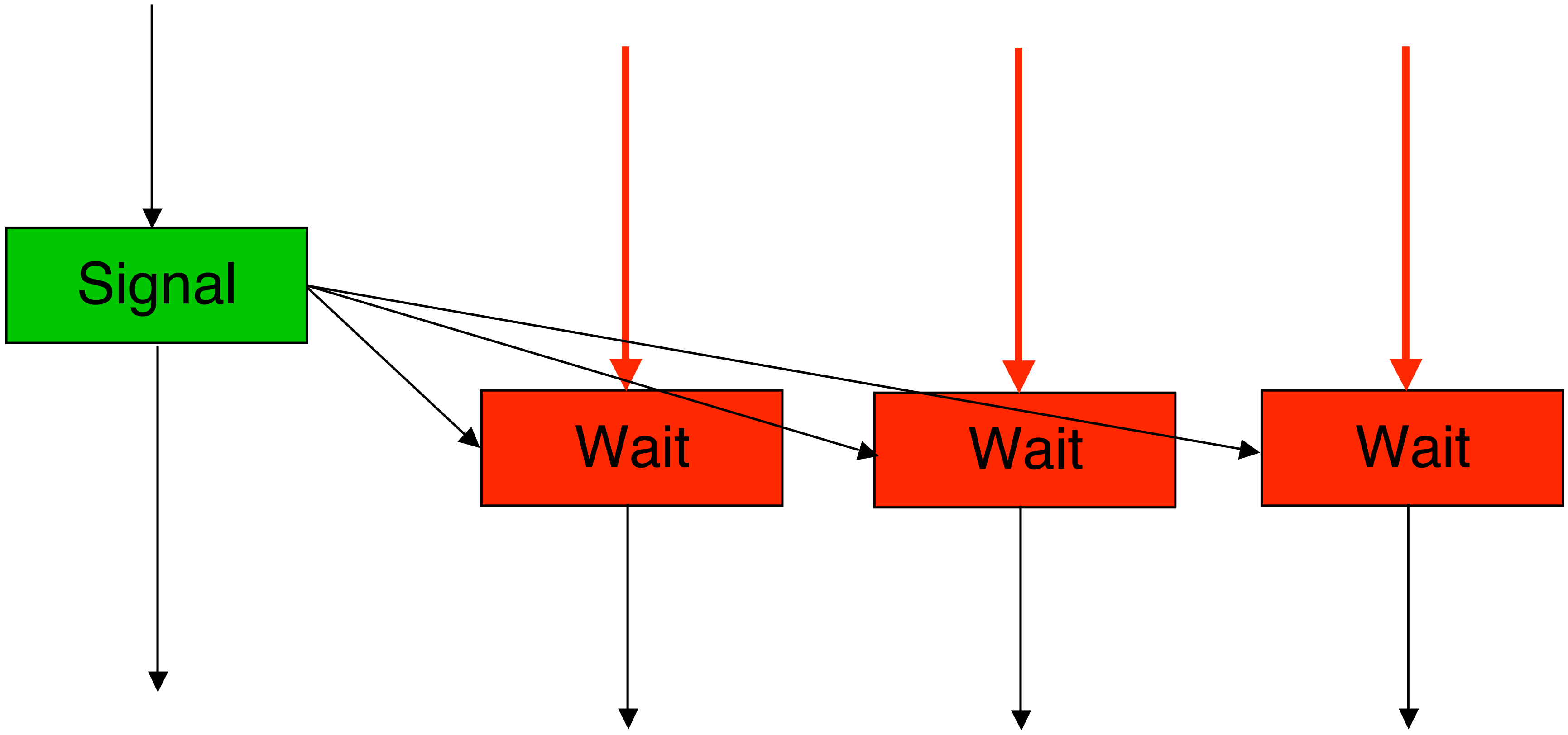
1-1 Signal Synchronization



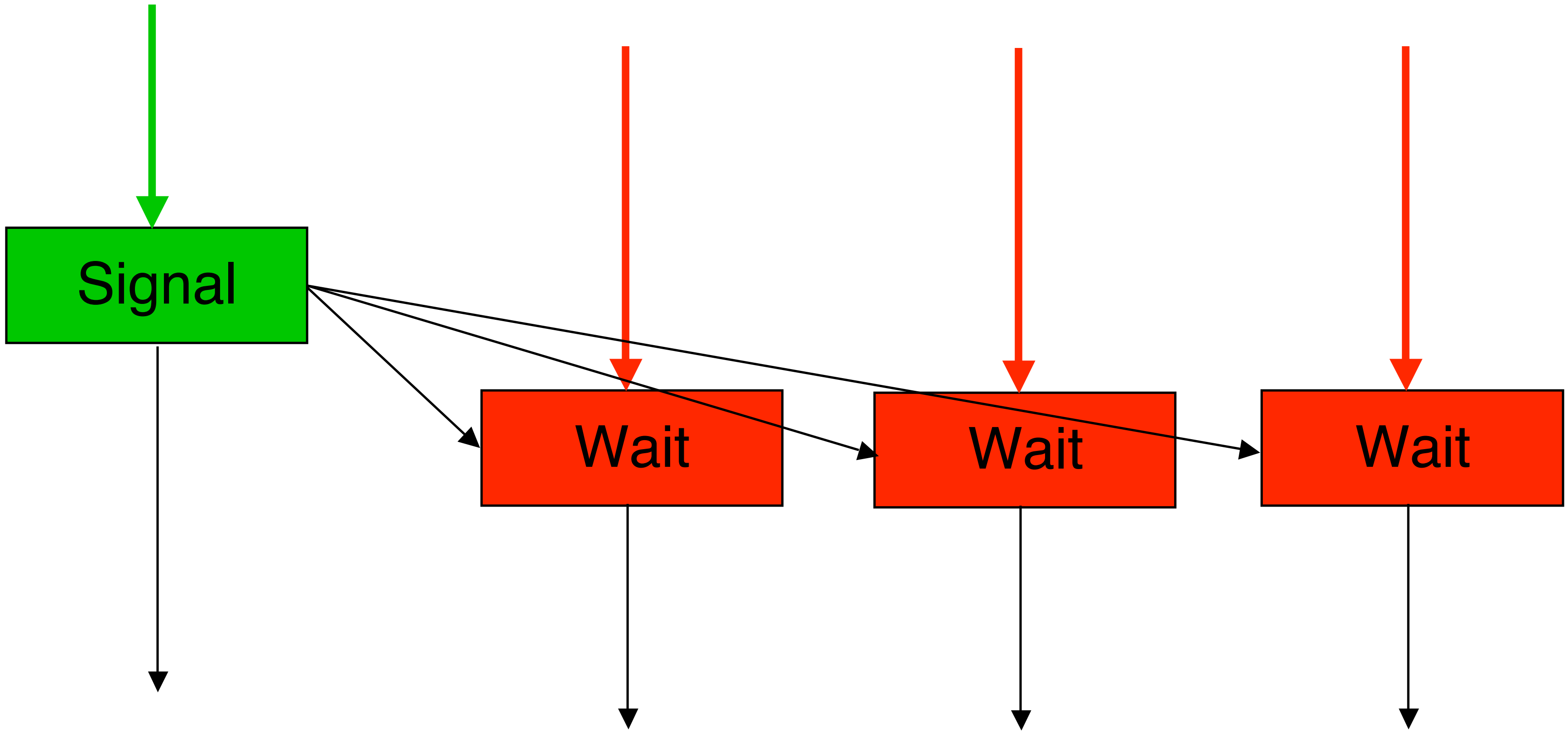
1-1 Signal Synchronization



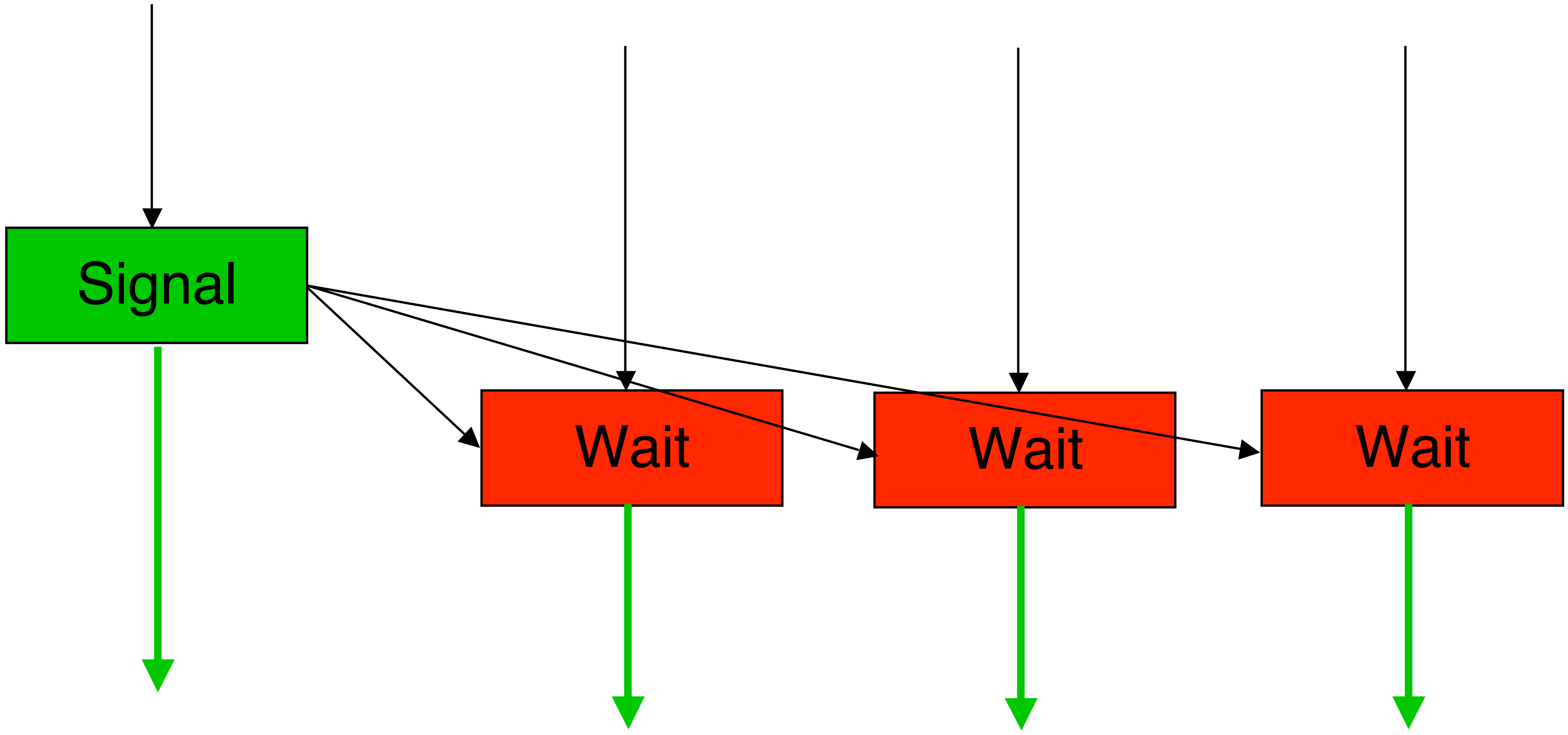
Avalanche Signal Synchronization



Avalanche Signal Synchronization



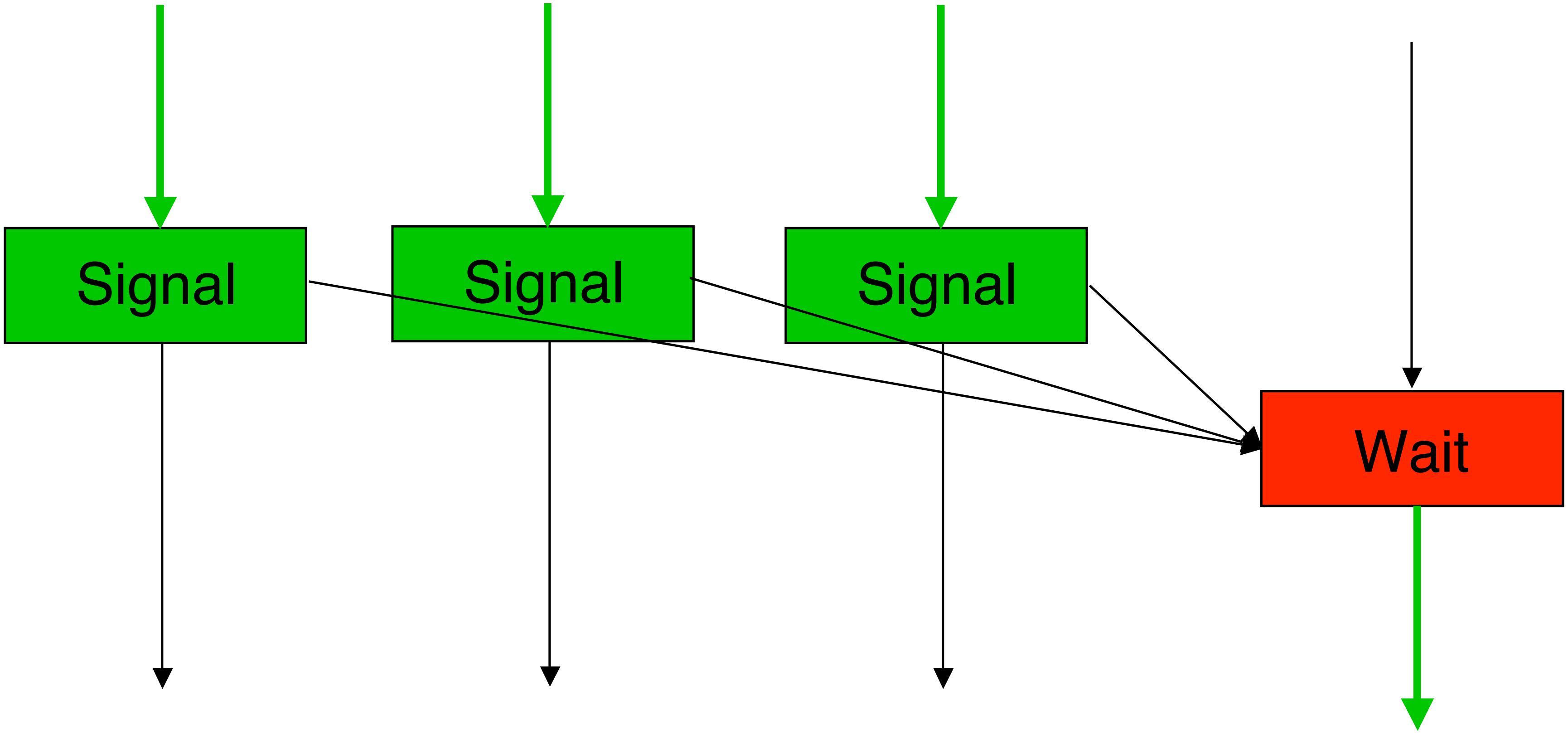
Avalanche Signal Synchronization



Multi-Join Synchronization

- The opposite case of avalanche occurs with **n-way join synchronization**: n processes have to post before the waiting process or processes proceed.
- This occurs in **barriers**, for example.

