# Threading Building Blocks and OpenMP Implementation Issues

**John Mellor-Crummey**

**Department of Computer Science**
**Rice University**

**johnmc@rice.edu**

# Context

- Language-based parallel programming models
  - —Cilk, Cilk++: extensions to C and C++
- Library and directive-based models
  - —Threading Building Blocks (TBB): library-based model
  - —OpenMP: directive-based programming model

# Outline

- Threading Building Blocks
  - overview
  - tasks and scheduling
  - scalable allocator
- OpenMP
  - overview
  - lightweight techniques for managing parallel regions on BG/Q

# Thread Building Blocks

- C++ template library used for multi-threading
- Features
  - like Cilk, a task-based programming model
  - abstracts away implementation details of task parallelism
    - number of threads
    - mapping of task to threads
    - mapping of threads to processors
    - memory and locality
  - advantages
    - reduces source code complexity over PThreads
    - runtime determines parameters automatically
    - automatically scales parallelism to exploit all available cores

# Threading Building Blocks Components

- Parallel algorithmic templates
  - parallel_for
  - parallel_reduce
  - pipeline
  - parallel_sort
  - parallel_while
  - parallel_scan
- Synchronization primitives
  - atomic ops on integer types
  - mutex
  - spin_mutex
  - queuing_mutex
  - spin_rw_mutex
  - queuing_rw_mutex
- Concurrent containers
  - concurrent hash map
  - concurrent queue
  - concurrent vector
- Task scheduler
- Memory allocators
  - cache_aligned_allocator
  - scalable_allocator
- Timing
  - tick_count

# Comparing Implementations of Fibonacci

## TBB

```
class FibTask : public task {   // describe the task
public:
        const long n;
        long* const sum;
    FibTask(long n_, long* sum_):
        n(n_),  sum(sum_) {}
    task* execute() {  // do the work
        if(n<=cutoff) *sum=SerialFib(n);
        else {   // generate more tasks
          long x, y;
          FibTask& a = *new(allocate_child())
                FibTask(n-1, &x);
          FibTask& b = *new(allocate_child())
                FibTask(n-2, &y);
          set_ref_count(3);
          spawn(b);
          spawn_and_wait_for_all(a);
          *sum=x+y;
    }
```

## Cilk++

```
int fib(n)  {
  if (n < 2) return n;
  else {
    int n1, n2;
    n1 = cilk_spawn fib(n-1);
    n2 = fib(n-2);
    cilk_sync;
    return (n1 + n2);
  }
```

# Comparing Fibonacci Scaffolding

**TBB**

```
long ParallelFib(long n)  {
   long sum;
   FibTask& a = *new(task::allocate_root())
          FibTask(n, &sum);
   task::spawn_root_and_wait(a);
   return sum;
}
long SerialFib( long n ) {
   if (n<2) return n;
   else return SerialFib(n-1)+SerialFib(n-2);
}
int main() {
  task_scheduler_init init;
  ParallelFib(40);
}
```

**Cilk++**

```
int main() {
   fib(40);
}
```

# TBB Scheduler

- **Automatically balances the load across cores**
  - **work stealing**
- **Schedules tasks to exploit the cache locality of applications**
- **Avoids over-subscription of resources**
  - **uses tasks for scheduling**
  - **chooses the right* number of threads**

***right depends on some assumptions**

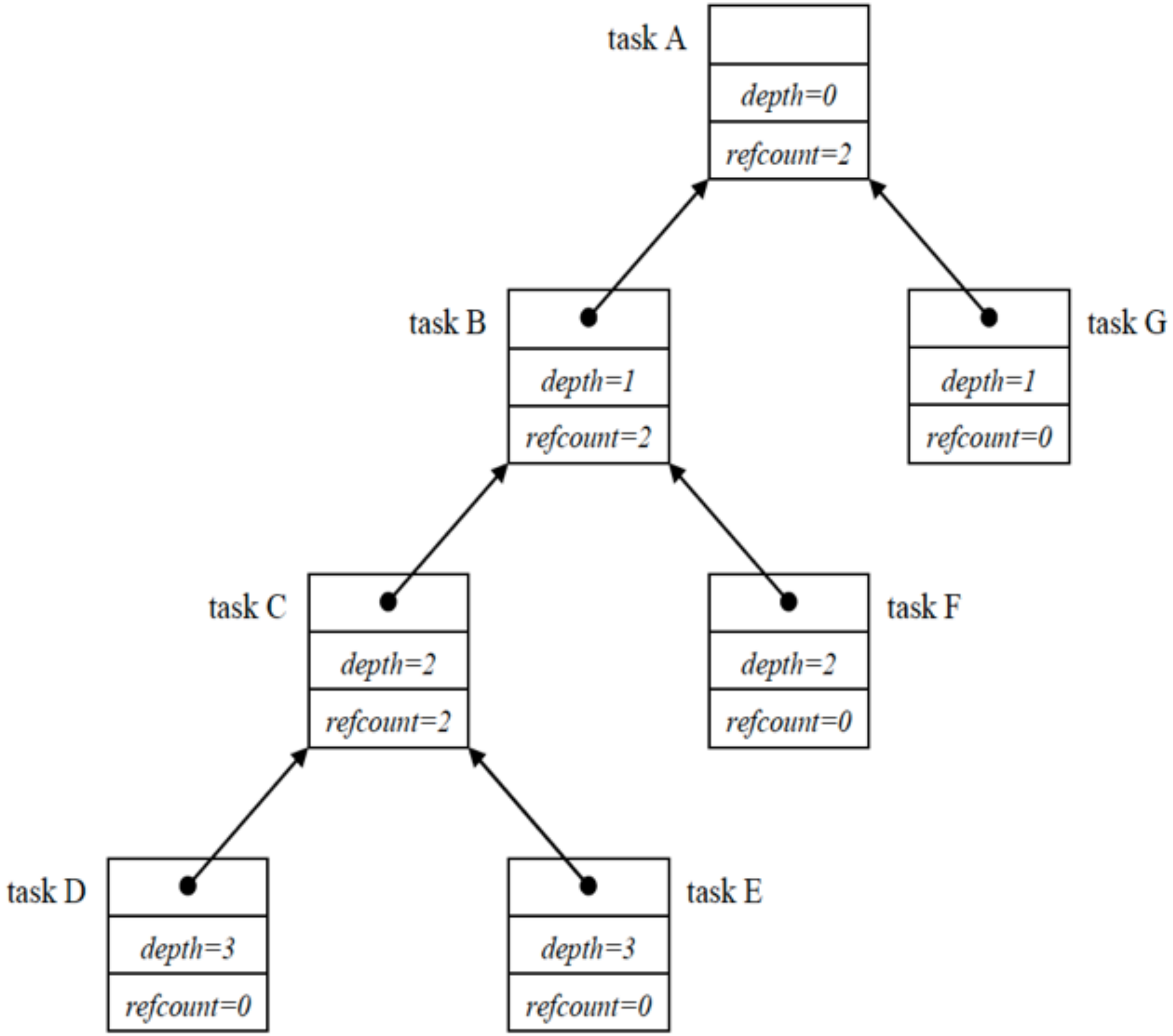
# Work Stealing Overview

- Task pool
  - —each thread maintains a pool of ready tasks
  - —if the pool is empty, try to steal from others
- $E[T_p] = O(T_1/P + T_\infty)$
  - —$T_p$ is the parallel time of the application
  - —$T_1$ is the sequential time of the application
  - —$T_\infty$ is the critical path length
- TBB uses randomized work stealing to map tasks to threads

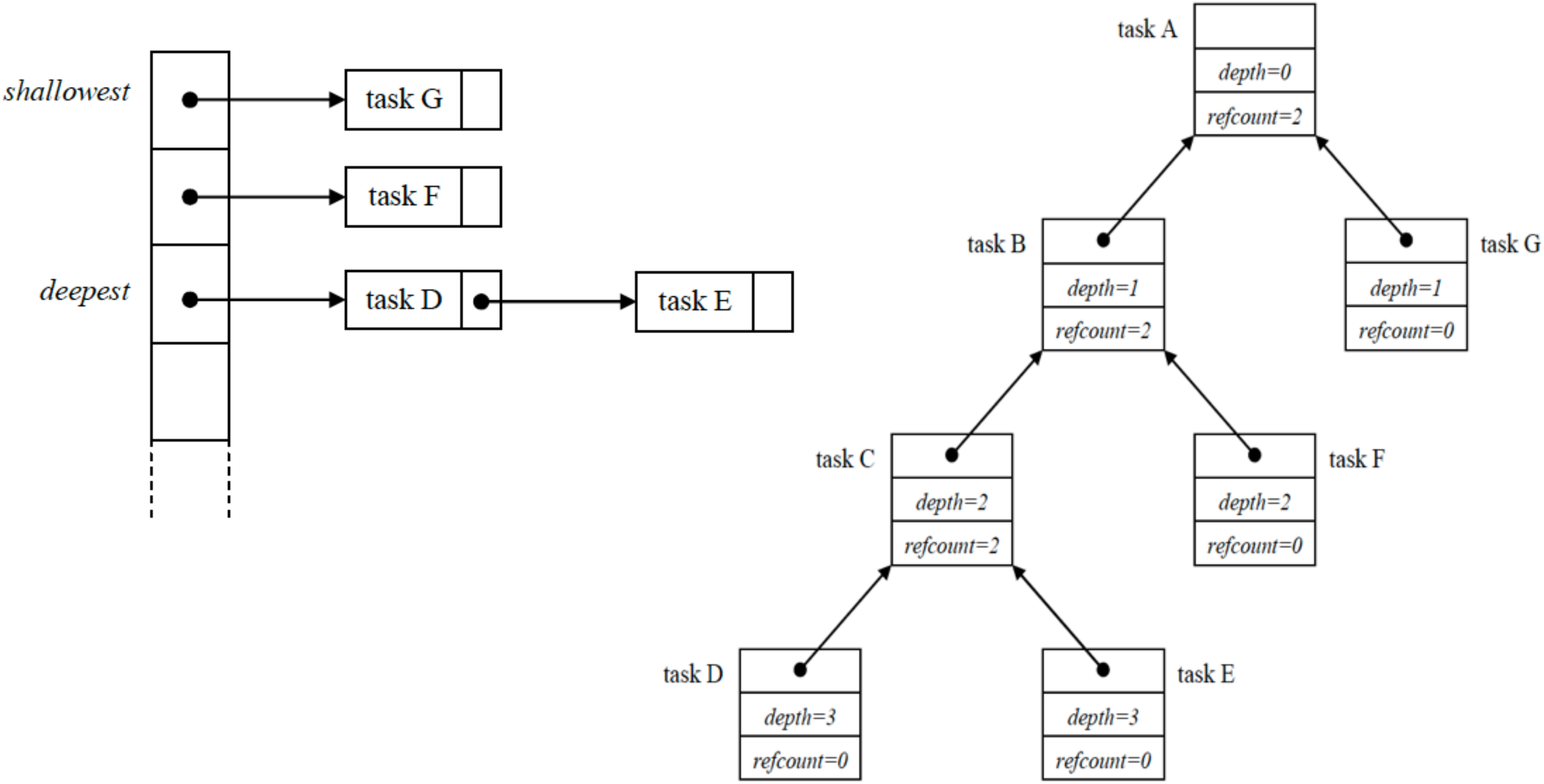
# Task Graph

- A directed graph
- Nodes are tasks
- Parents wait for children to complete
- Task node
  - —ref_count: number of active children
  - —depth: depth from the root
- Tasks can be generated dynamically