Multi-Join Synchronization



Threads vs. Processes

- Typically processes connote *heavyweight* things, threads *lightweight* ones
- Processes, e.g. in UNIX, contain much baggage:
  - page table
  - file descriptor table
  - processor state
  - resource tables, etc.

Threads vs. Processes

- Threads concentrate only on the processor state.

- Consequently, threads can be switched much more quickly.

- This provides opportunities of latency-hiding for memory access and i/o.

Threads vs. Processes

- Threads typically share logical memory within a process.
- Processes typically do not share logical memory, except for special shareable segments.
- shmalloc = “shared memory allocate”, kind of an after-thought (companion “shfree”)

shmalloc

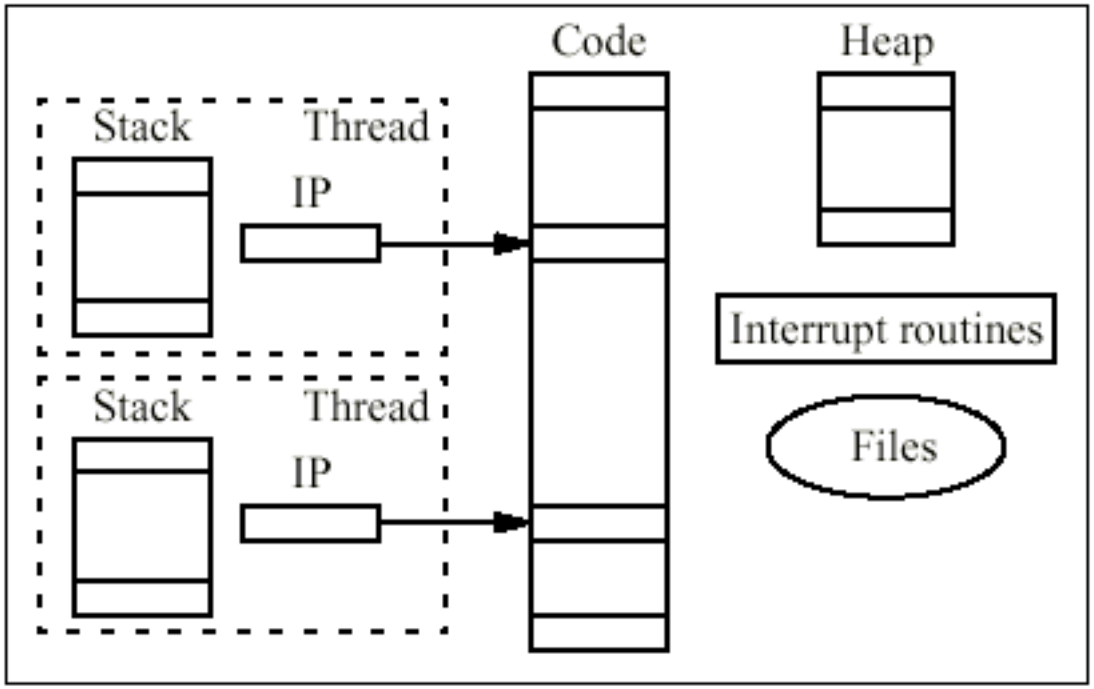
- #include <mpp/shmem.h>
- void *shmalloc(size_t size);
- void shfree(void *ptr);
- void *shrealloc(void *ptr, size_t size);
- void *shmemalign(size_t alignment, size_t size);
- extern long malloc_error;

size_t

The stdlib.h and stddef.h header files define a datatype called **size_t** which is used to represent the size of an object. Library functions that take sizes expect them to be of type size_t, and the sizeof operator evaluates to size_t.

The actual type of size_t is platform-dependent; a common mistake is to assume size_t is the same as unsigned int, which can lead to programming errors, particularly as 64-bit architectures become more prevalent.

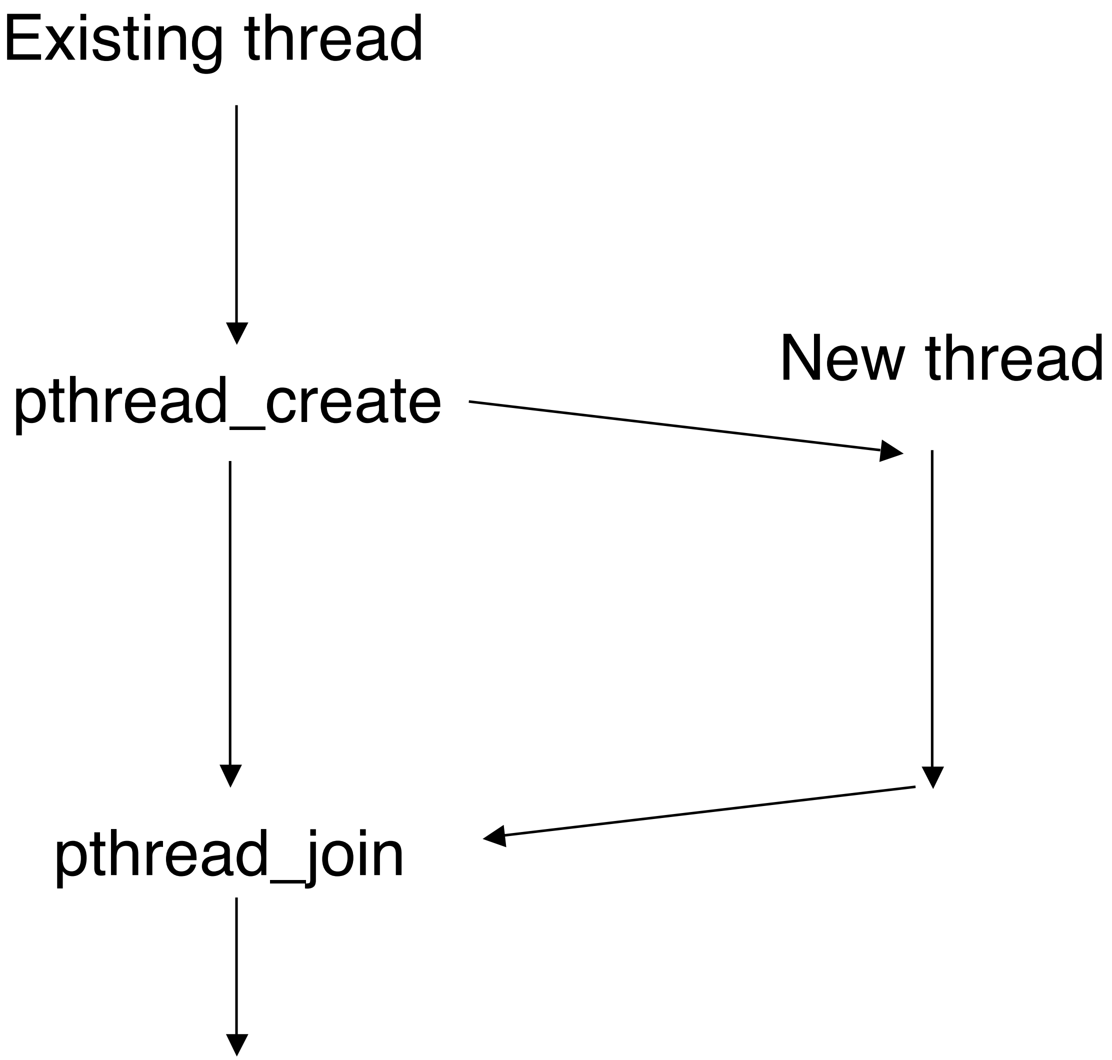
Threads within one Process



Pthreads (Posix Threads)

- Posix = an API standard, for a variety of system aspects (threads, real-time, etc.)
- Posix = “Portable UNIX”

Thread Creation and Joining



Existing thread

pthread_create

New thread

pthread_join

pthread_create and _exit

- **pthread_create**(

  ```
  pthread_create(   pthread_t &tid,                  // thread id
                    NULL,                            // attributes
                    (void*)threadCode(void*),        // code
                    (void*) parameter);              // params
  ```

  creates new pthread running threadCode; parameter is passed to threadCode

- **pthread_exit**((void*) value)

  terminates thread, passing value if joined to another thread

- **pthread_join**(

  ```
  pthread_join(     pthread_t tid,                   // thread id
                    (void**) result);
  ```

  waits for thread tid, result is pointer to value

pthread_exit

- **pthread_exit**((void*) value)
- terminates thread, passing value if joined to another thread
- Note: storage for result must be allocated dynamically or outside of the thread code.

pthread_join

- **pthread_join**(pthread_t tid, // thread id
  (void**) result);

- waits for thread tid, result is that sent by _exit

pthread1.c example

```
struct package
{
char* msg;
};

void* threadCode(void* arg)
  {
  struct package *realArg = arg;
  printf("Hello from %s.\n", realArg->msg);
  pthread_exit(realArg->msg);
  }
```

pthread1.c example

```
struct package
{
char* msg;
};

void* threadCode(void* arg)
  {
  struct package *realArg = arg;
  printf("Hello from %s.\n", realArg->msg);
  pthread_exit(realArg->msg);
  }
```

```
int main(int argc, char** argv)
  {
  struct package arg1, arg2;
  char *result;
  pthread_t tid1, tid2;

  arg1.msg = "thread1";
  arg2.msg = "thread2";

  pthread_create(&tid1, NULL, threadCode, &arg1);
  pthread_create(&tid2, NULL, threadCode, &arg2);

  printf("Hello from main.\n");

  pthread_join(tid1, (void*)&result);
  printf("Thread 1 joined, result is %s.\n", result);

  pthread_join(tid2, (void*)&result);
  printf("Thread 2 joined, result is %s.\n", result);
  }
```

pthread1.c example

```
struct package
{
char* msg;
};

void* threadCode(void* arg)
  {
  struct package *realArg = arg;
  printf("Hello from %s.\n", realArg->msg);
  pthread_exit(realArg->msg);
  }
```

```
int main(int argc, char** argv)
  {
  struct package arg1, arg2;
  char *result;
  pthread_t tid1, tid2;

  arg1.msg = "thread1";
  arg2.msg = "thread2";

  pthread_create(&tid1, NULL, threadCode, &arg1);
  pthread_create(&tid2, NULL, threadCode, &arg2);

  printf("Hello from main.\n");

  pthread_join(tid1, (void*)&result);
  printf("Thread 1 joined, result is %s.\n", result);

  pthread_join(tid2, (void*)&result);
  printf("Thread 2 joined, result is %s.\n", result);
  }
```

output

```
Hello from main.
Hello from thread1.
Hello from thread2.
Thread 1 joined, result is thread1.
Thread 2 joined, result is thread2.
```

Exercise

- Describe how you would implement matrix multiply using pthreads.

Thread Safety

- Some library routines might not be “thread safe”.
- This is typically because they are not “reentrant”, i.e. they assume certain fixed memory locations rather than allocate all of their storage individually.

Thread Locking (non-busy-wait)

```
// global
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, NULL);
…

// in competing threads
pthread_mutex_lock(&mutex);

… critical section …

pthread_mutex_unlock(&mutex);
```

pthread2.c example

```
struct package
{
char* msg;
pthread_mutex_t* mutex;
};


/* Using a mutex below, we should never see the hello and goodbye of two
 * threads interleaved.
 */

void* threadCode(void* arg)
  {
  struct package *realArg = arg;
  pthread_mutex_lock(realArg->mutex);
  printf("Hello from %s.\n", realArg->msg);
  sleep(1);
  printf("Goodbye from %s.\n", realArg->msg);
  pthread_mutex_unlock(realArg->mutex);
  pthread_exit(realArg->msg);
  }
```

pthread2.c example

```
int main(int argc, char** argv)
 {
 pthread_mutex_t mutex1;
 struct package arg1, arg2;
 char *result;
 pthread_t tid1, tid2;

 pthread_mutex_init(&mutex1, NULL);
 arg1.msg = "thread1";
 arg2.msg = "thread2";
 arg1.mutex = &mutex1;
 arg2.mutex = &mutex1;        // one mutex is shared with both threads

 pthread_create(&tid1, NULL, threadCode, &arg1);
 pthread_create(&tid2, NULL, threadCode, &arg2);

 pthread_mutex_lock(&mutex1);
 printf("Hello from main.\n");
 sleep(1);
 printf("Goodbye from main.\n");
 pthread_mutex_unlock(&mutex1);

 pthread_join(tid1, (void*)&result);
 printf("Thread 1 joined, result is %s.\n", result);

 pthread_join(tid2, (void*)&result);
 printf("Thread 2 joined, result is %s.\n", result);
 }
```

output

```
Hello from main.
Goodbye from main.
Hello from thread1.
Goodbye from thread1.
Hello from thread2.
Thread 1 joined, result is thread1.
Goodbye from thread2.
Thread 2 joined, result is thread2.
```

Condition Variables

- Condition variables allow one thread to signal another.

Condition Variables

```
// global
pthread_cond_t cond;
pthread_cond_init(&cond, NULL);
…

// in separate threads
pthread_cond_wait(&cond, &mutex);

pthread_cond_signal(&cond);    // 1-1 signaling
pthread_cond_broadcast(&cond); // avalanche
```

Condition Variables

```
// global
pthread_cond_t cond;
pthread_cond_init(&cond, NULL);
…

// in separate threads
pthread_cond_wait(&cond, &mutex);

pthread_cond_signal(&cond);    // 1-1 signaling
pthread_cond_broadcast(&cond); // avalanche
```

Why is this here?

pthread3.c example

```
void* threadCode(void* arg)
 {
 struct package *realArg = arg;
 printf("Hello from %s.\n", realArg->msg);
 if( !strcmp(realArg->msg, "thread1") )
   {
   sleep(1);
   printf("Signalling in %s.\n", realArg->msg);
   pthread_cond_signal(realArg->cond);
   }
 else
   {
   printf("Waiting in %s.\n", realArg->msg);
   pthread_cond_wait(realArg->cond, realArg->mutex);
   printf("No longer waiting in %s.\n", realArg->msg);
   sleep(1);
   }
 printf("Goodbye from %s.\n", realArg->msg);
 pthread_exit(realArg->msg);
 }
```

```
struct package
{
char* msg;
pthread_cond_t* cond;
pthread_mutex_t* mutex;
};
```

pthread3.c example

```
int main(int argc, char** argv)
 {
 pthread_cond_t cond1;
 pthread_mutex_t mutex1;
 struct package arg1, arg2;
 char *result;
 pthread_t tid1, tid2;

 pthread_cond_init(&cond1, NULL);
 arg1.msg = "thread1";
 arg2.msg = "thread2";
 arg1.cond = &cond1;
 arg2.cond = &cond1;          // one cond is shared with both threads
 arg1.mutex = &mutex1;
 arg2.mutex = &mutex1;         // one mutex is shared with both threads

 pthread_create(&tid1, NULL, threadCode, &arg1);
 pthread_create(&tid2, NULL, threadCode, &arg2);

 printf("Hello from main.\n");
 sleep(1);
 printf("Goodbye from main.\n");

 pthread_join(tid1, (void*)&result);
 printf("Thread 1 joined, result is %s.\n", result);

 pthread_join(tid2, (void*)&result);
 printf("Thread 2 joined, result is %s.\n", result);
 }
```

output

```
Hello from main.
Hello from thread1.
Hello from thread2.
Waiting in thread2.
Goodbye from main.
Signalling in thread1.
Goodbye from thread1.
No longer waiting in thread2.
Thread 1 joined, result is thread1.
Goodbye from thread2.
Thread 2 joined, result is thread2.
```

Major, Major Caveat

- From the man page:
  - The pthread_cond_signal() and pthread_cond_broadcast() functions have *no effect if there are no threads currently blocked on cond*.
- This means that a collection of threads may well exhibit **time-dependent behavior** when using this primitive: **signals may be "lost"**

Semaphores

- Semaphores are a better alternative to conditional variables
- They **don’t lose signals** that may have occurred before the wait statement.
- Exactly one wait is enabled per every signal.
- Unfortunately, they are not part of Posix

Semaphores

- Each semaphore has an associated count, initially 0 by default. (May be set at > 0)
- Invariant:

  count > 0 → no processes waiting

  count = number of wait operations before blocking
- Behavior:
  - **wait**, or P, or down:

    ```
    if( count > 0 ) count--; else wait on queue;
    ```
  - **signal**, or V, or up:

    ```
    if( queue non-empty )
            wakeup one on queue;
    else count++;
    ```

Exercise

- Implement a semaphore data type using mutexes and conditional variables.

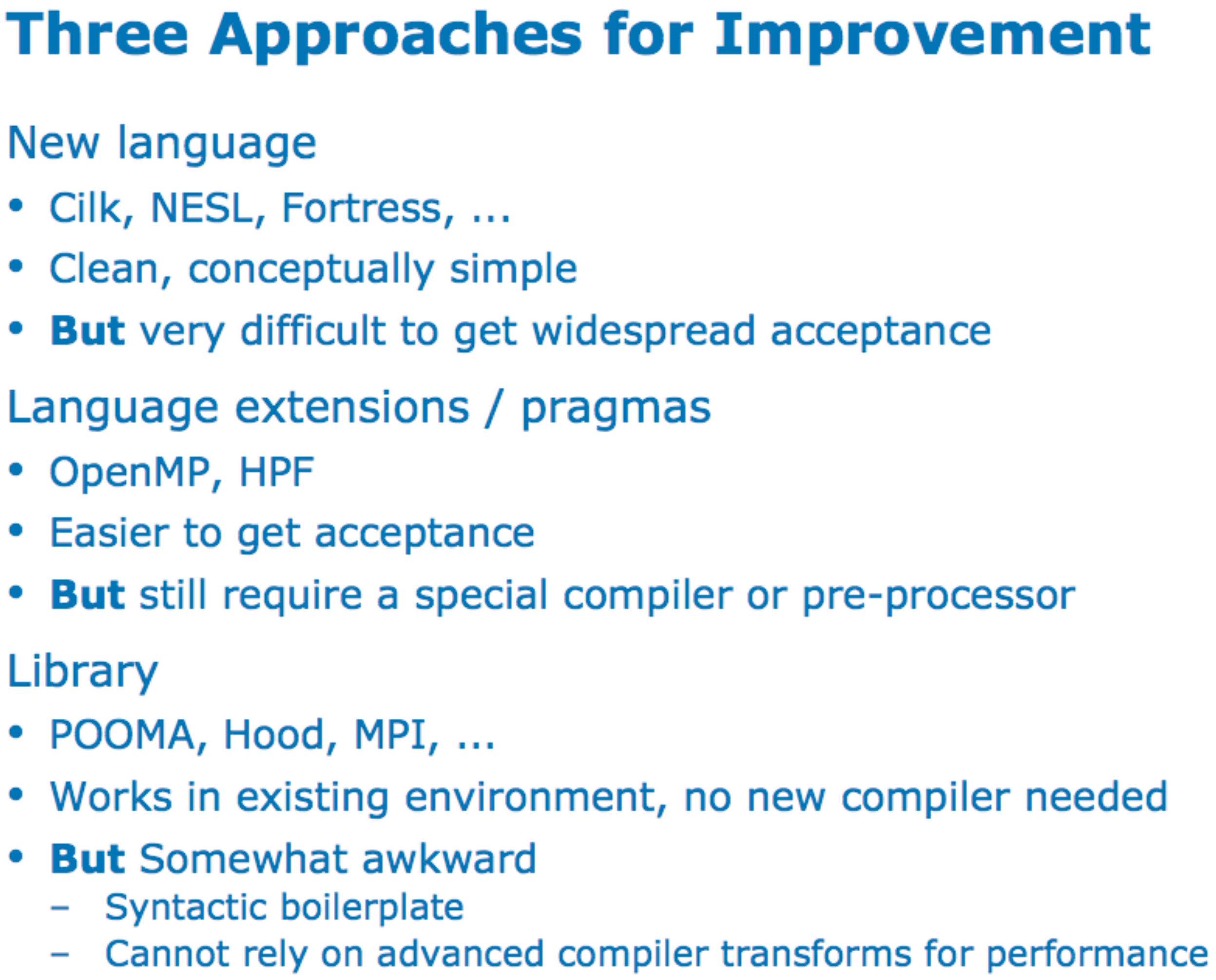
Other Threading Models

- Intel TBB (Thread Building Blocks) for C++
  - http://www.threadingbuildingblocks.org
- Cilk++
  - http://www.cilk.com
- Java
- Solaris threads, lightweight processes (lwp's)
- Microsoft models (COM, .NET, etc.)
- Open MP
  - http://openmp.org/wp/

Intel TBB (Thread Building Blocks)

- Library for C++
- Can be optimized for caching and locality (important)
- Higher level, more structured than pthreads
- Relies on templates,
  a la Standard Library

TBB Rationale



Three Approaches for Improvement

New language

- Cilk, NESL, Fortress, ...
- Clean, conceptually simple
- **But** very difficult to get widespread acceptance

Language extensions / pragmas

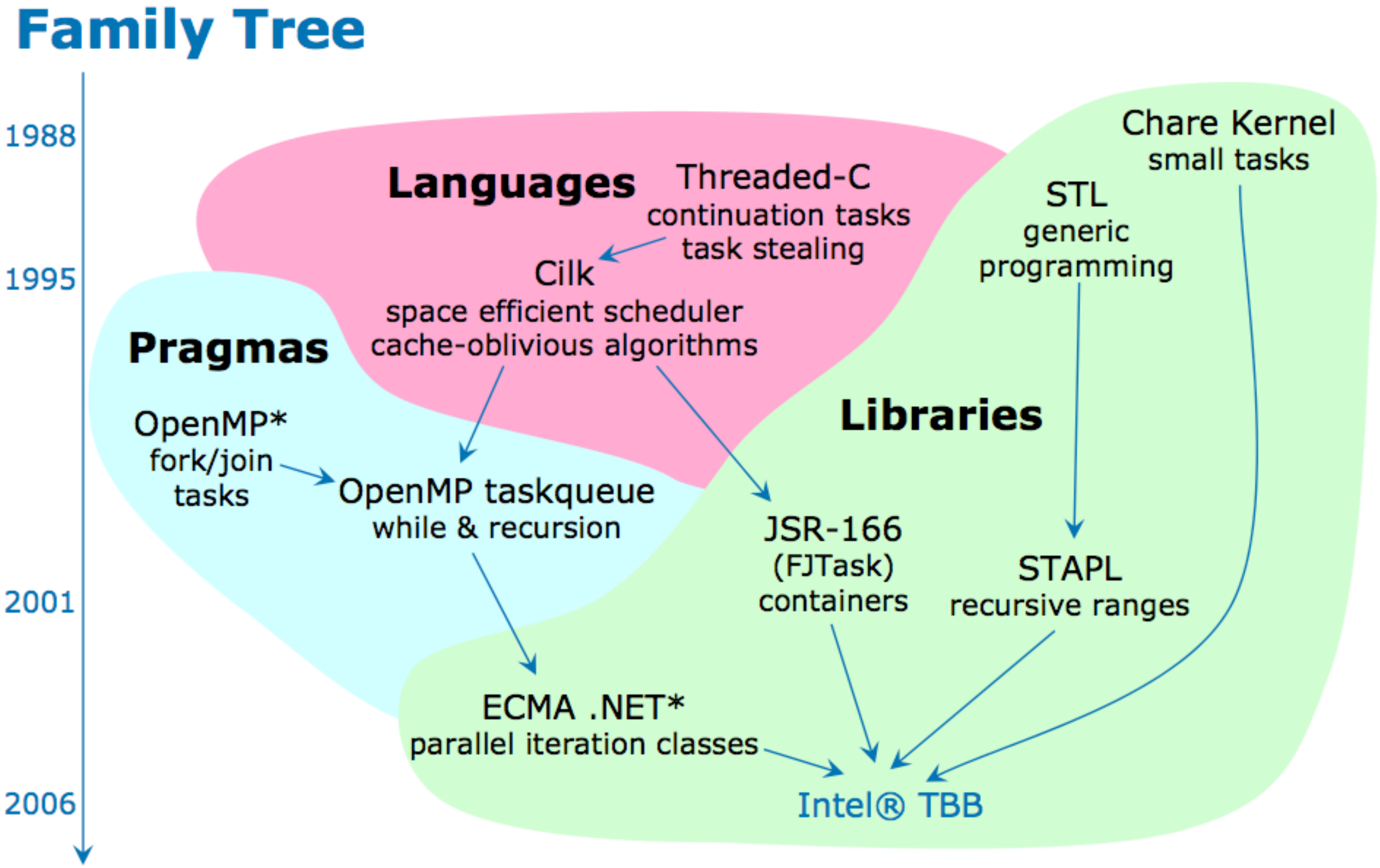
- OpenMP, HPF
- Easier to get acceptance
- **But** still require a special compiler or pre-processor