# Intermediate Task Graph for Fibonacci



# Local Task Pool for Task Graph



**A task is entered into a pool when ref_count = 0 (ready to run)**

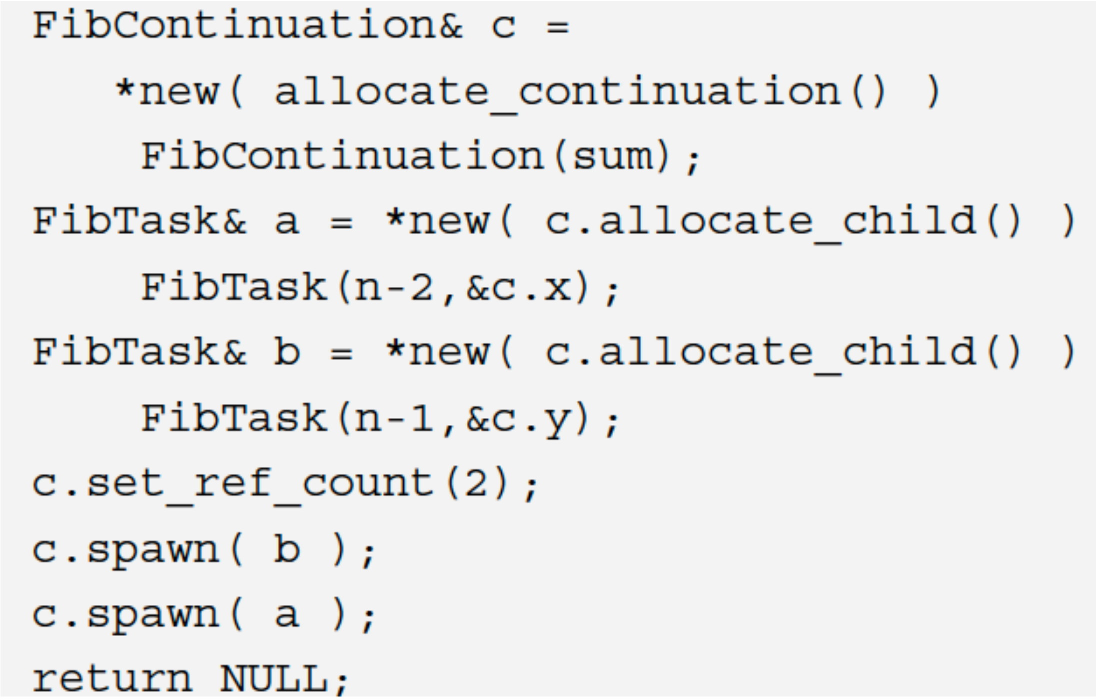
# Scheduling Strategy

- **Scheduling based on the task pool**
  - — if the shallowest task first executes
    - more tasks will be generated
    - may consume excessive memory
  - —if the deepest task first executes
    - maintains good data locality
    - limits parallelism
- **Breadth-first theft and depth-first work**
  - —first execute its local pool's <u>deepest</u> task
  - — If no task in its local pool, steal other pool's <u>shallowest</u> task

# Avoiding Stack Overflow

- Force a thread to steal only tasks deeper than its waiting one
  - —benefit: limits the growth of the stack
  - —cost: limits the choice of tasks to steal

- Specify continuation tasks
  - —replace the task itself as a continuation task
  - —return and free its stack space
  - —when children complete, it can be spawned
  - —only executing tasks are on the stack



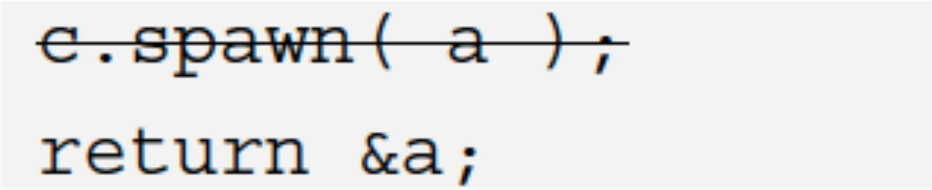
```
FibContinuation& c =
   *new( allocate_continuation() )
    FibContinuation(sum);
FibTask& a = *new( c.allocate_child() )
    FibTask(n-2,&c.x);
FibTask& b = *new( c.allocate_child() )
    FibTask(n-1,&c.y);
c.set_ref_count(2);
c.spawn( b );
c.spawn( a );
return NULL;
```

# Using Continuation Tasks

- Weaknesses
  - live state passed to child cannot reside in parent frame
  - creates an additional task object
    - overhead for fine-grain tasks
- Further optimizations to reduce overhead
  - scheduler bypass
  - task recycling

# Scheduler Bypass

- The execute function explicitly returns the next task to execute



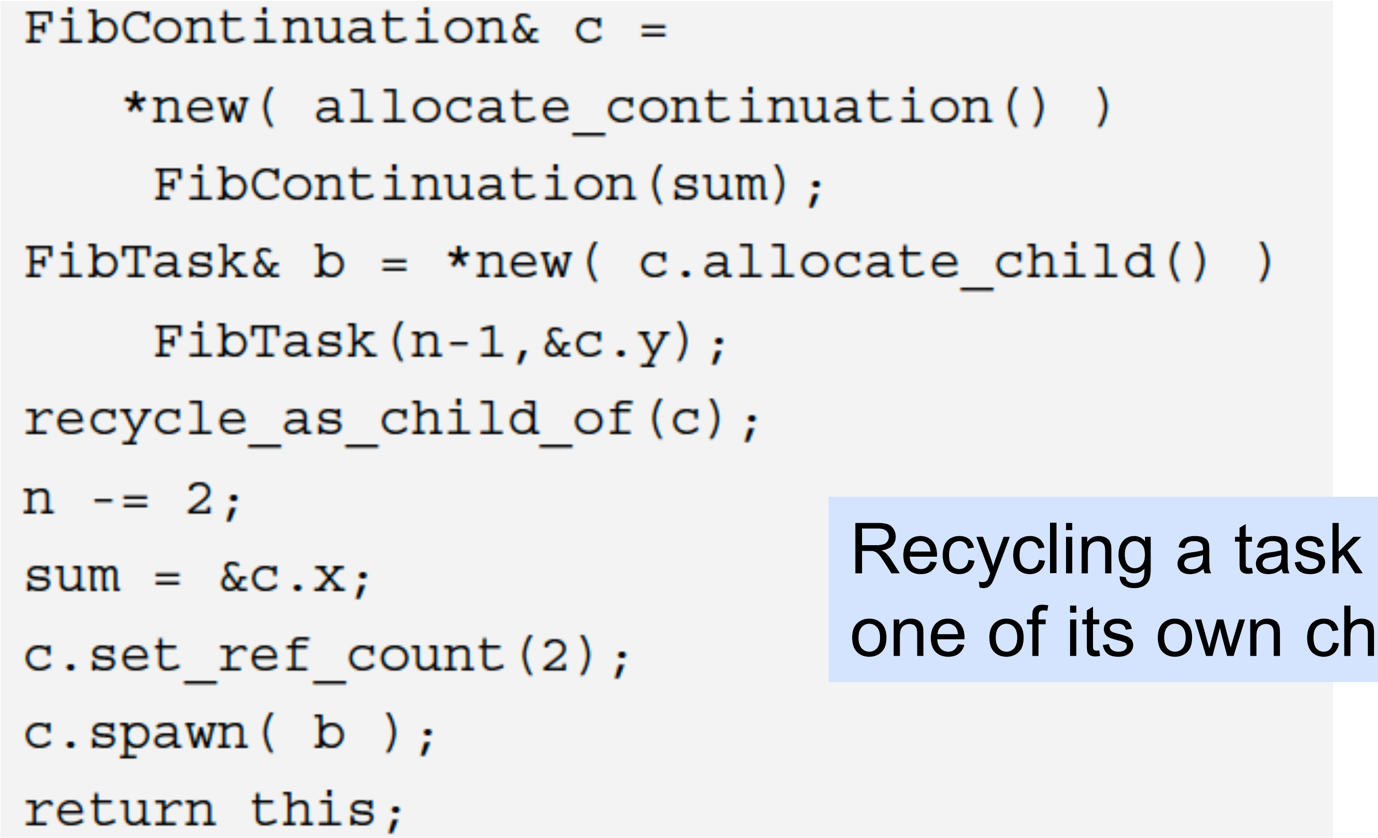
~~`c.spawn( a );`~~

`return &a;`

- No need to select a task if we already know what to execute
- Saves the push/pop on the deque