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

Source: Arch Robison, Intel, HPCC ‘07, Houston

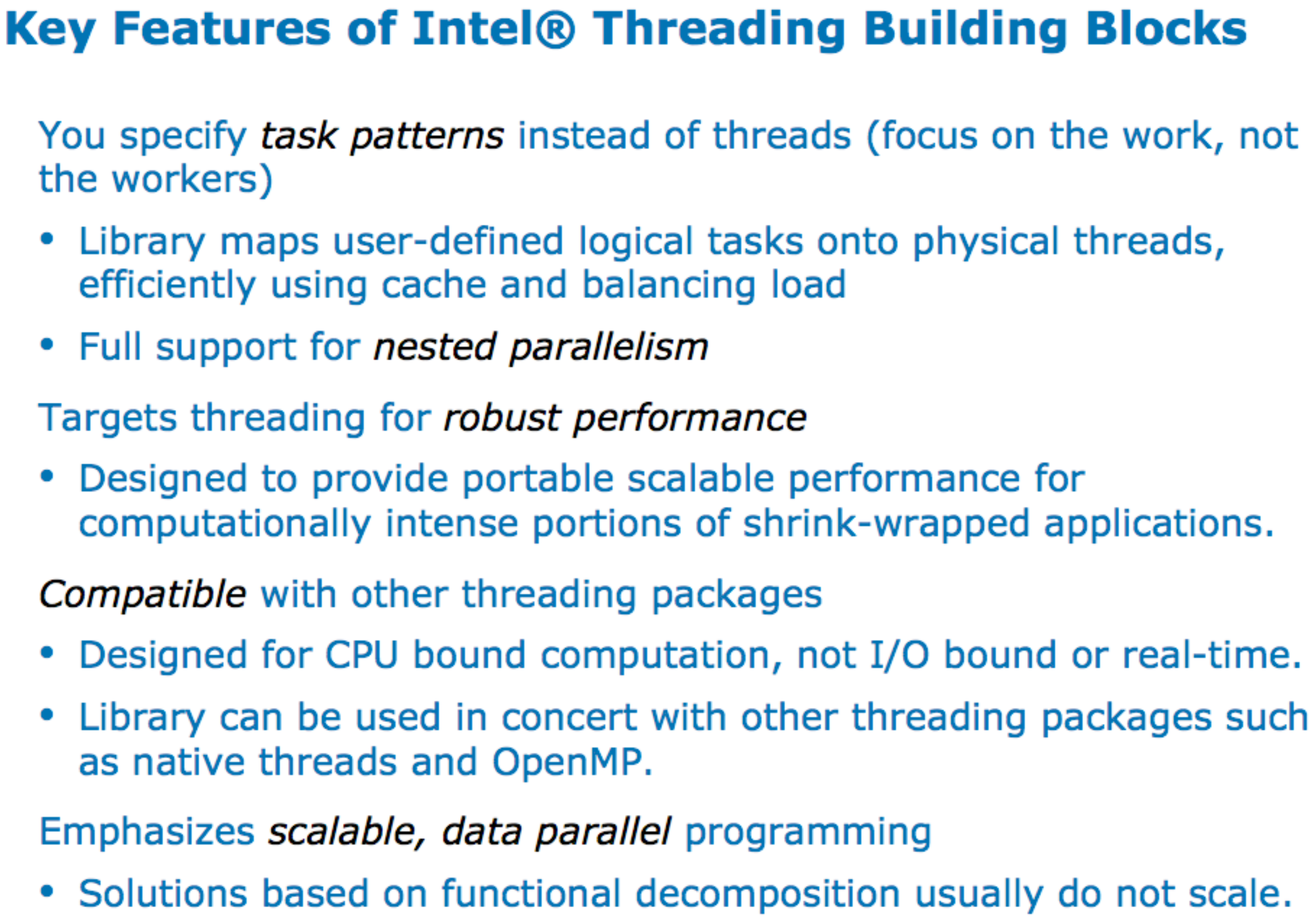
TBB Background



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

Source: Arch Robison, Intel, HPCC ‘07, Houston

TBB Features



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

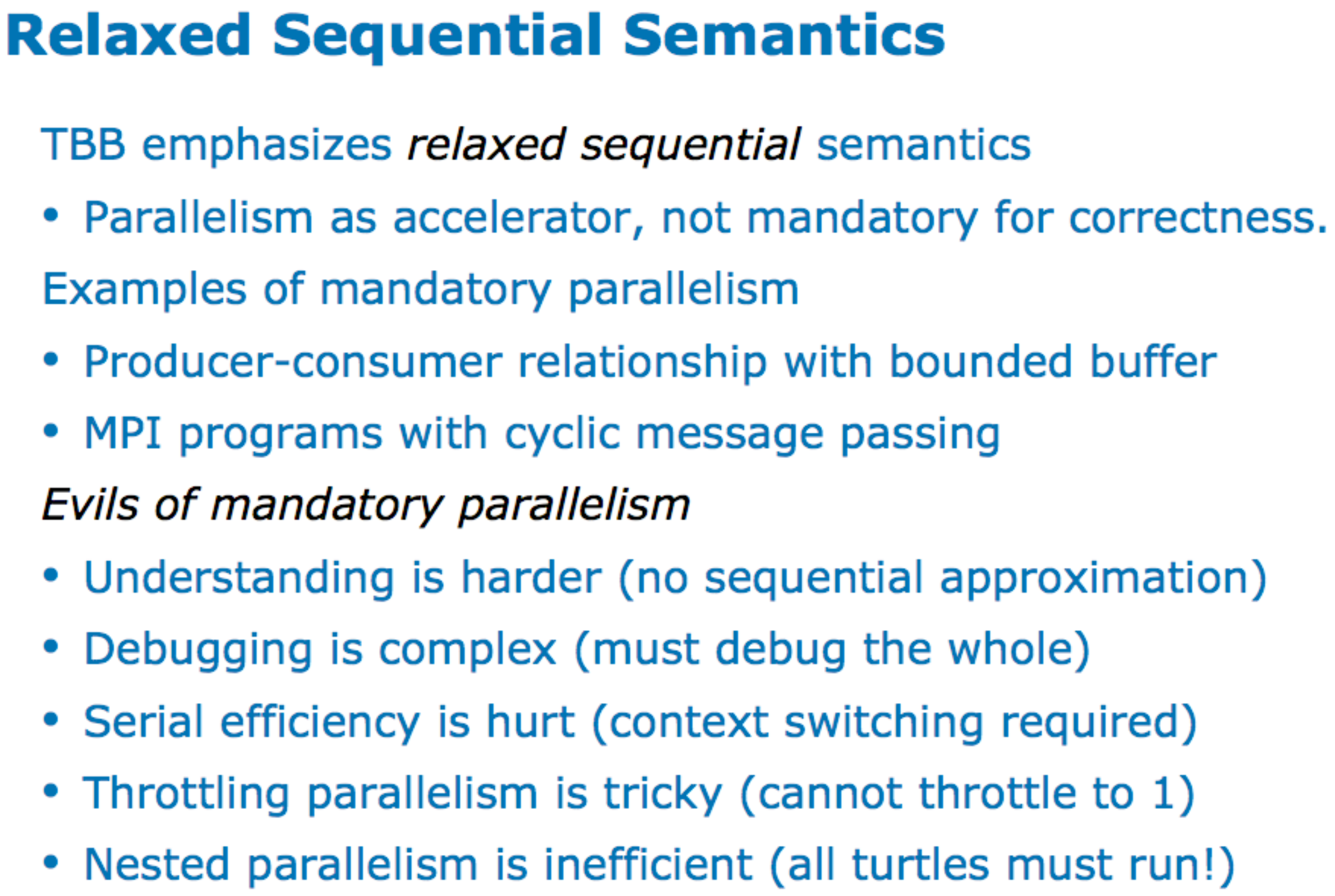
- Designed for CPU bound computation, not I/O bound or real-time.
- Library can be used in concert with other threading packages such as native threads and OpenMP.

Emphasizes *scalable, data parallel* programming

- Solutions based on functional decomposition usually do not scale.

Source: Arch Robison, Intel, HPCC ‘07, Houston

Optional vs. Mandatory Parallelism



Relaxed Sequential Semantics

TBB emphasizes *relaxed sequential* semantics

- Parallelism as accelerator, not mandatory for correctness.

Examples of mandatory parallelism

- Producer-consumer relationship with bounded buffer
- MPI programs with cyclic message passing

Evils of mandatory parallelism

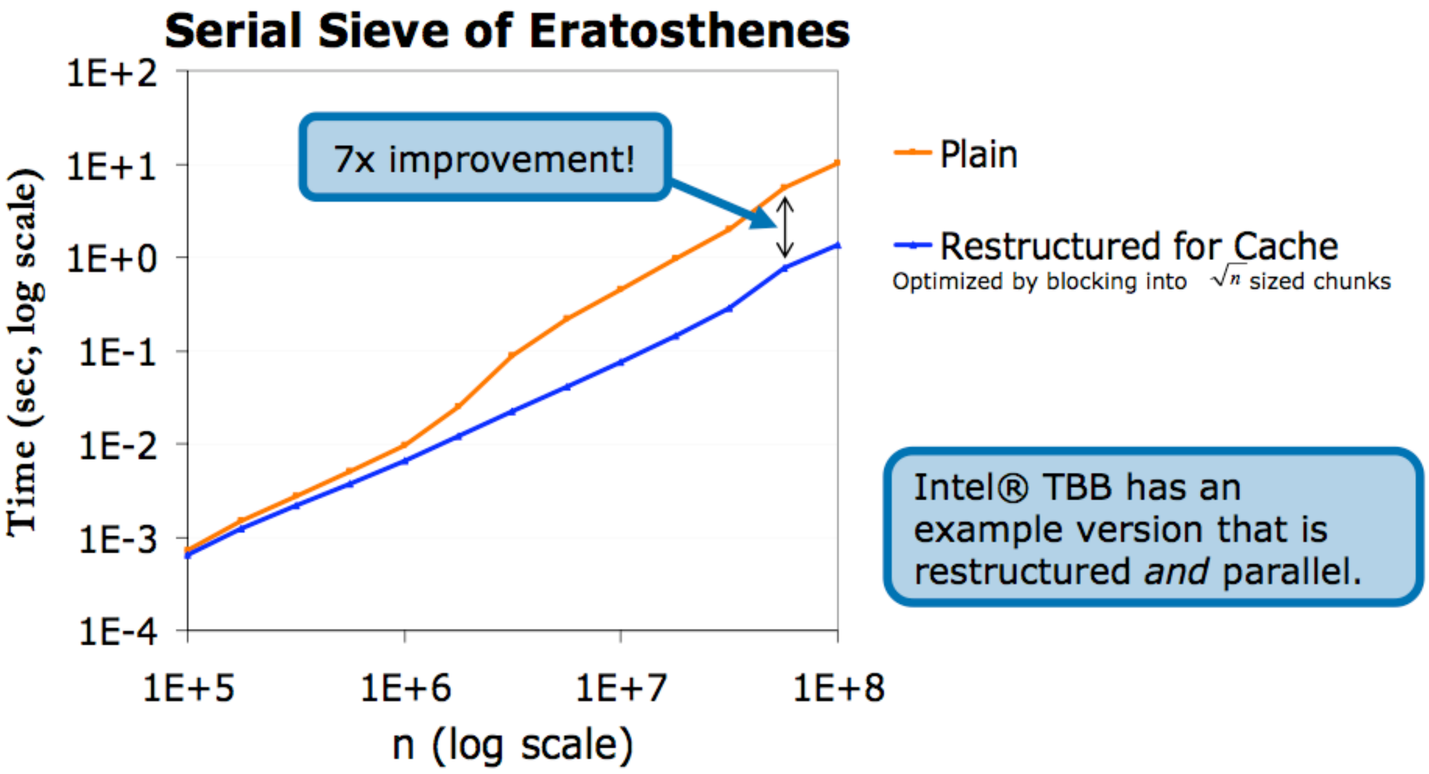
- Understanding is harder (no sequential approximation)
- Debugging is complex (must debug the whole)
- Serial efficiency is hurt (context switching required)
- Throttling parallelism is tricky (cannot throttle to 1)
- Nested parallelism is inefficient (all turtles must run!)

Source: Arch Robison, Intel, HPCC ‘07, Houston

Cache Optimization

Optimizing for Cache Is Critical

Optimizing for cache can beat small-scale parallelism



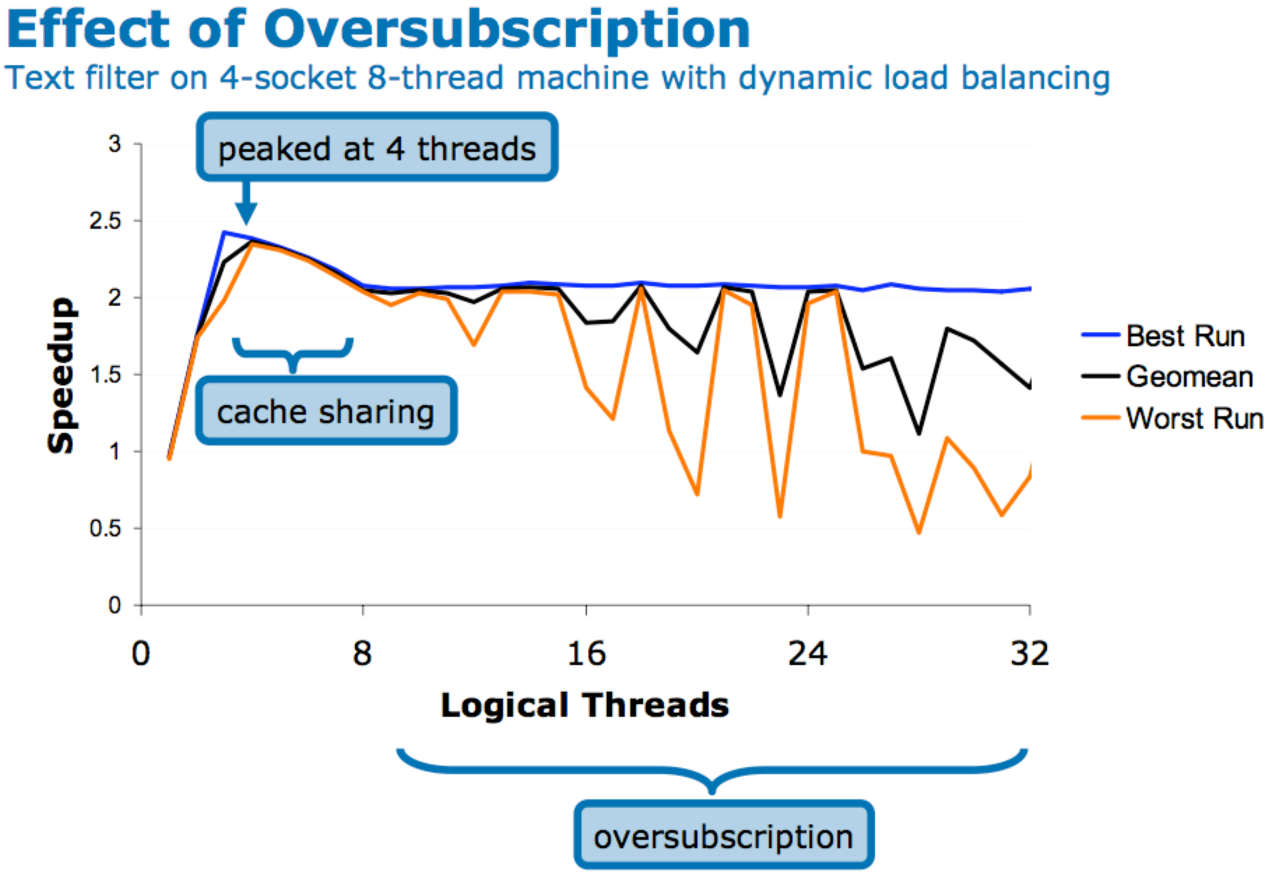
Serial Sieve of Eratosthenes

7x improvement!

Intel® TBB has an example version that is restructured *and* parallel.

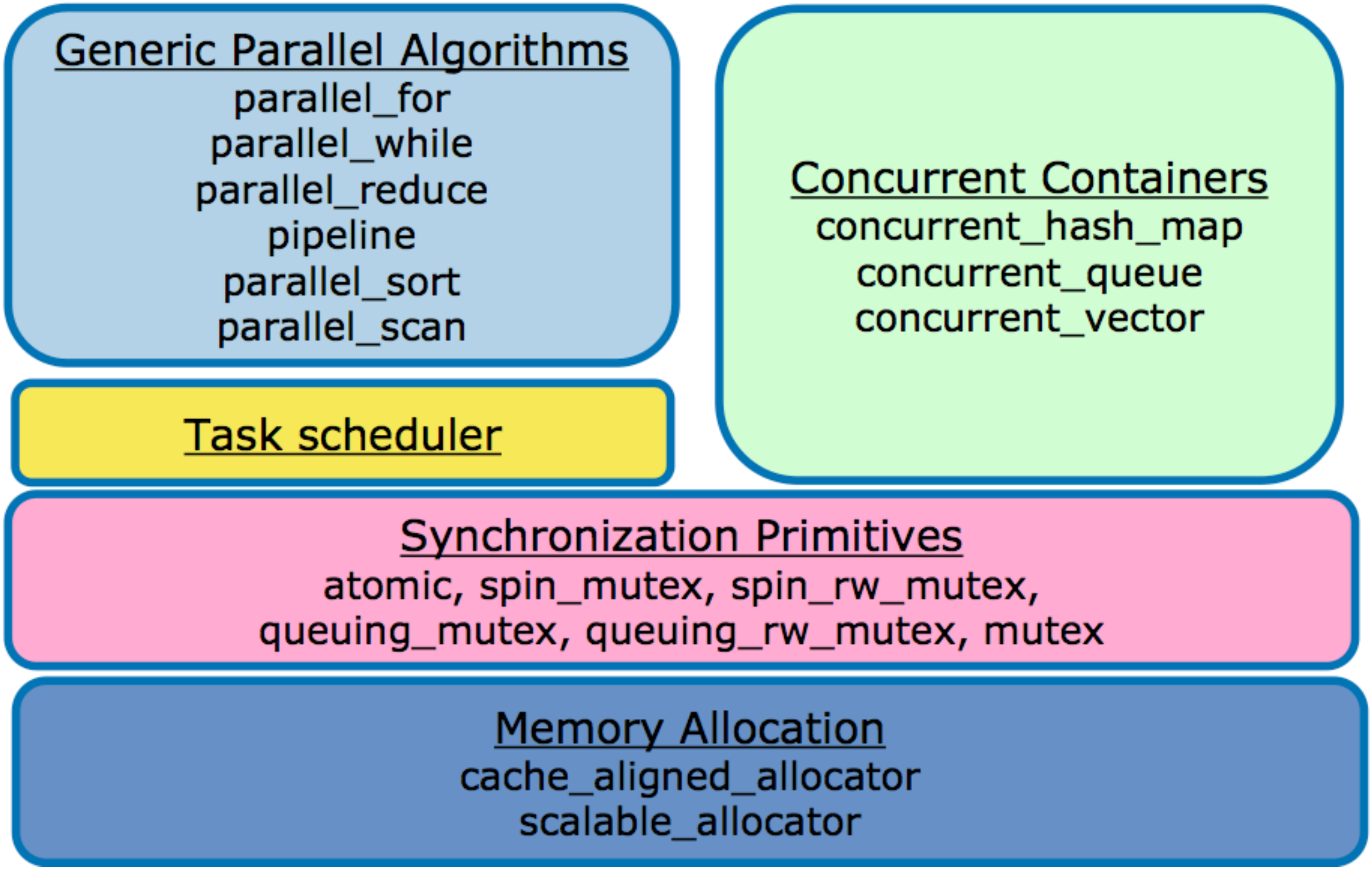
Source: Arch Robison, Intel, HPCC ‘07, Houston

Granularity Control



Source: Arch Robison, Intel, HPCC '07, Houston

TBB Components



Generic Parallel Algorithms
parallel_for
parallel_while
parallel_reduce
pipeline
parallel_sort
parallel_scan

Concurrent Containers
concurrent_hash_map
concurrent_queue
concurrent_vector

Task scheduler

Synchronization Primitives
atomic, spin_mutex, spin_rw_mutex,
queuing_mutex, queuing_rw_mutex, mutex

Memory Allocation
cache_aligned_allocator
scalable_allocator

Source: Arch Robison, Intel, HPCC ‘07, Houston

TBB Parallelism Example

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

Parallelize by **dividing iteration space** of i into chunks.

```
void ParallelApplyFoo( float a[], size_tn )
{
parallel_for( blocked_range<int>( 0, n ),
          ApplyFoo(a),                  // wrapped version of Foo

          auto_partitioner() );         // grain-size control
}
```

Source: Arch Robison, Intel, HPCC ‘07, Houston

Wrapping for Parallelism

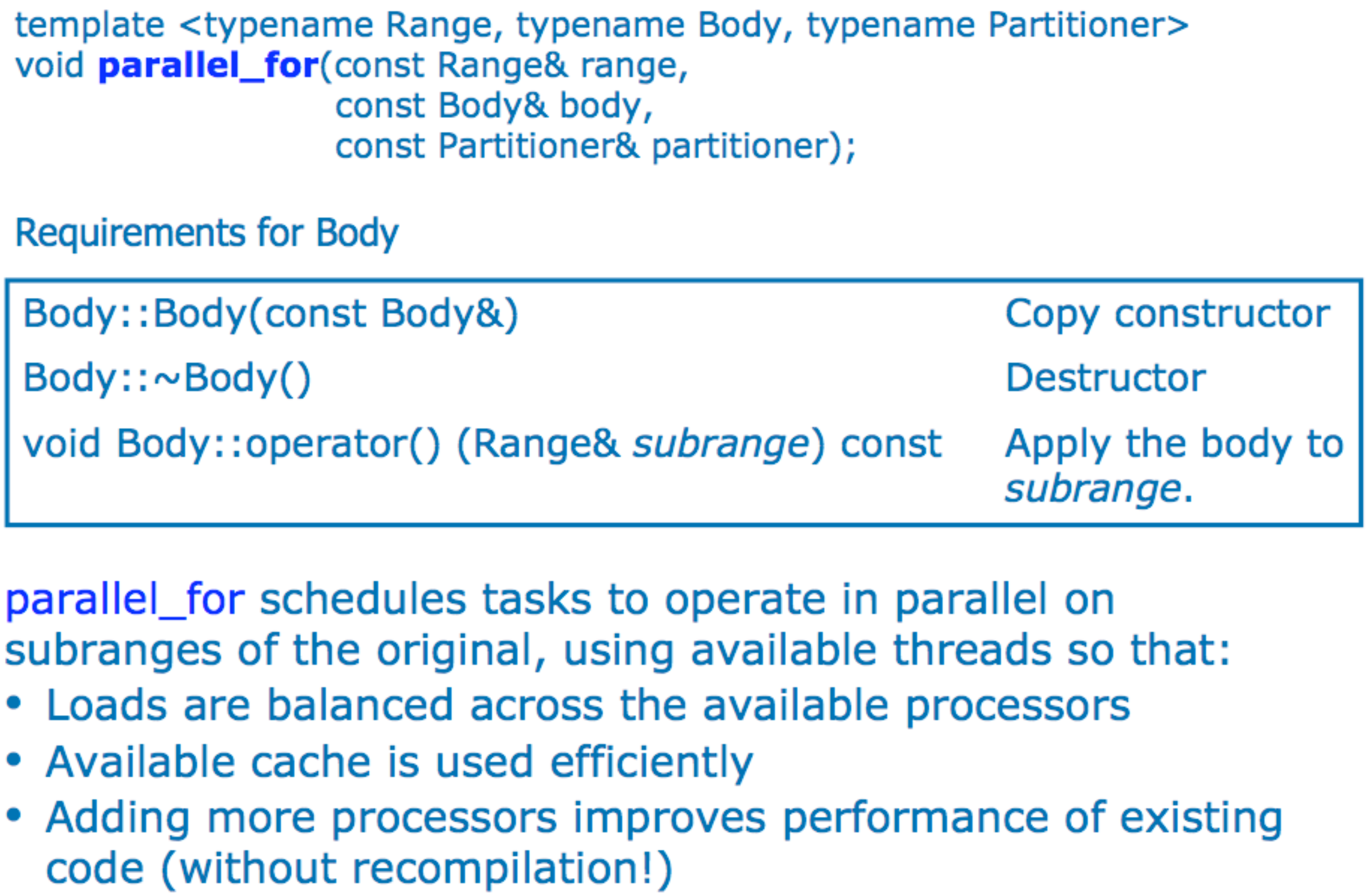
```
class ApplyFoo                          // new wrapper class
{
float *const my_a;

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]);
        }
};
```

Source: Arch Robison, Intel, HPCC ‘07, Houston

Patterns Represented by Library Templates



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

Requirements for Body

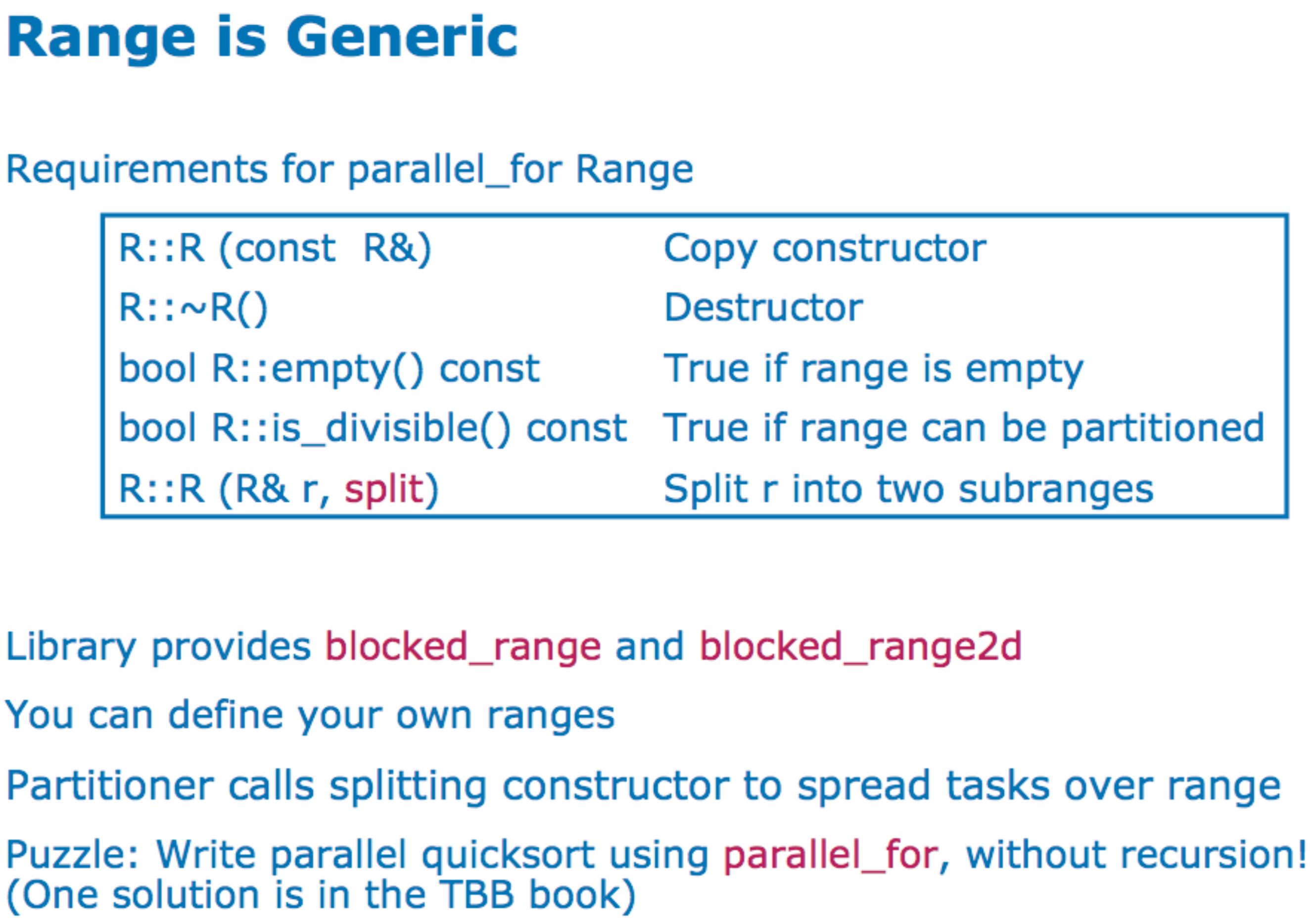
Body::Body(const Body&)	Copy constructor
Body::~Body()	Destructor
void Body::operator() (Range& *subrange*) const	Apply the body to *subrange*.

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!)