# Task Recycling

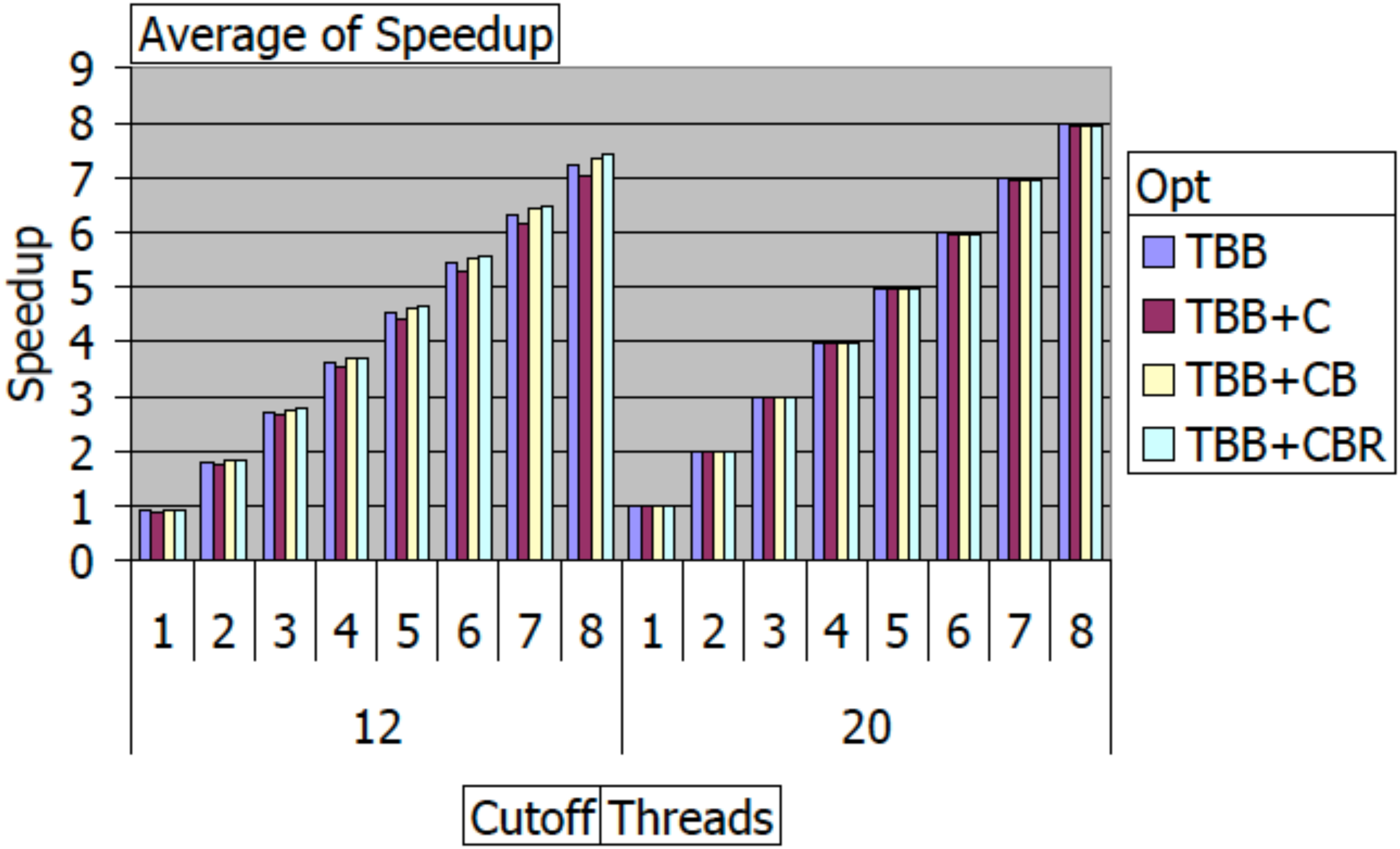
- **Normally a task is freed when it returns from execute**
- **Task object can live beyond the return**
- **Avoid repeated allocation and deallocation**



```
FibContinuation& c =
   *new( allocate_continuation() )
    FibContinuation(sum);
FibTask& b = *new( c.allocate_child() )
    FibTask(n-1,&c.y);
recycle_as_child_of(c);
n -= 2;
sum = &c.x;
c.set_ref_count(2);
c.spawn( b );
return this;
```

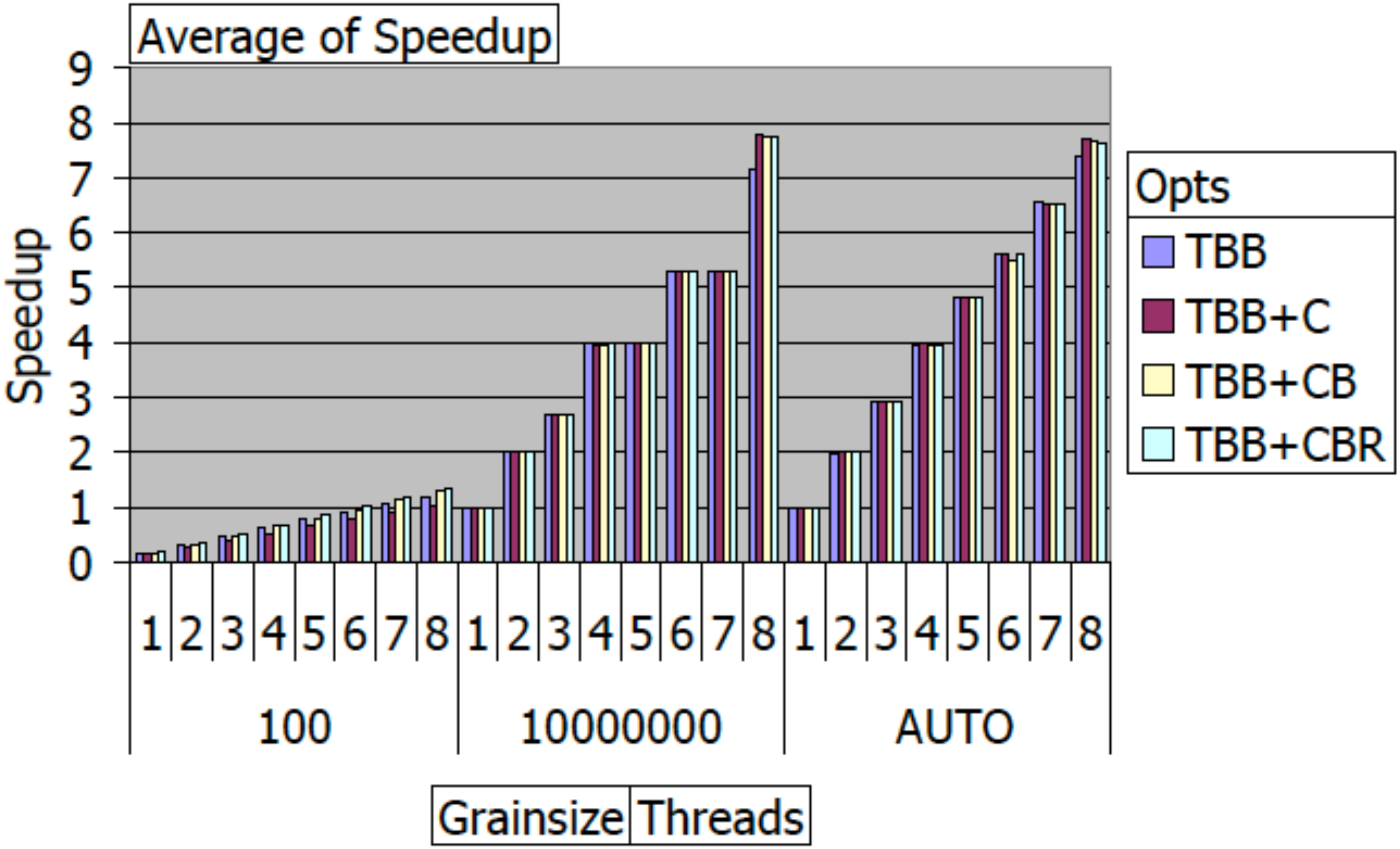
Recycling a task as one of its own children

# Fib: Scheduling Optimizations + Cutoff



# parallel_for: Grainsize vs. Auto Cutoff

**Iterates over 100M integers**



# Experimental Conclusions

- Tasks need sufficient grain size for performance
- Continuation passing can cause overhead with fine-grain tasks
- Scheduler bypass and task recycling can help reduce overhead

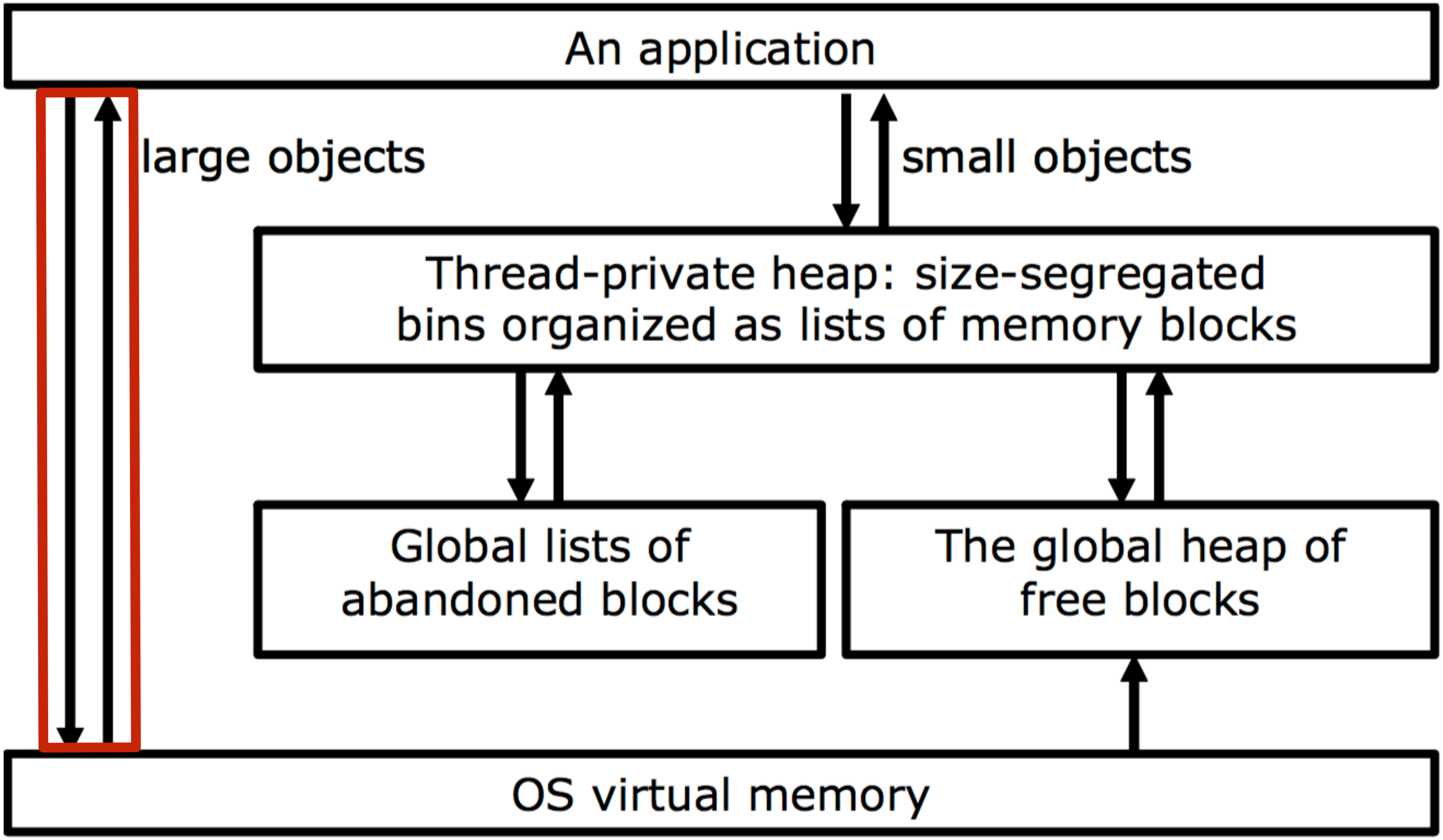
# Using the TBB Scheduler

- **Manual tuning is required**
  - select cutoff
  - manage continuations
  - recycle tasks
  - mechanics: use APIs to do it
    - no compiler support
- **Comparing with Cilk/Cilk++**
  - TBB is portable and flexible, but lacks the convenience