Source: Arch Robison, Intel, HPCC ‘07, Houston

Range Interface Requirements



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

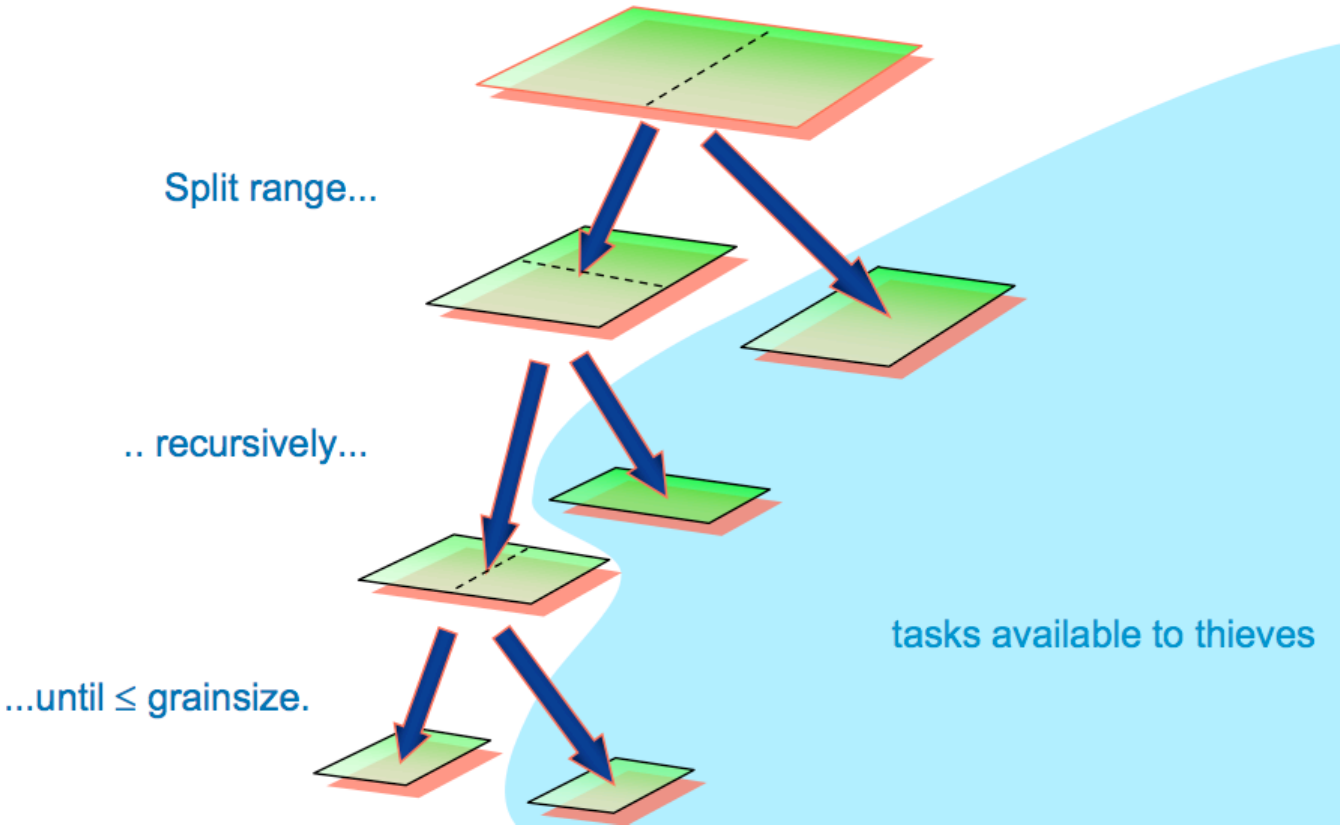
Partitioner calls splitting constructor to spread tasks over range

Puzzle: Write parallel quicksort using parallel_for, without recursion!
(One solution is in the TBB book)

Source: Arch Robison, Intel, HPCC '07, Houston

Work-Stealing and Grain Control

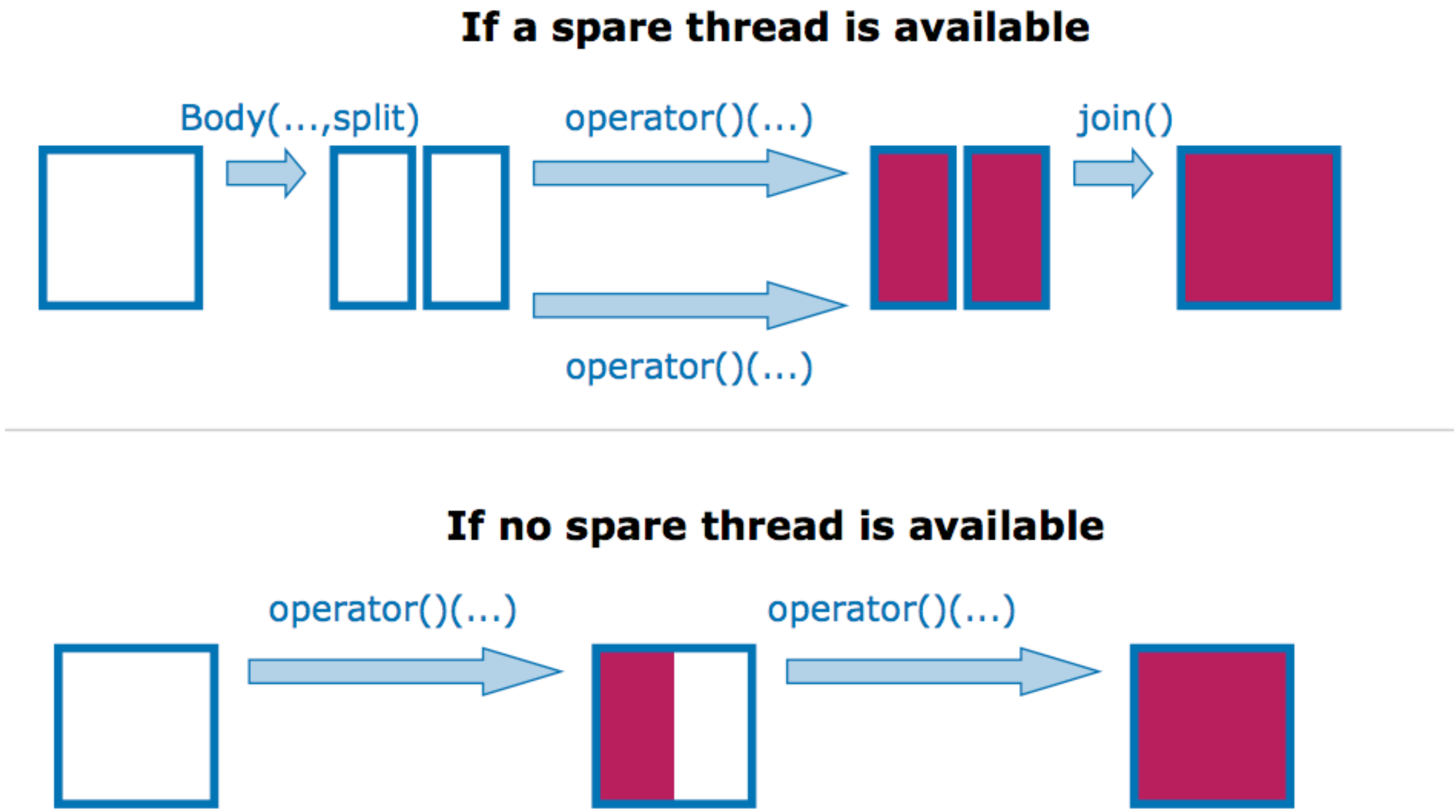
How this works on `blocked_range2d`



Source: Arch Robison, Intel, HPCC ‘07, Houston

Work-Stealing and Grain Control

Lazy Parallelism in parallel_reduce

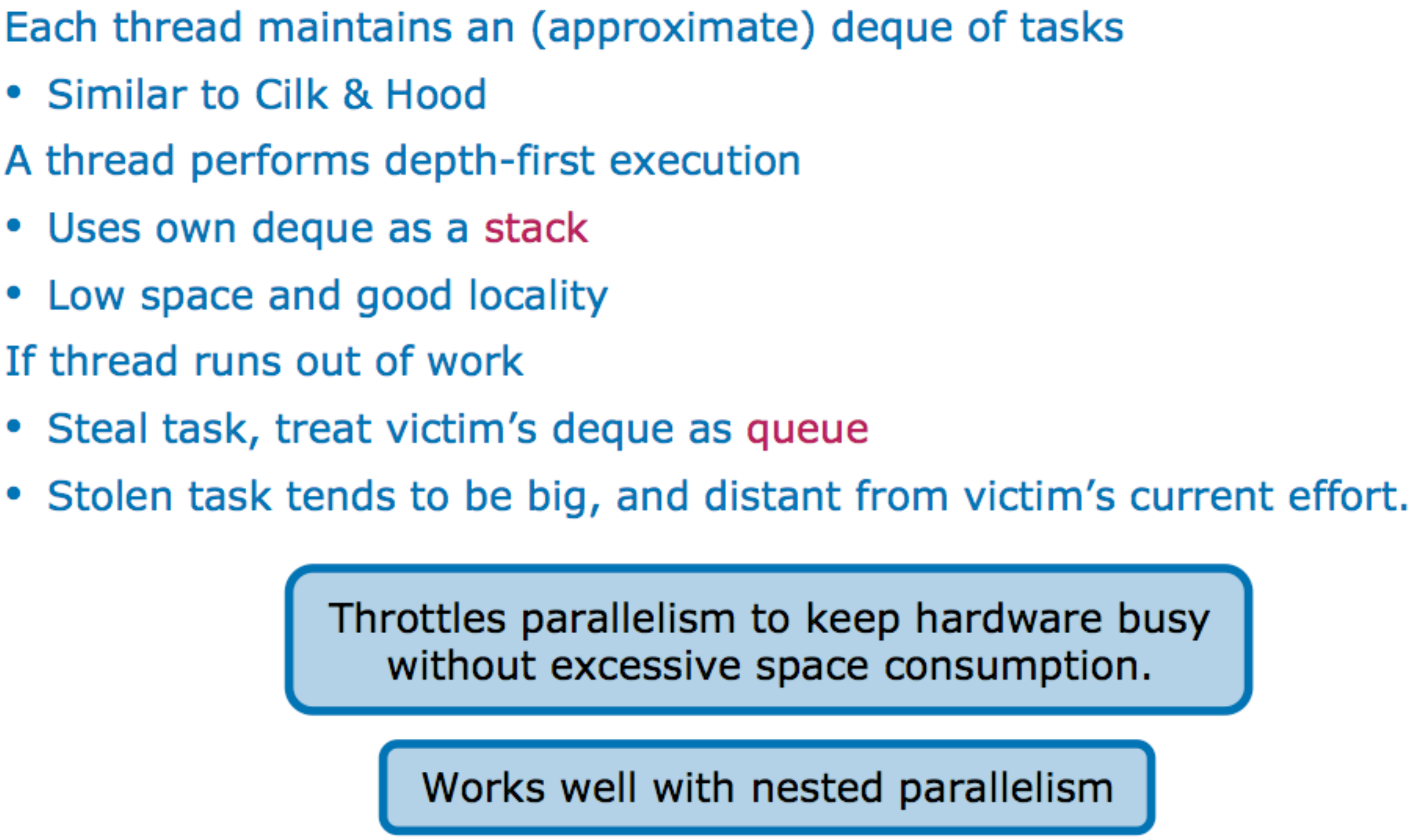


If a spare thread is available

If no spare thread is available

Source: Arch Robison, Intel, HPCC ‘07, Houston

Work-Stealing



Each thread maintains an (approximate) deque of tasks

- Similar to Cilk & Hood

A thread performs depth-first execution

- Uses own deque as a stack
- Low space and good locality

If thread runs out of work

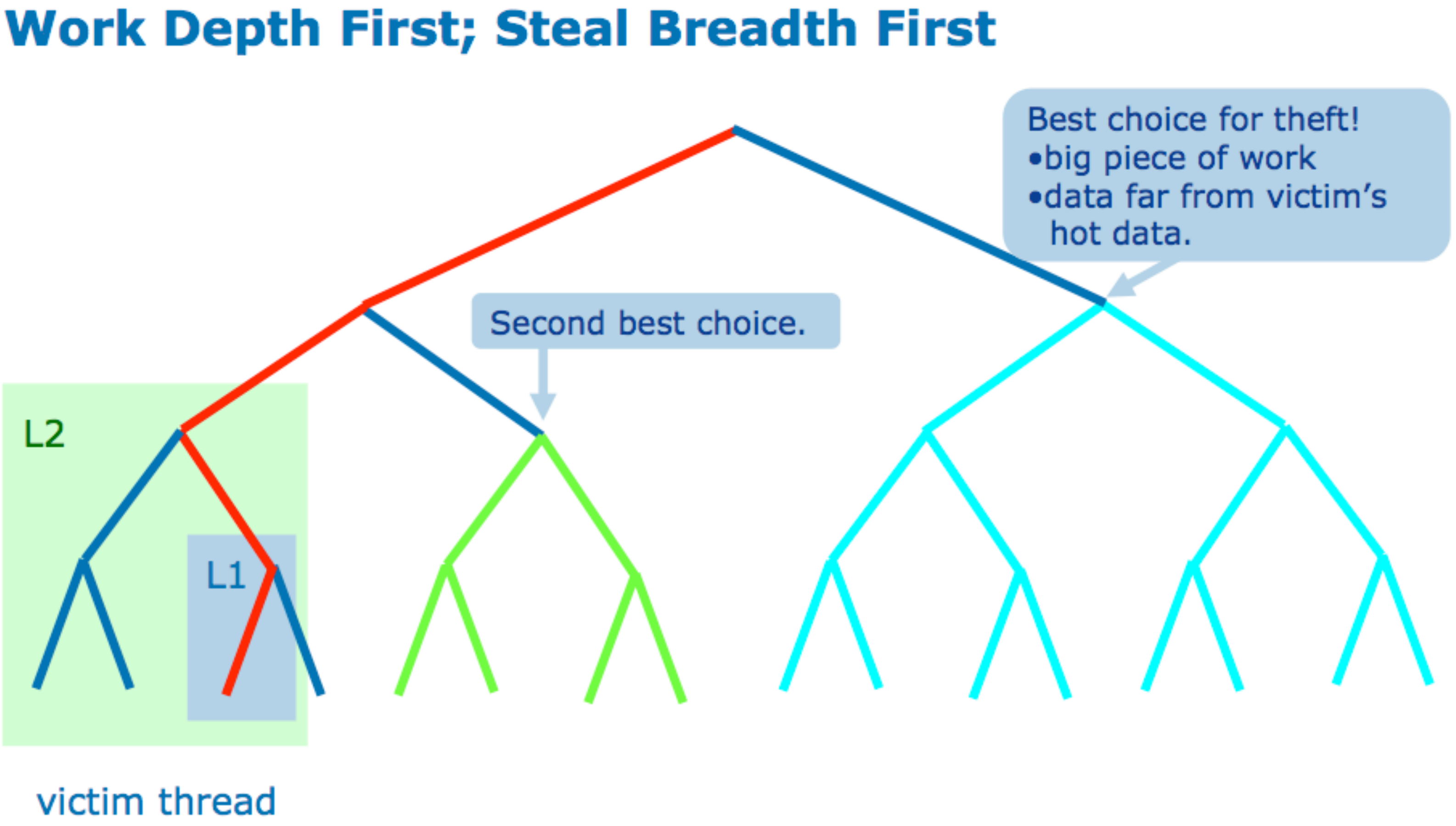
- Steal task, treat victim's deque as queue
- Stolen task tends to be big, and distant from victim's current effort.

Throttles parallelism to keep hardware busy
without excessive space consumption.