# TBB Memory Allocator

- Goals of serial allocators
  - use space efficiently
  - minimize CPU overhead
- Additional goals of allocators for multithreaded code
  - scalable to large thread counts with low overhead
    - avoid synchronization bottlenecks
  - data locality (prevent false sharing)

# TBB Allocator: Large Blocks


An application

large objects

small objects

Thread-private heap: size-segregated bins organized as lists of memory blocks

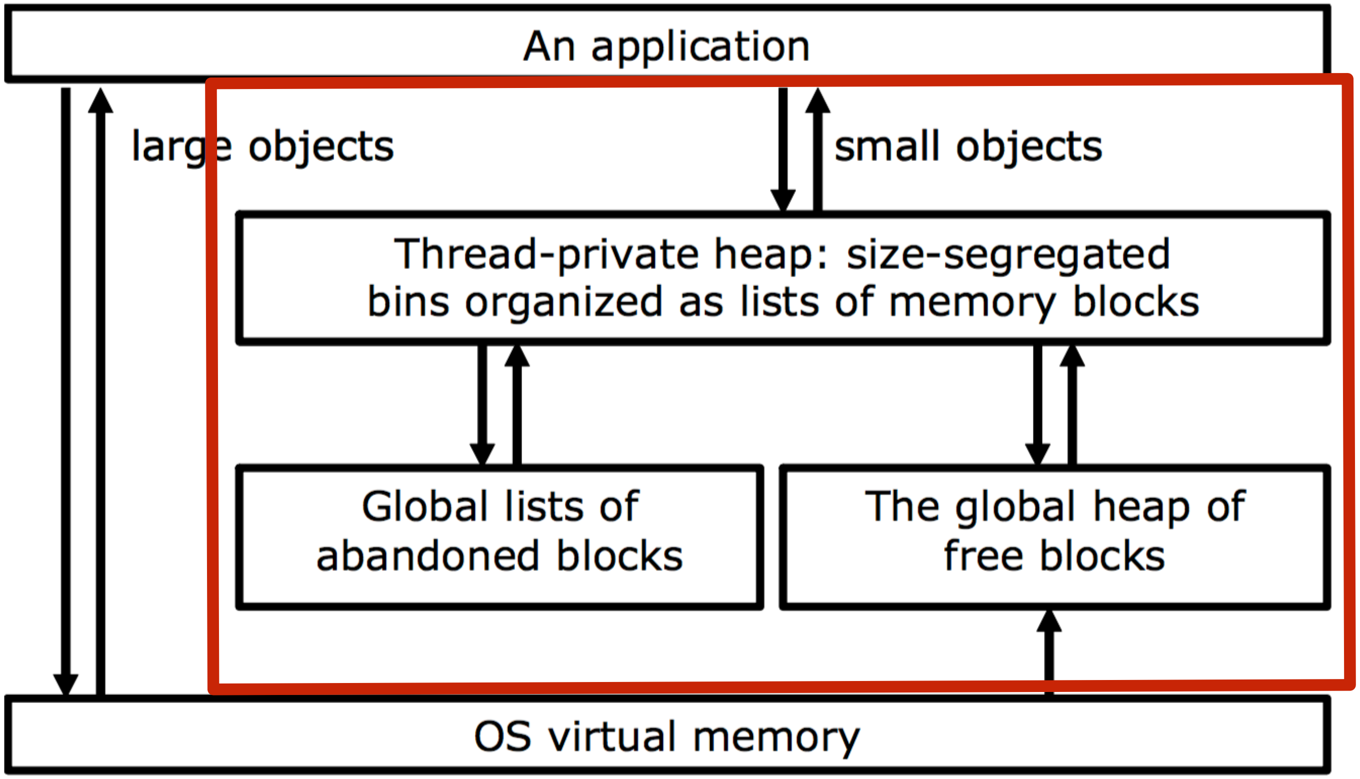
Global lists of abandoned blocks

The global heap of free blocks

OS virtual memory

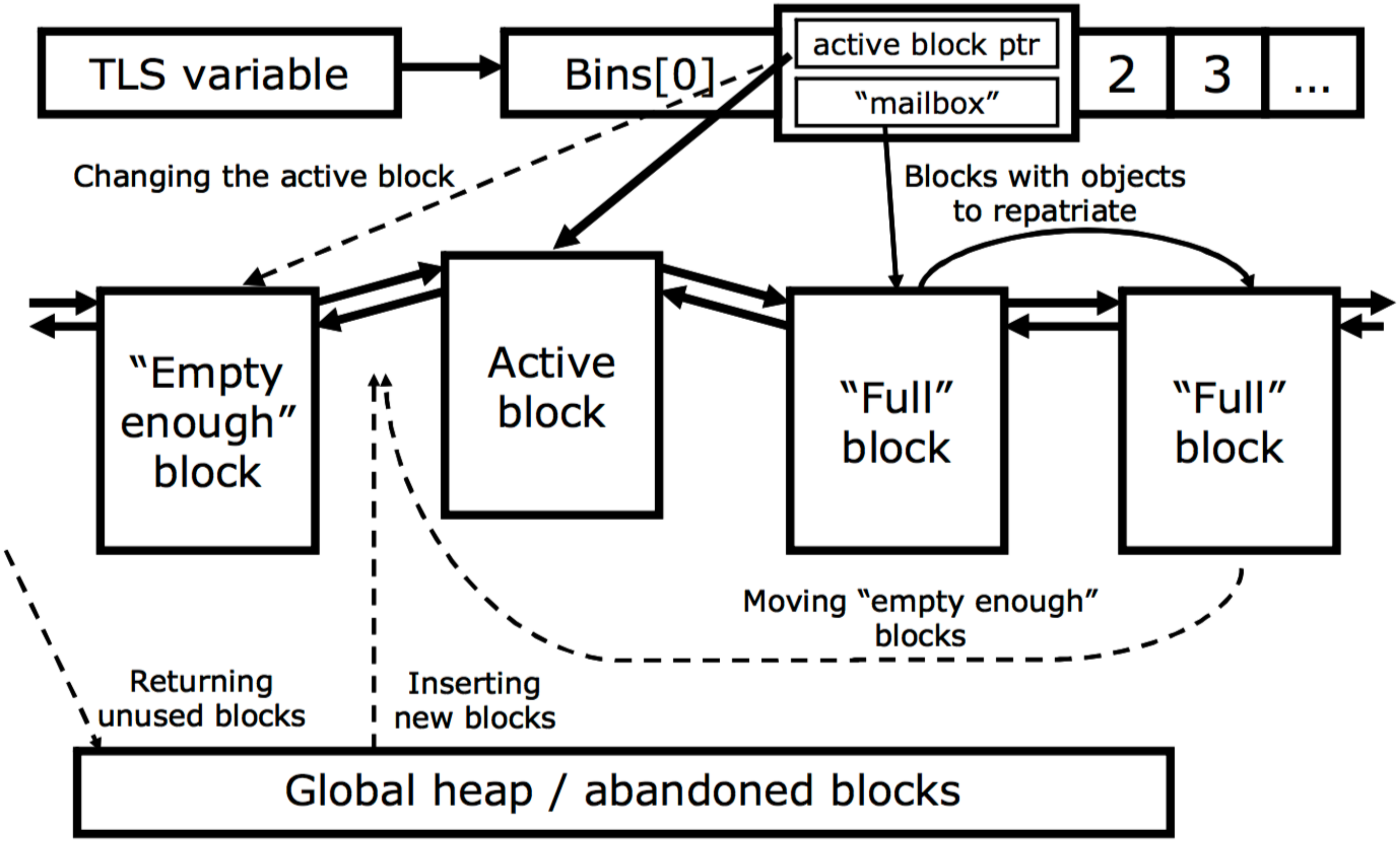
- **Allocate large objects directly from OS**
  - **large = ~8K**

# TBB Allocator: Small Blocks



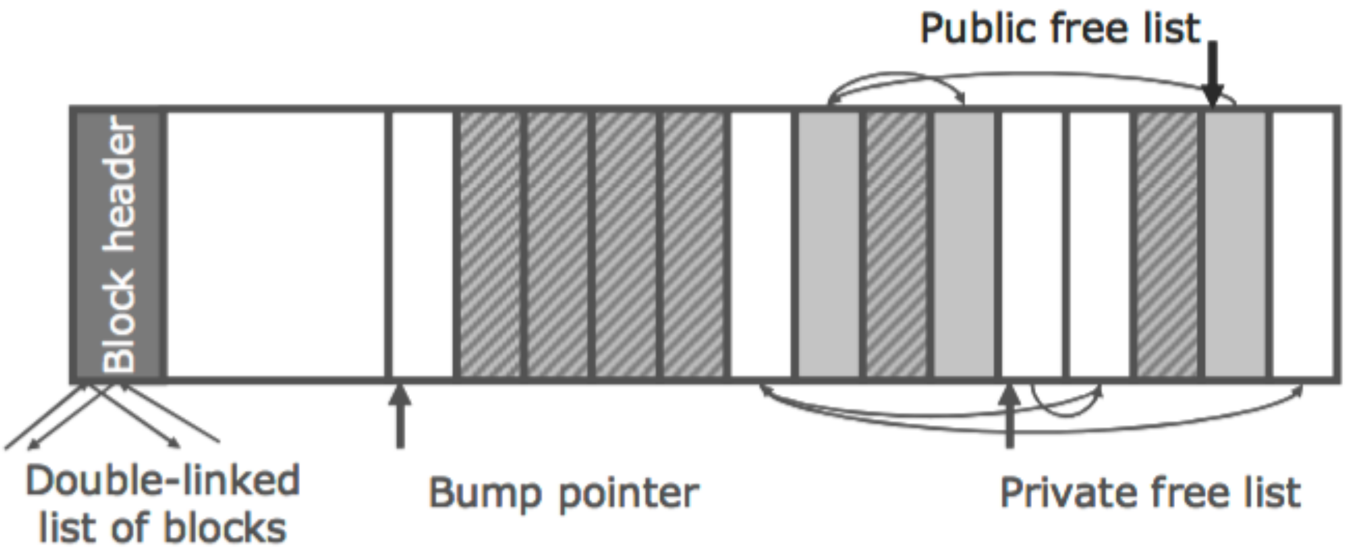
- Minimize requests and maximize reuse
  - —request memory from OS in 1MB chunks
  - —divide each chunk into 16K- byte aligned blocks
  - —never return memory to OS
- Request new blocks only when none available in both local and global heap

# TBB Allocator: Managing Private Heap



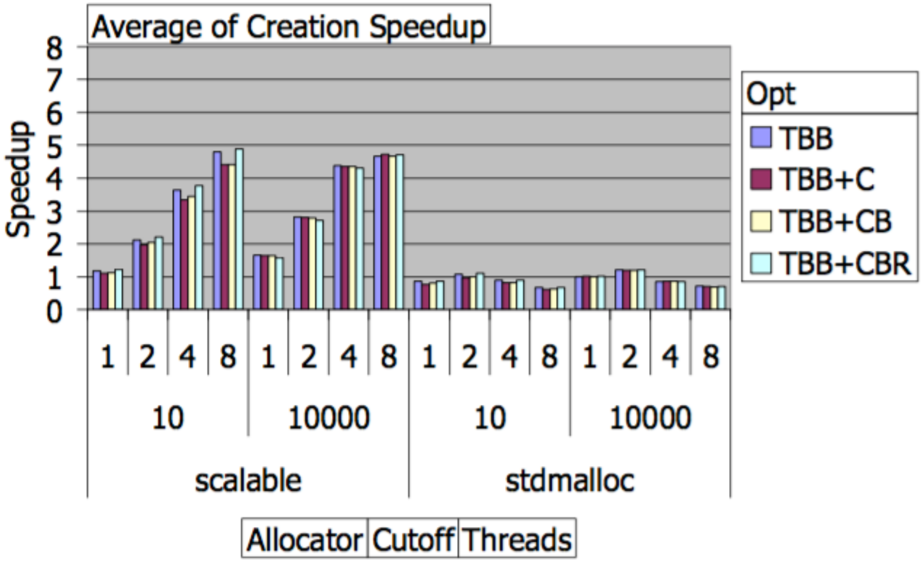
- **Thread-private heaps**
  - —reduces down synchronization overhead and false sharing
- **Heap is a set of bins**
  - —doubly linked list of blocks containing objects of similar size
  - —no per object header needed: bin has uniform size objects

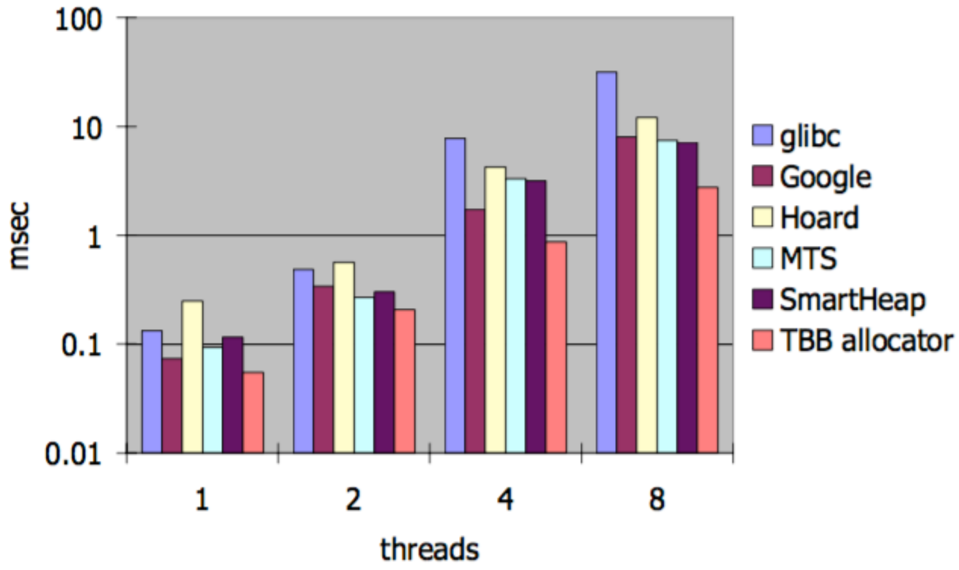
# TBB Allocator: Memory Block

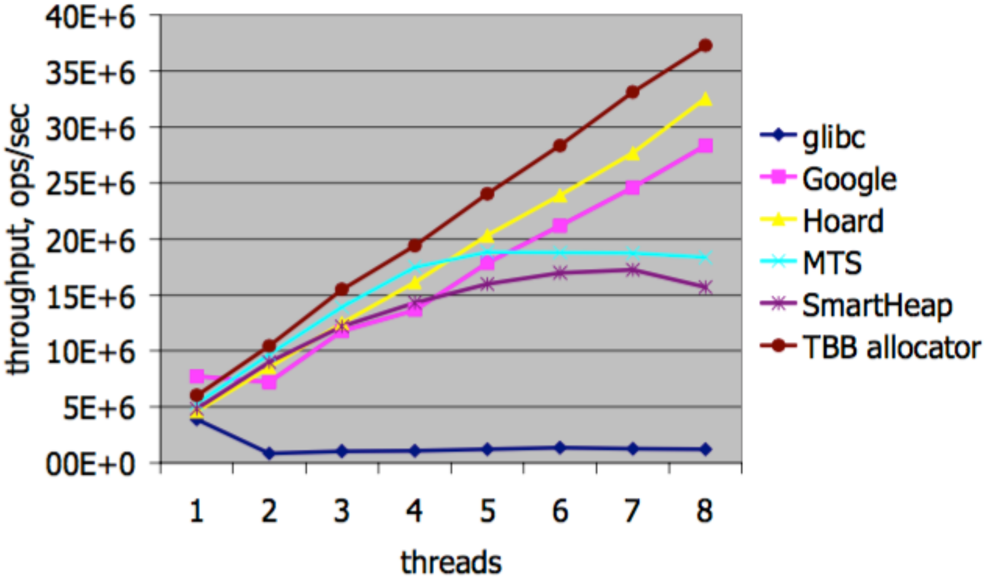


- Block header contains size information
- Objects are tightly packed in the block
- Separate free lists
  - —private free list: no synchronization needed
  - —public free list: other threads return items to my block
- Allocate from private free list when available
- Check public free list if empty

# TBB Allocator Performance







# TBB Scheduler

- **Automatically balances the load across cores**
  - —work stealing
- **Schedules tasks to exploit the cache locality of applications**
- **Avoids over-subscription of resources**
  - —uses tasks for scheduling
  - —chooses the right* number of threads

***Issues with TBB on Blue Gene/Q**

- TBB may create more software threads than hardware threads
- TBB doesn’t interact well with bound threads

# What is OpenMP?

**Open specifications for Multi Processing**

- An API for explicit multi-threaded, shared memory parallelism
- Three components
  - compiler directives
  - runtime library routines
  - environment variables
- Higher-level than library-based programming models
  - implicit mapping and load balancing of work
- Portable
  - API is specified for C/C++ and Fortran
  - implementations on almost all platforms
- Standard

# OpenMP Targets Ease of Use

- OpenMP does not require that single-threaded code be changed for threading
  - enables incremental parallelization of a serial program
- OpenMP only adds compiler directives
  - pragmas (C/C++); significant comments in Fortran
    - if a compiler does not recognize a directive, it can ignore it
  - simple & limited set of directives for shared memory programs
  - significant parallelism possible using just 3 or 4 directives
    - both coarse-grain and fine-grain parallelism
- If OpenMP is disabled when compiling a program, the program will execute sequentially

# Components of OpenMP

## *Directives*

- ***Parallel regions***
- ***Work sharing***
- ***Synchronization***
- ***Data-sharing attributes***
  - ***private***
  - ***firstprivate***
  - ***lastprivate***
  - ***shared***
  - ***reduction***
- ***Orphaning***

## *Environment variables*

- ***Number of threads***
- ***Scheduling type***
- ***Dynamic thread adjustment***
- ***Nested parallelism***

## *Runtime environment*

- ***Number of threads***
- ***Thread ID***
- ***Dynamic thread adjustment***
- ***Nested parallelism***
- ***Timers***
- ***API for locking***

# A First OpenMP Example: Vector Sum

### For-loop with independent iterations

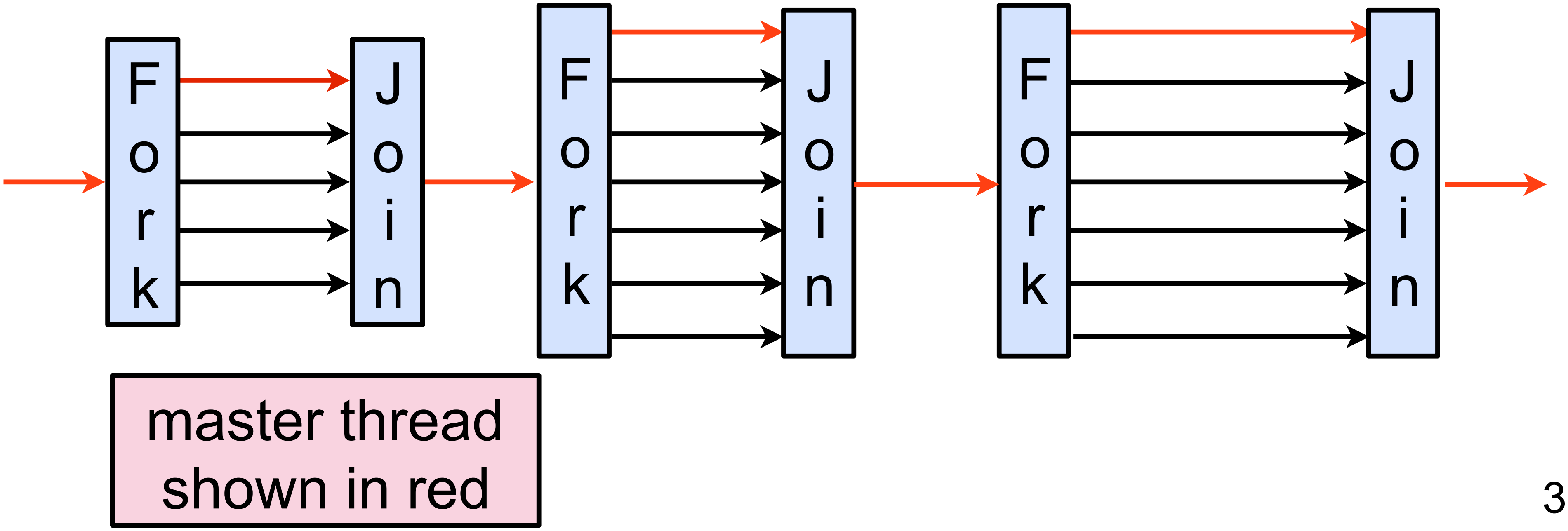
```
  for (i = 0; i < n; i++)
    c[i] = a[i] + b[i];
```

### For-loop parallelized using an OpenMP pragma

```
#pragma omp parallel for  \
        shared(n, a, b, c)\
        private(i)
for (i = 0; i < n; i++)
    c[i] = a[i] + b[i];
```