Works well with nested parallelism

Source: Arch Robison, Intel, HPCC '07, Houston

Work-Stealing and Locality



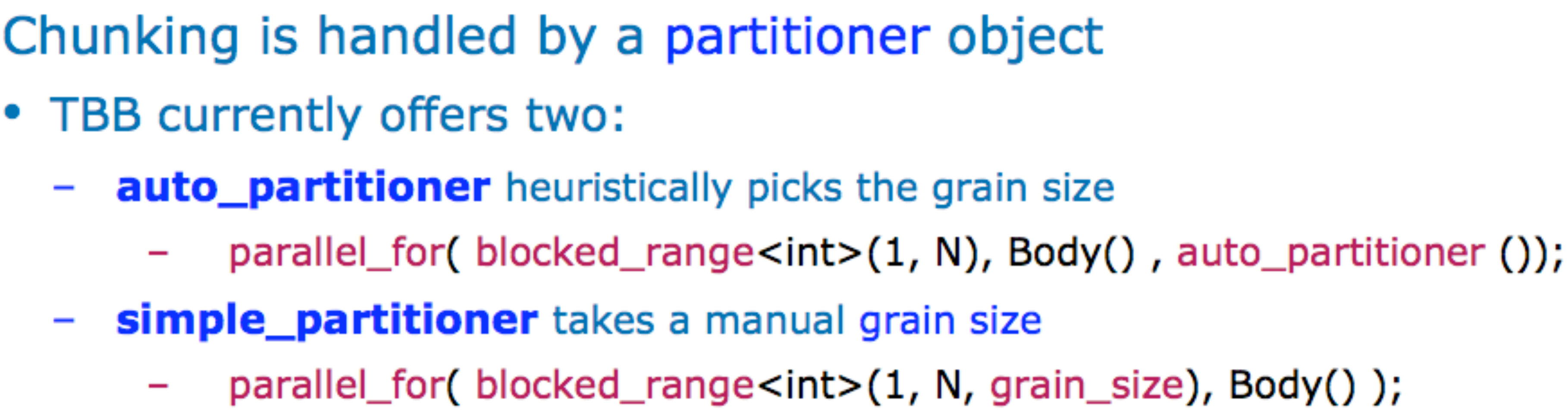
Work Depth First; Steal Breadth First

Best choice for theft!
- big piece of work
- data far from victim's hot data.

Second best choice.

Source: Arch Robison, Intel, HPCC '07, Houston

Grain Control

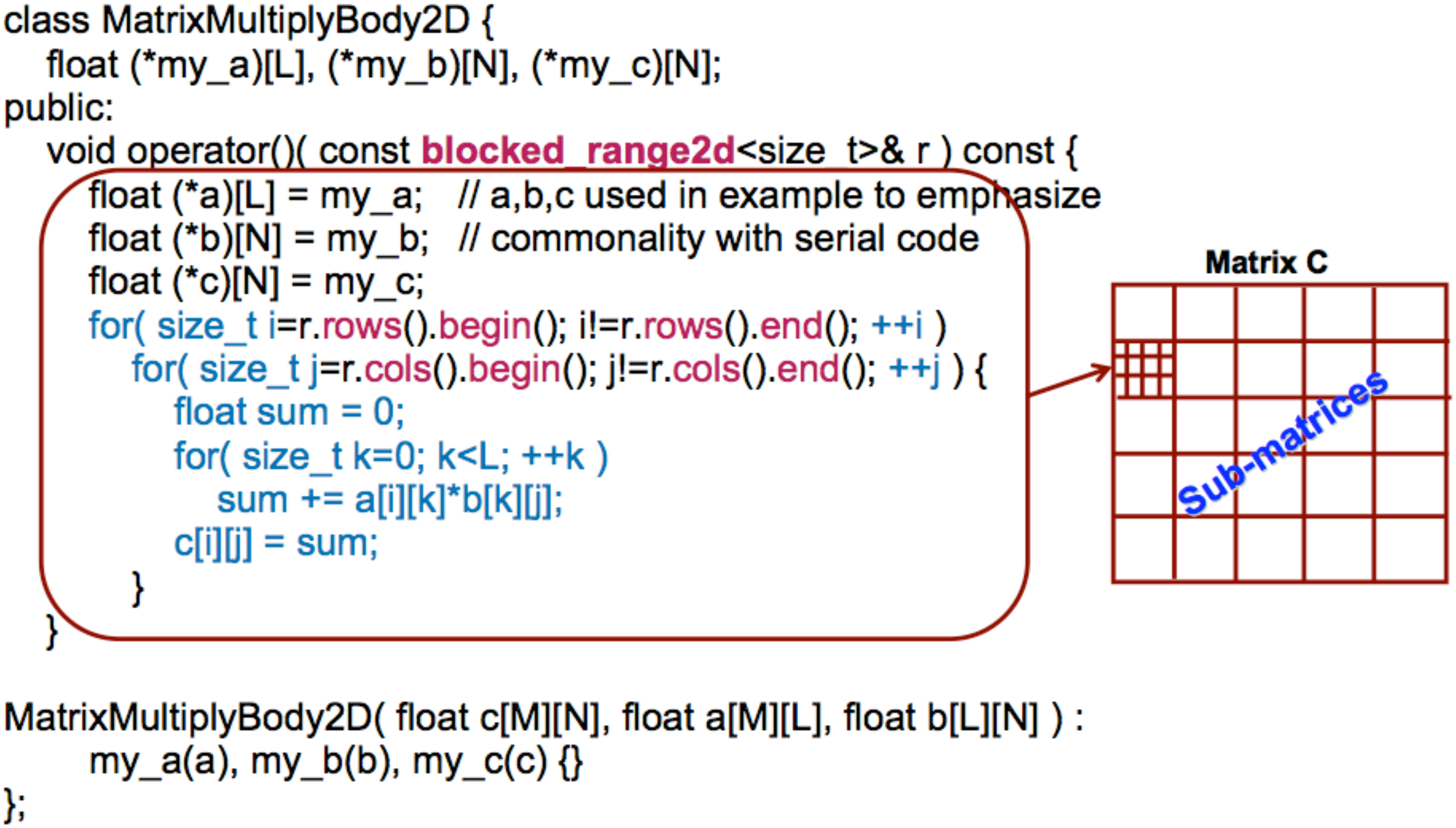


Chunking is handled by a **partitioner object**

- TBB currently offers two:
  - **auto_partitioner** heuristically picks the grain size
    - `parallel_for( blocked_range<int>(1, N), Body() , auto_partitioner ());`
  - **simple_partitioner** takes a manual grain size
    - `parallel_for( blocked_range<int>(1, N, grain_size), Body() );`

Source: Arch Robison, Intel, HPCC ‘07, Houston

Matrix-Multiply Example

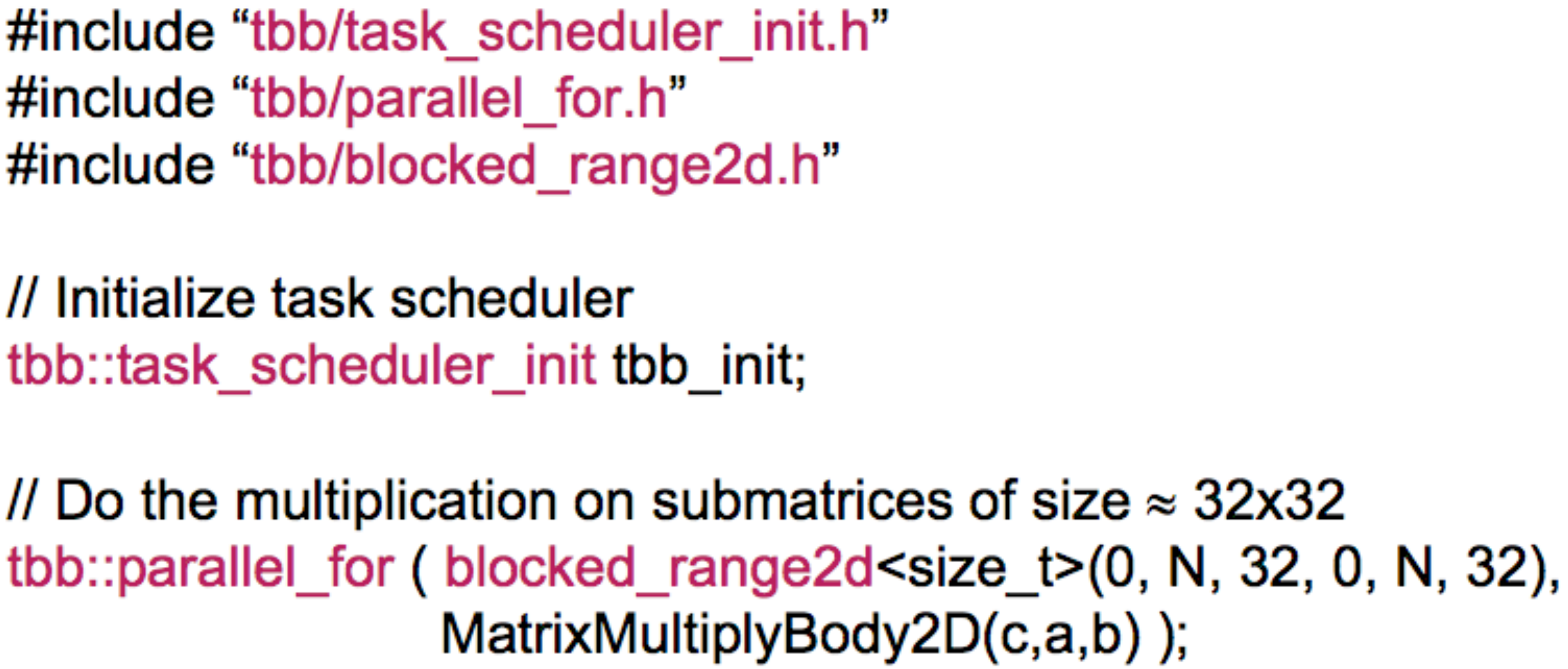


```
class MatrixMultiplyBody2D {
    float (*my_a)[L], (*my_b)[N], (*my_c)[N];
public:
    void operator()( const blocked_range2d<size_t>& r ) const {
        float (*a)[L] = my_a;   // a,b,c used in example to emphasize
        float (*b)[N] = my_b;   // commonality with serial code
        float (*c)[N] = my_c;
        for( size_t i=r.rows().begin(); i!=r.rows().end(); ++i )
            for( size_t j=r.cols().begin(); j!=r.cols().end(); ++j ) {
                float sum = 0;
                for( size_t k=0; k<L; ++k )
                    sum += a[i][k]*b[k][j];
                c[i][j] = sum;
            }
    }

MatrixMultiplyBody2D( float c[M][N], float a[M][L], float b[L][N] ) :
        my_a(a), my_b(b), my_c(c) {}
};
```

Source: Arch Robison, Intel, HPCC ‘07, Houston

Matrix-Multiply Example



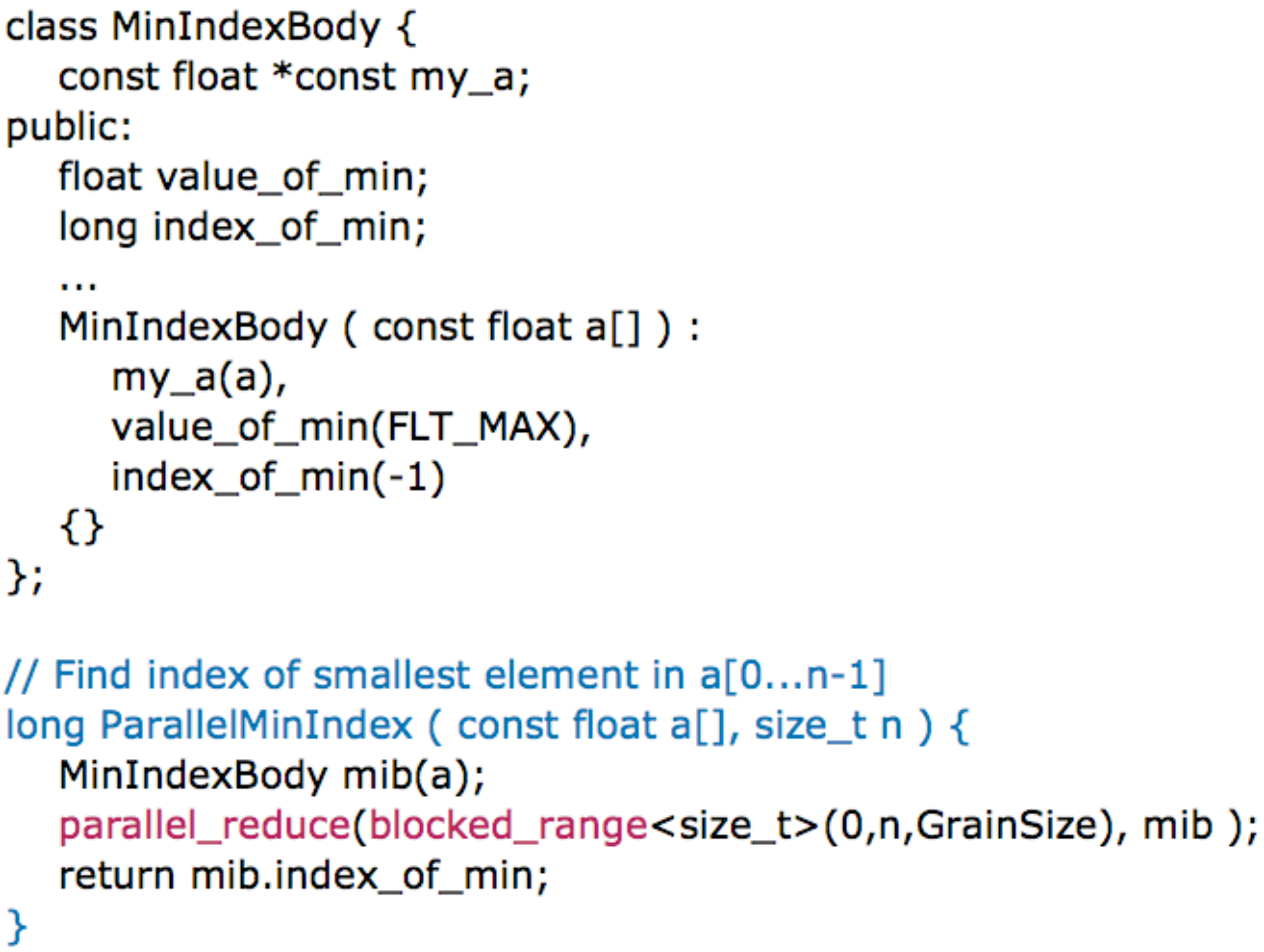
```
#include "tbb/task_scheduler_init.h"
#include "tbb/parallel_for.h"
#include "tbb/blocked_range2d.h"

// Initialize task scheduler
tbb::task_scheduler_init tbb_init;

// Do the multiplication on submatrices of size ≈ 32x32
tbb::parallel_for ( blocked_range2d<size_t>(0, N, 32, 0, N, 32),
                    MatrixMultiplyBody2D(c,a,b) );
```