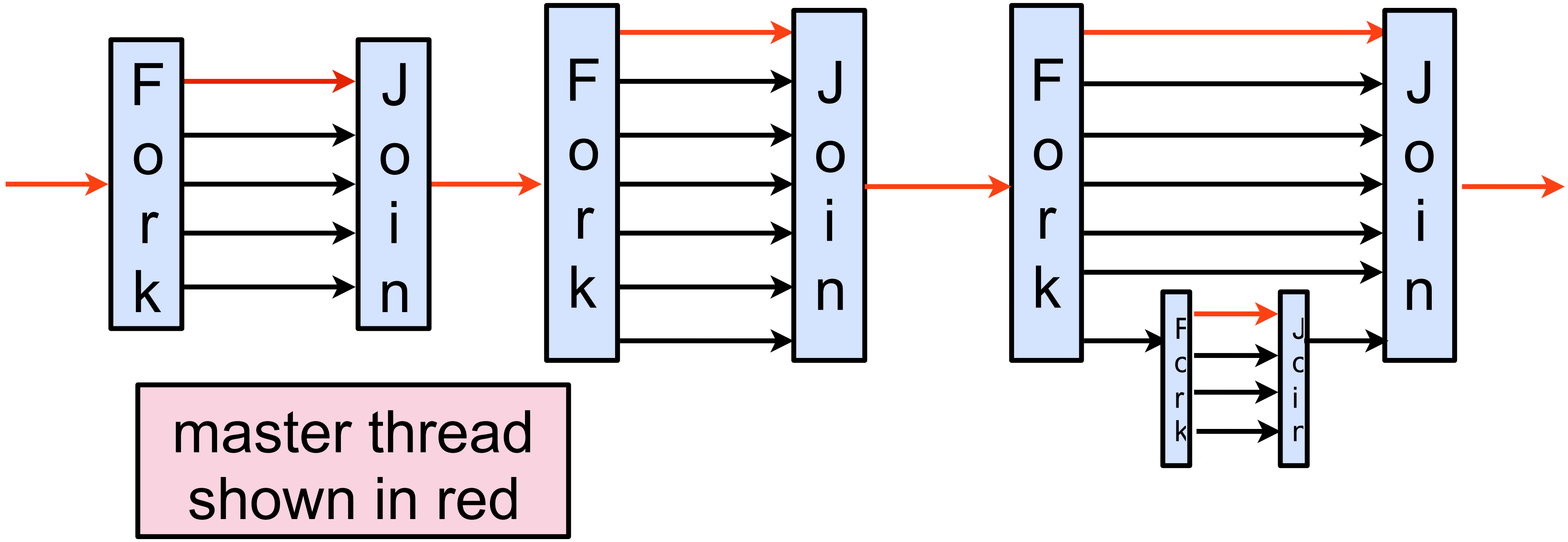
# Fork-Join Parallelism in OpenMP

- OpenMP program begins execution as a single master thread
- Master thread executes sequentially until 1st parallel region
- When a parallel region is encountered, the master thread …
  - —creates a group of threads
  - —becomes the master of this thread group
  - —is assigned thread id 0 within the group



# Nested Parallelism

- **Nested parallelism enabled using the `OMP_NESTED` environment variable**
  - `OMP_NESTED` = TRUE → nested parallelism is enabled
- **Each parallel directive creates a new team of threads**



# A Simple Example Using `parallel` and `for`

**Program**

```
void main()  {
#pragma omp parallel num_threads(3)
{
  int i;
  printf(“Hello world\n”);
  #pragma omp for
  for (i = 1; i <= 4; i++) {
     printf(“Iteration %d\n”,i);
  }
  printf(“Goodbye world\n”);
}
}
```

Implicit barrier at end parallel region

**Output**

```
Hello world
Hello world
Hello world
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Goodbye world
Goodbye world
Goodbye world
```

# Fibonacci using OpenMP Tasks

```
int fib ( int n )
    {
        int x,y;
        if ( n < 2 ) 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;
    }
```

```
int main (int argc, char **argv)
{
    int n, result;
    n = atoi(argv[1]);
#pragma omp parallel
{
#pragma omp single
{
    result = fib(n);
}
}
    printf("fib(%d) = %d\n",
       n, result);
}
```

create team of threads to execute tasks

only one thread performs the outermost call

suspends parent task until children finish

# OpenMP for Manycore Systems

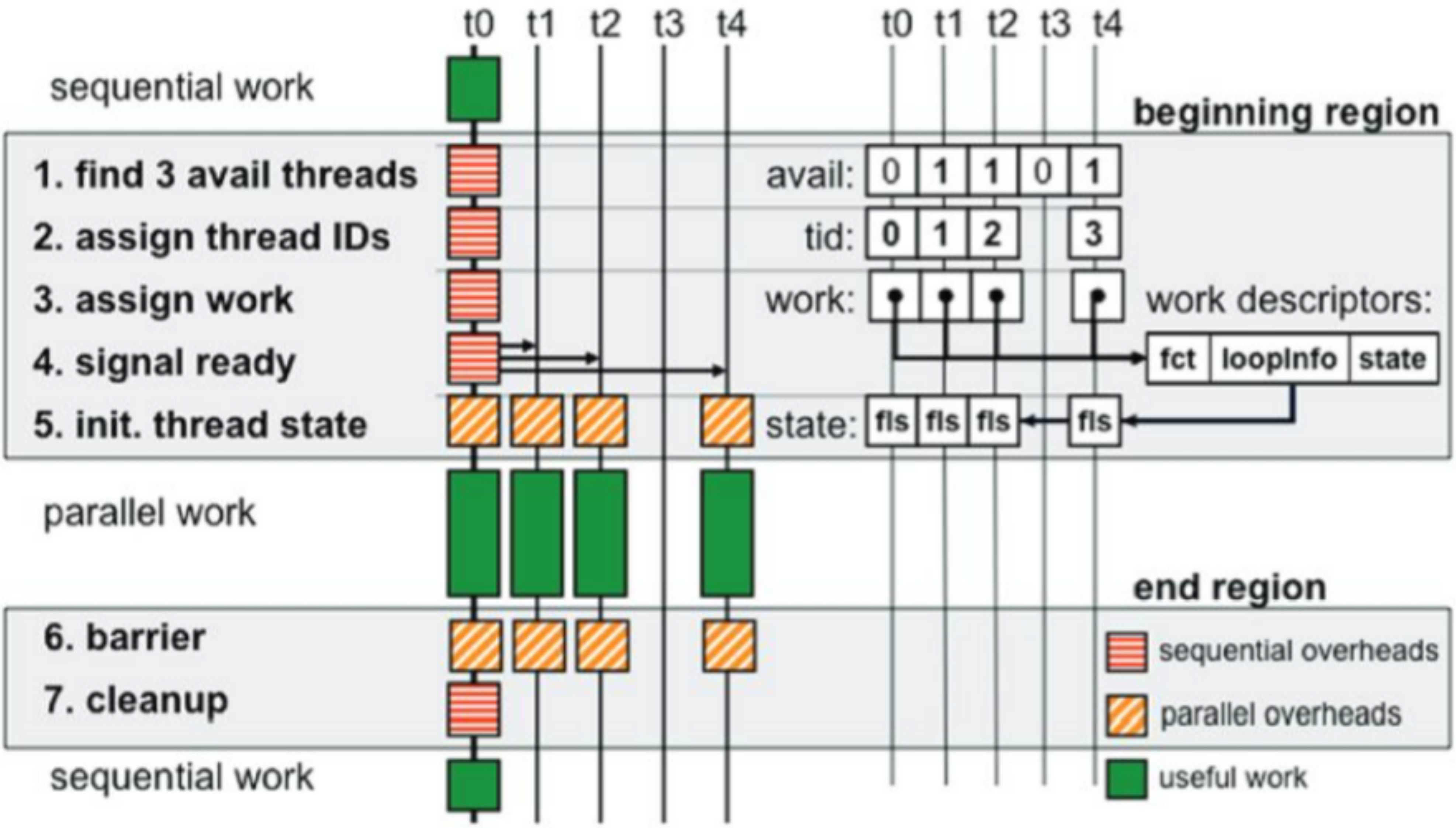
- **Large thread counts are becoming increasingly common**
- **Need low overhead OpenMP runtime to efficiently exploit many fine-grain threads**
- **Algorithm and data structure choices can reduce OpenMP overheads**
  - —improved design of runtime algorithms and data structures reduces overhead by 5x for 64 OpenMP threads
  - —strategies scale well with number of threads

# OpenMP Overhead Concerns

**Assembling a thread team for an OpenMP parallel construct is costly**

- **Sequential overhead: work by master thread**
  - identify available threads
  - create a team of threads
  - assign work to threads in team
  - signal that the parallel region can start
- **Parallel overhead: costs incurred by each thread in team**
  - initialize state of each thread
  - synchronize threads upon completion