Source: Arch Robison, Intel, HPCC '07, Houston

Parallel-Reduce Example (1 of 2)

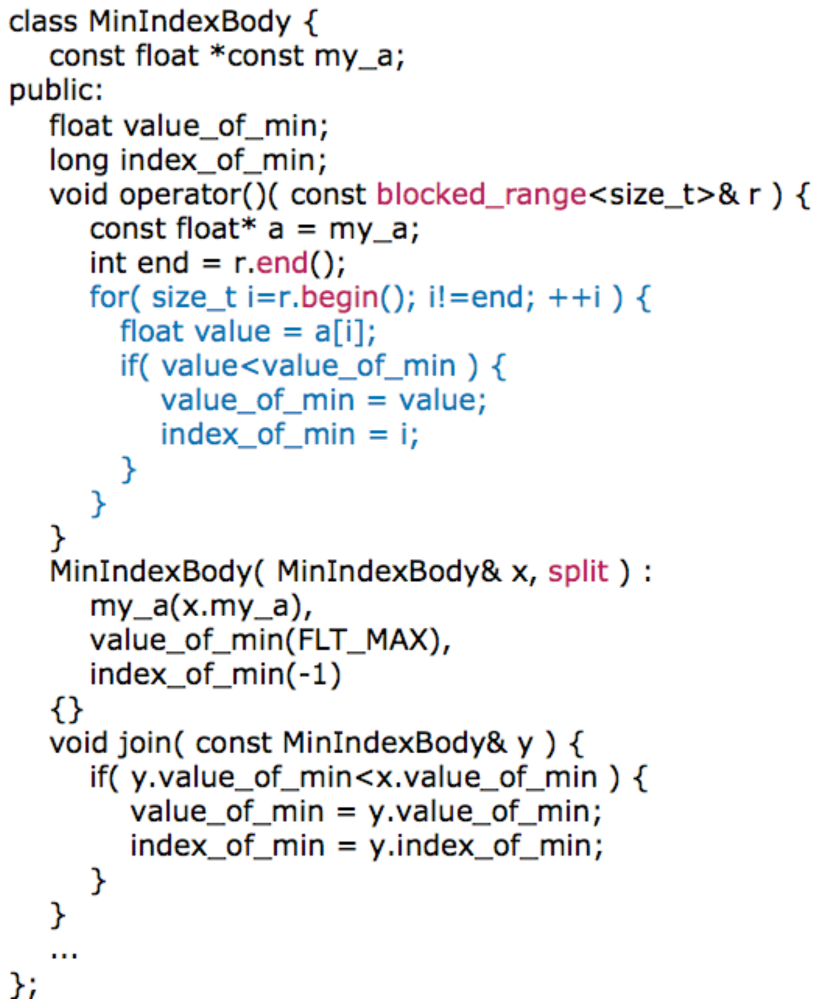


```
class MinIndexBody {
    const float *const my_a;
public:
    float value_of_min;
    long index_of_min;
    ...
    MinIndexBody ( const float a[] ) :
        my_a(a),
        value_of_min(FLT_MAX),
        index_of_min(-1)
    {}
};

// Find index of smallest element in a[0...n-1]
long ParallelMinIndex ( const float a[], size_t n ) {
    MinIndexBody mib(a);
    parallel_reduce(blocked_range<size_t>(0,n,GrainSize), mib );
    return mib.index_of_min;
}
```

Source: Arch Robison, Intel, HPCC ‘07, Houston

Parallel-Reduce Example (2 of 2)



```
class MinIndexBody {
   const float *const my_a;
public:
   float value_of_min;
   long index_of_min;
   void operator()( const blocked_range<size_t>& r ) {
      const float* a = my_a;
      int end = r.end();
      for( size_t i=r.begin(); i!=end; ++i ) {
         float value = a[i];
         if( value<value_of_min ) {
            value_of_min = value;
            index_of_min = i;
         }
      }
   }
   MinIndexBody( MinIndexBody& x, split ) :
      my_a(x.my_a),
      value_of_min(FLT_MAX),
      index_of_min(-1)
   {}
   void join( const MinIndexBody& y ) {
      if( y.value_of_min<x.value_of_min ) {
         value_of_min = y.value_of_min;
         index_of_min = y.index_of_min;
      }
   }
   ...
};
```

accumulate result

split

join

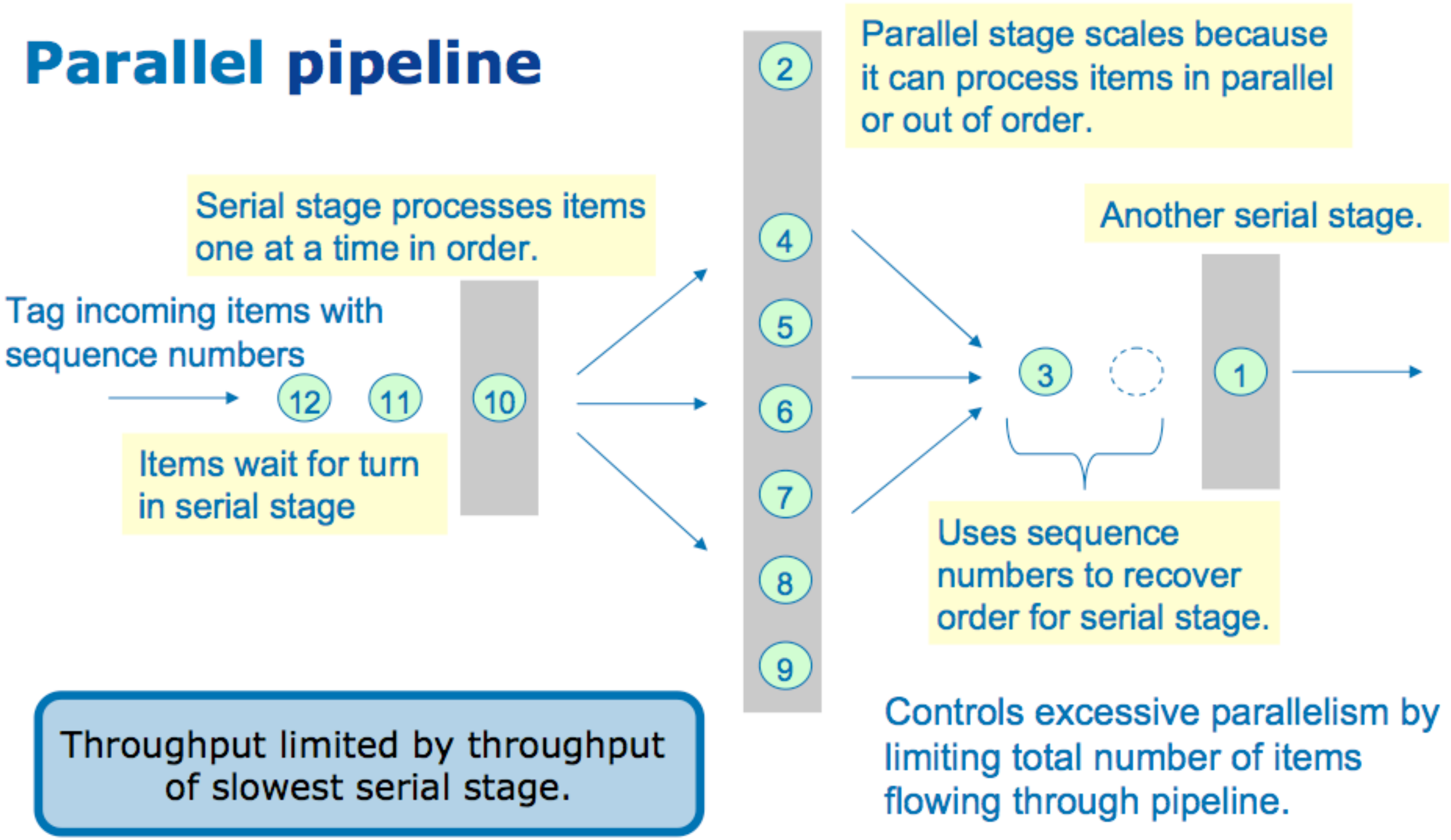
Source: Arch Robison, Intel, HPCC ‘07, Houston

Similar Templates

- parallel_scan
- parallel_sort
- parallel_while

- parallel_pipeline

Parallel Pipeline Concept



Parallel pipeline

Parallel stage scales because it can process items in parallel or out of order.

Serial stage processes items one at a time in order.

Another serial stage.

Tag incoming items with sequence numbers

Items wait for turn in serial stage

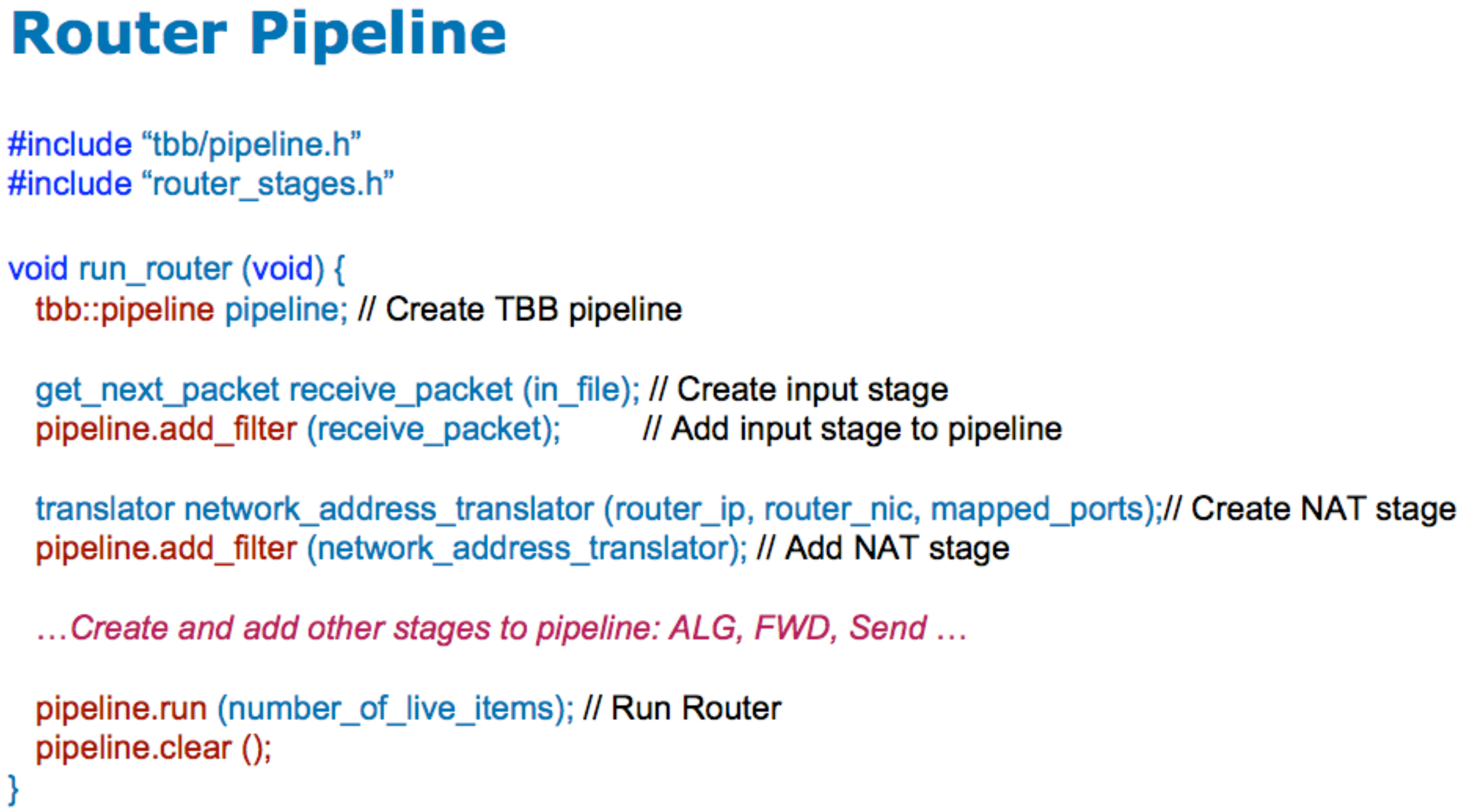
Uses sequence numbers to recover order for serial stage.

Throughput limited by throughput of slowest serial stage.

Controls excessive parallelism by limiting total number of items flowing through pipeline.

Source: Arch Robison, Intel, HPCC ‘07, Houston

Parallel Pipeline Code



Router Pipeline

```
#include “tbb/pipeline.h”
#include “router_stages.h”

void run_router (void) {
  tbb::pipeline pipeline; // Create TBB pipeline

  get_next_packet receive_packet (in_file); // Create input stage
  pipeline.add_filter (receive_packet);       // Add input stage to pipeline

  translator network_address_translator (router_ip, router_nic, mapped_ports);// Create NAT stage
  pipeline.add_filter (network_address_translator); // Add NAT stage

  …Create and add other stages to pipeline: ALG, FWD, Send …

  pipeline.run (number_of_live_items); // Run Router
  pipeline.clear ();
}
```

Source: Arch Robison, Intel, HPCC ‘07, Houston