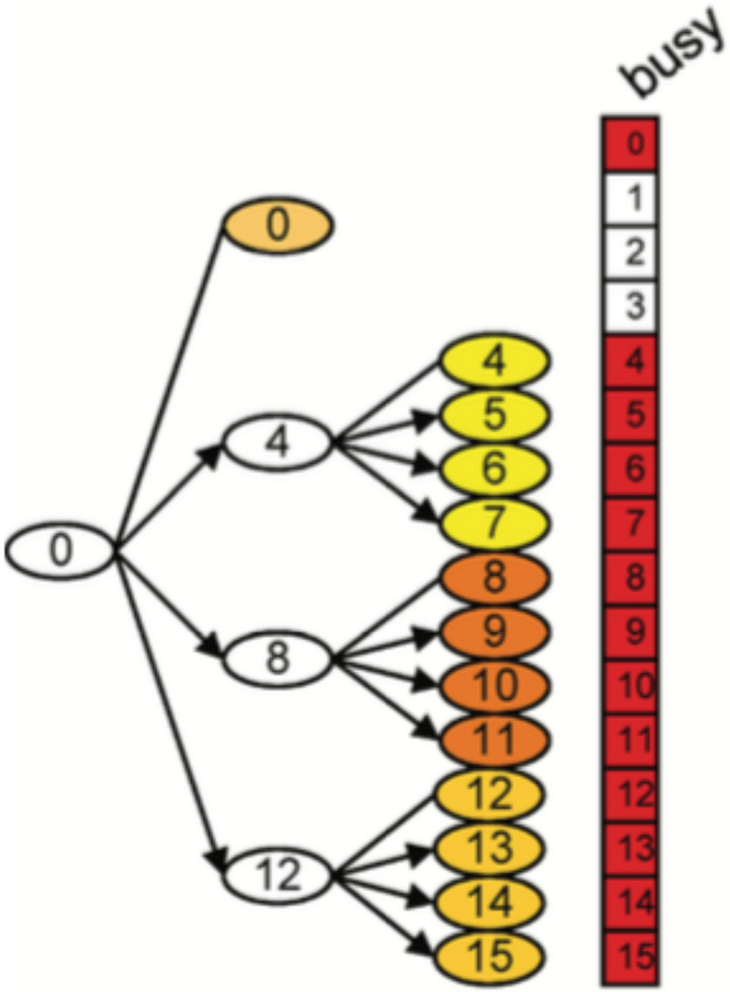
# Creating a Parallel Region



# Allocating Threads for a Team

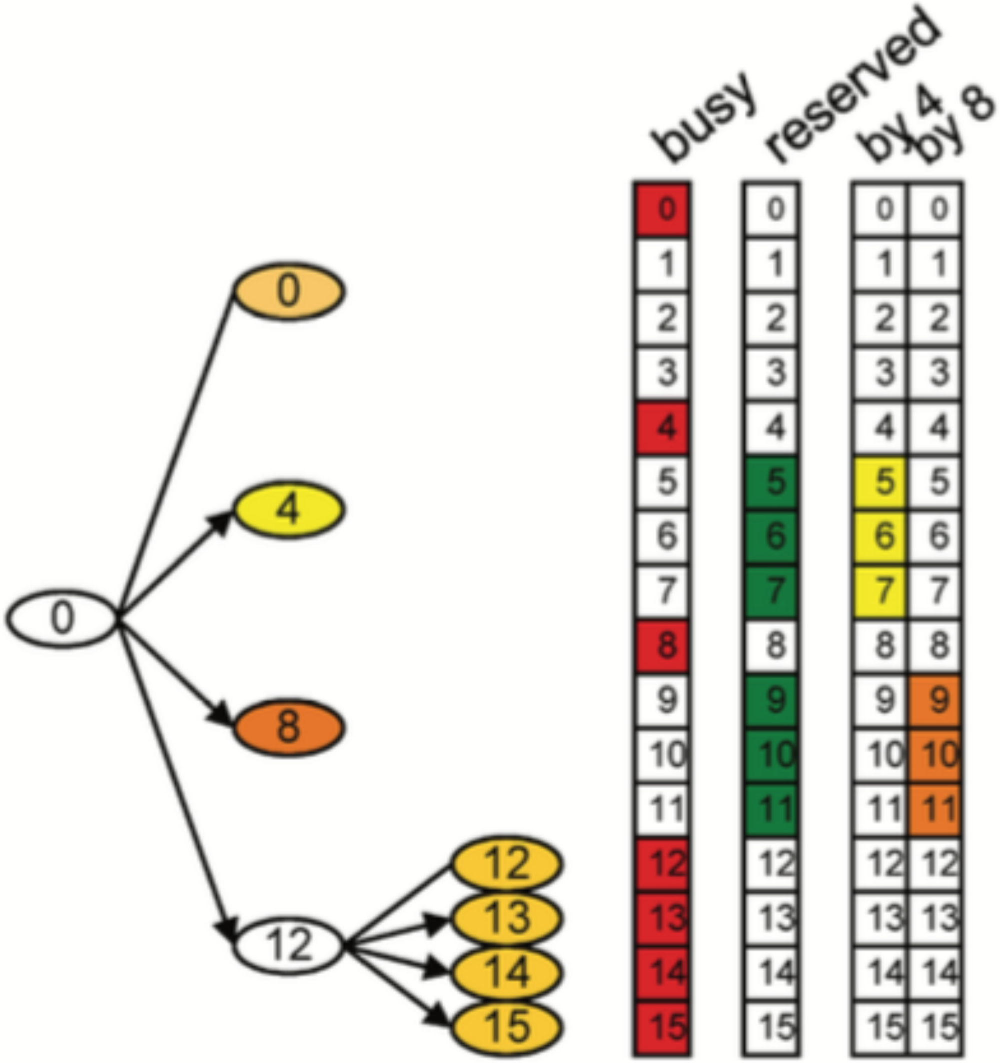
## Consider the Following

- System with 16 threads: t0-t15
- Master thread t0 creates worker threads t4, t8, and t12
- Each worker encounters a parallel region and becomes a master thread
  - —teams with masters t4, t8, and t12 each allocate 3 workers
- Every thread is busy except t1-t3



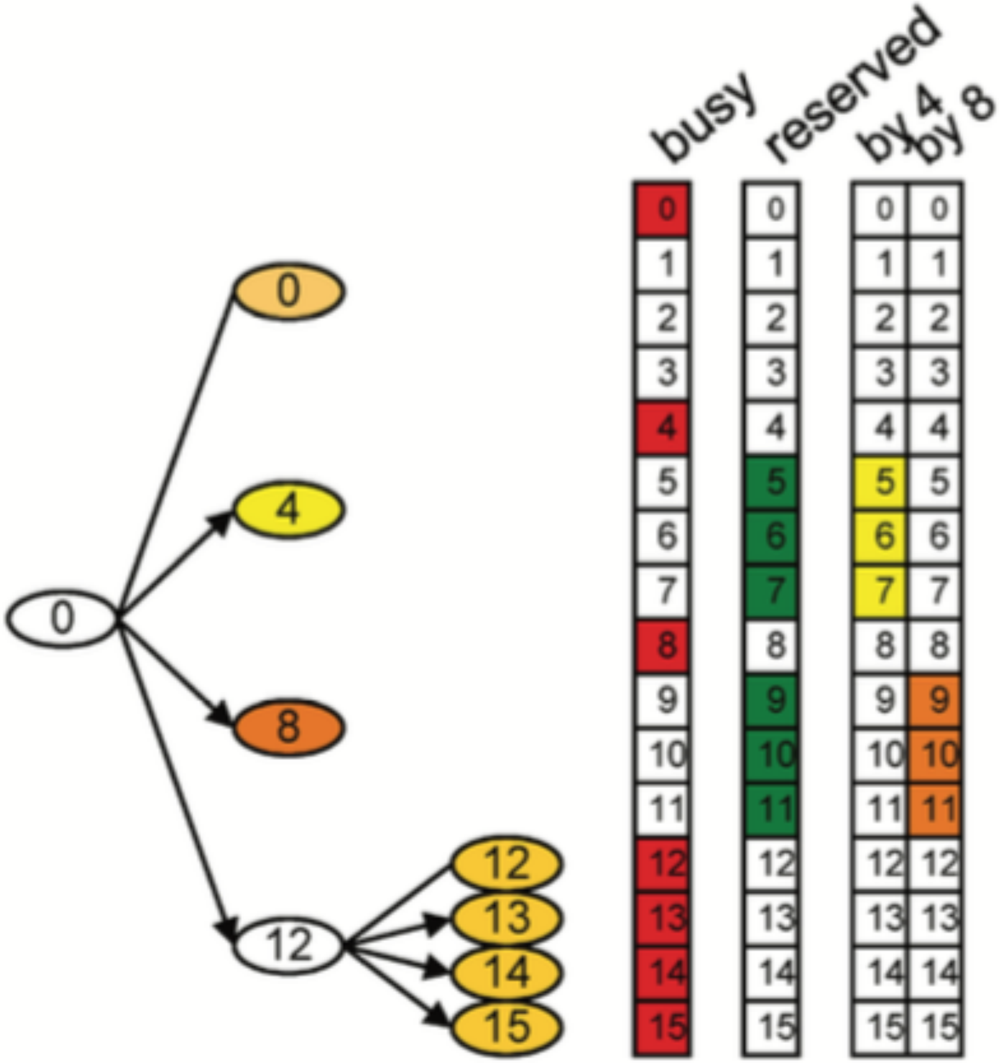
# Fast Thread Allocation using Bit Vectors - I

- **t4 & t8 now join and deallocate their workers**
- **Goal: reduce overhead of allocating threads**
  - —programs usually allocate the same thread count repeatedly
  - —can save time by reserving the worker threads after workers are deallocated
  - —maintain a global reserved bit vector
    - – new teams can be allocated without interfering with the recently deallocated threads



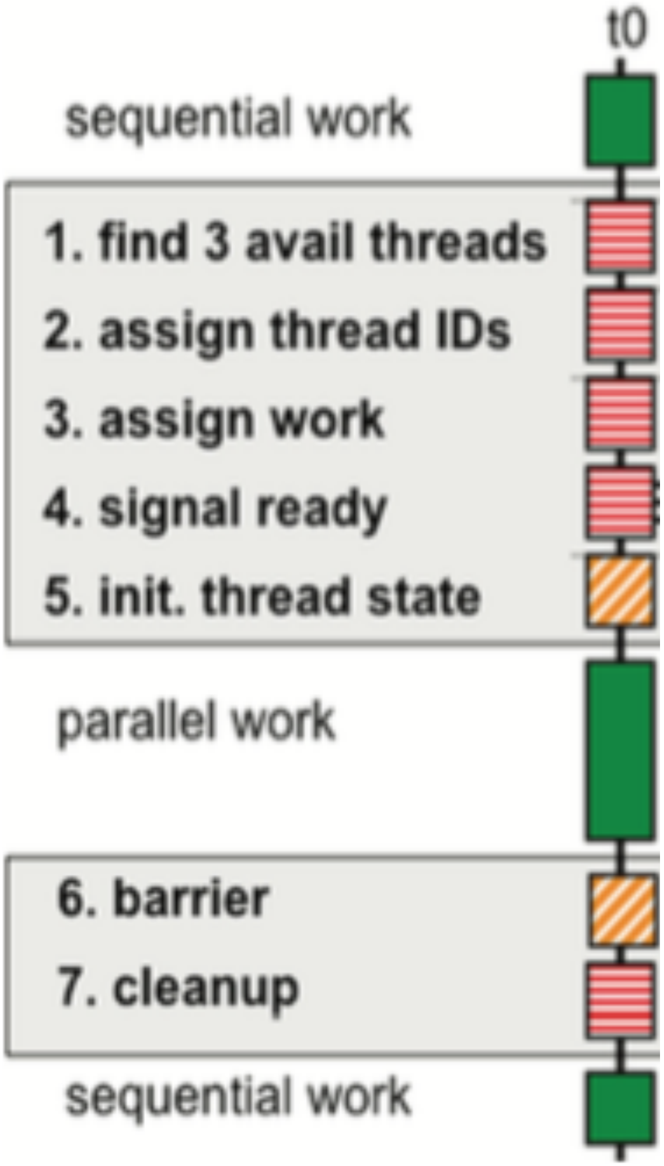
# Fast Thread Allocation using Bit Vectors - II

- t0 forks and checks the union of busy and reserved bit vectors
- t1-t3 are free so t0 allocates 3 workers to those threads
- t4 then forks and checks if its previous allocation has been invalidated
- If the past allocation is intact the team of t4-t7 can continue without allocation overhead



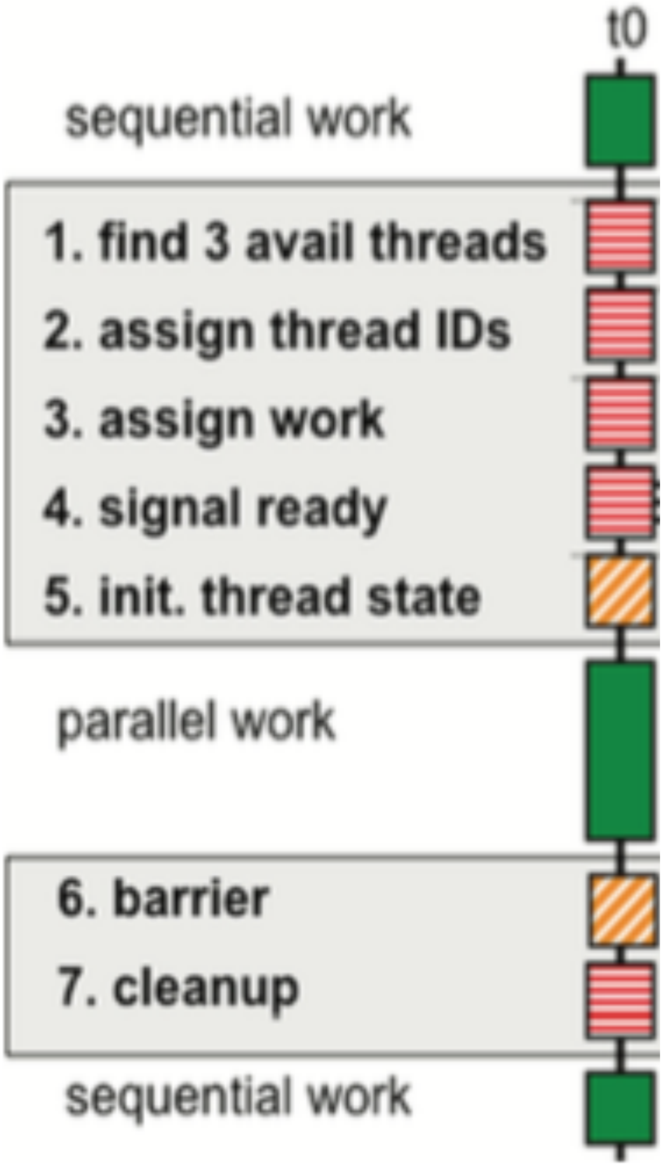
# Allocation and Work Descriptor Caching

- **Thread Allocation Caching (Step 1)**
  - —allocation overhead can be eliminated by determining if a past allocation is valid
- **Work Description Caching (Steps 2-3)**
  - —thread ID (TID) and work-descriptor ID (WID) assignments can be reused
  - —prior valid thread allocation will have the same TID and WID mapping so they are stored in a thread's local cache upon deallocation
  - —only modify content of shared work descriptor



# Signaling and Interface Refinements

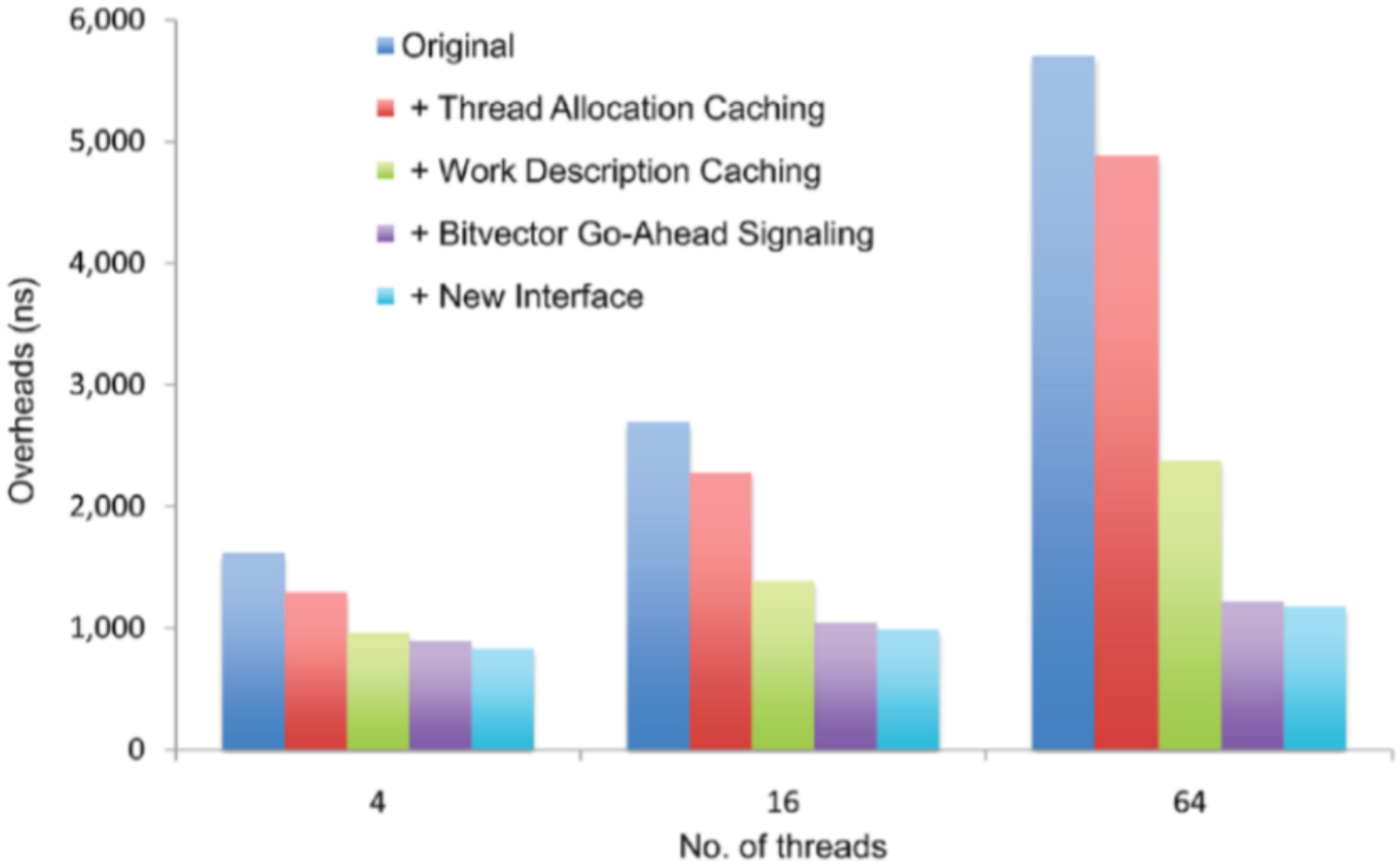
- **Bit Vector Go-Ahead Signaling (Step 4)**
  - —use a global "activate" bit vector to reduce the overhead of workers receiving the work-ready signal from their master
    - – XOR in bits for a thread set to activate all threads in set
- **Improved Application Runtime Interface**
  - —streamlining the interface between the OpenMP runtime/application reduces the overhead of localizing thread private variables and communication of default settings to the runtime



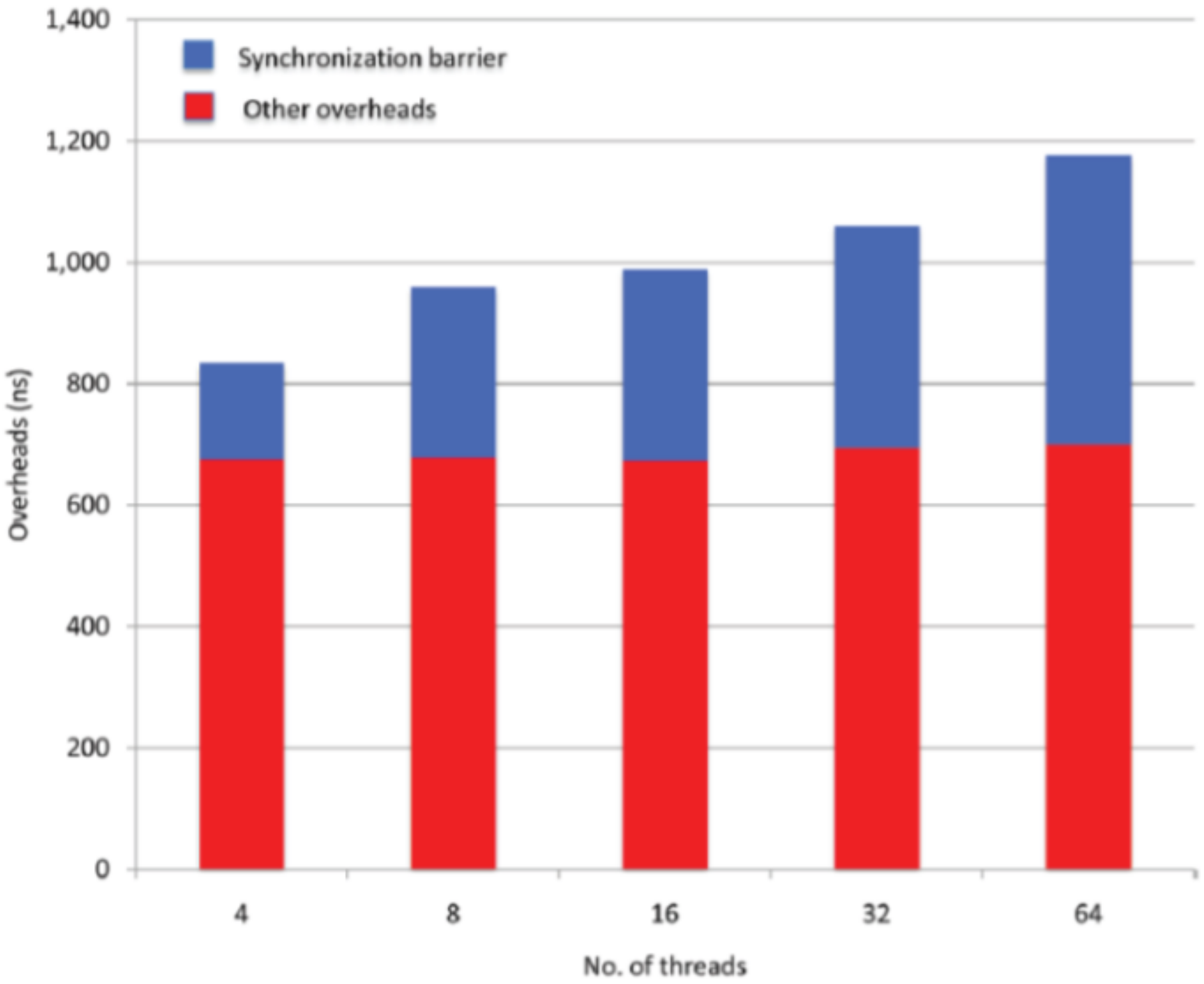
# Evaluation of OpenMP Overhead

- **Benchmarks**
  - Edinburgh Parallel Computing Center benchmark suite
    - designed to measure OpenMP overheads caused by synchronization and loop scheduling
- **System**
  - single Blue Gene/Q node
    - use all HW threads: 4 SMT threads per core
- **Runtime**
  - modified runtime with all optimizations
- **Expectation**
  - reductions in runtime overhead for larger thread counts

# Overhead vs. Thread Count



# Overhead Breakdown vs. Thread Count



# References

- Alexey Kukanov, Michael J. Voss. The Foundations for Scalable Multi-core Software in Intel Threading Building Blocks. Intel Technology Journal, Volume 11, Issue 4, 2007.
- A. E. Eichenberger and K. O'Brien. Experimenting with low-overhead OpenMP runtime on IBM Blue Gene/Q. IBM J. Res. Dev. 57, 1 (January 2013), 91-98. DOI=10.1147/JRD. 2012.2228769