

OpenMP 3.0:

Future Features and
Implementation in Open64

Presented by Deepak Eachempati
@ Open64 Tutorial/Workshop 2011

Agenda

- ▶ Brief Overview of OpenMP
- ▶ OpenMP 3.0 Additions
- ▶ Supporting OpenMP 3.0
 - ▶ Compiler Front-end and Back-end implementation
 - ▶ Runtime Implementation
- ▶ Experimental Results
- ▶ Status

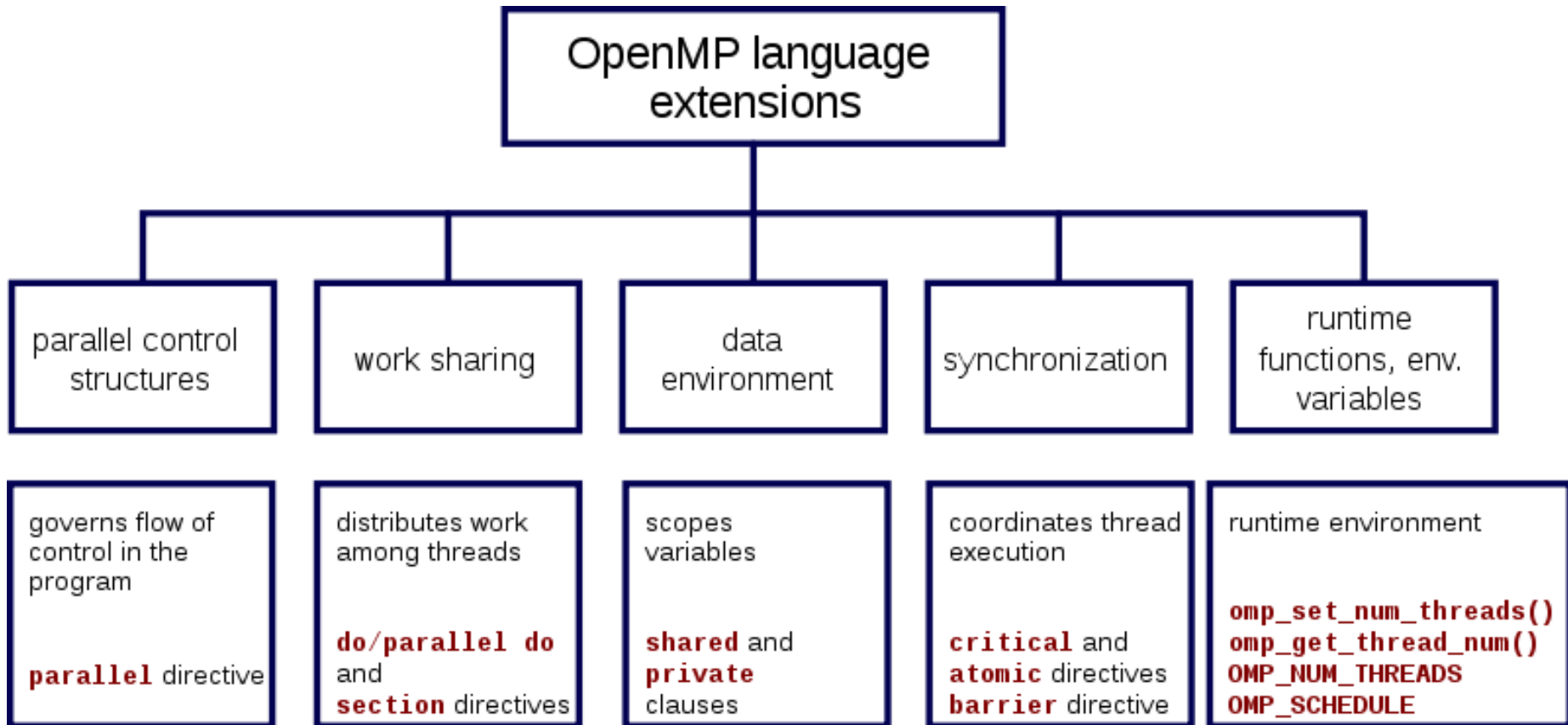
Agenda

- ▶ **Brief Overview of OpenMP**
- ▶ OpenMP 3.0 Additions
- ▶ Supporting OpenMP 3.0
 - ▶ Compiler Front-end and Back-end implementation
 - ▶ Runtime Implementation
- ▶ Experimental Results
- ▶ Status

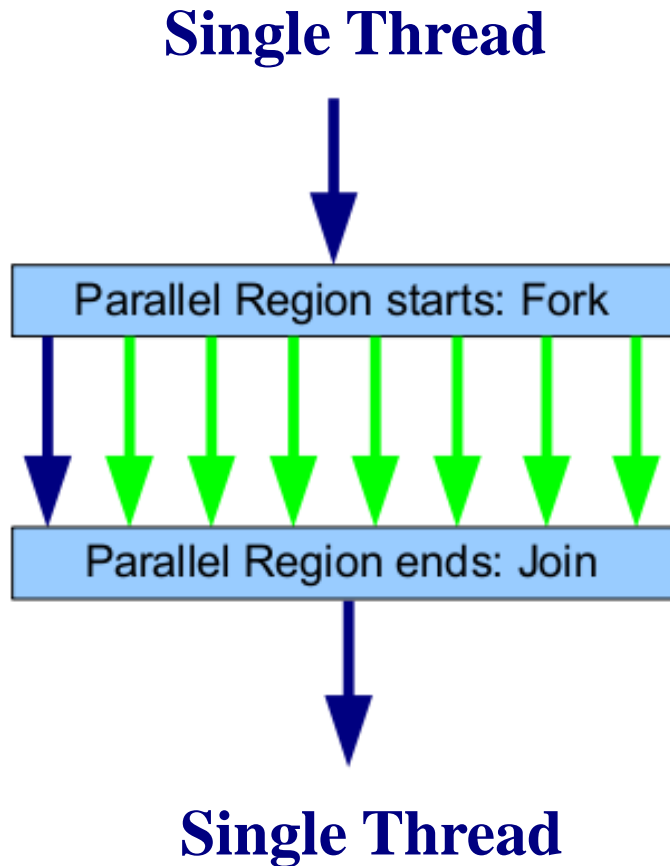
What is OpenMP?

- ▶ De-facto standard Application Programming Interface to write shared memory applications in C, C++ and Fortran
- ▶ Mainly comprises of:
 - ▶ Compiler Directives
 - ▶ Runtime Routines
 - ▶ Environment Variables
- ▶ OpenMP ARB Members:
 - ▶ Permanent: AMD, Cray, Fujitsu, HP, IBM, Intel, NEC, PGI, Oracle, Microsoft, TI, CAPS
 - ▶ Auxiliary: ANL, ASC/LLNL, cOMPunity, EPCC, LANL, NASA, RWTH Aachen University

Before OpenMP 3.0



Parallel Constructs



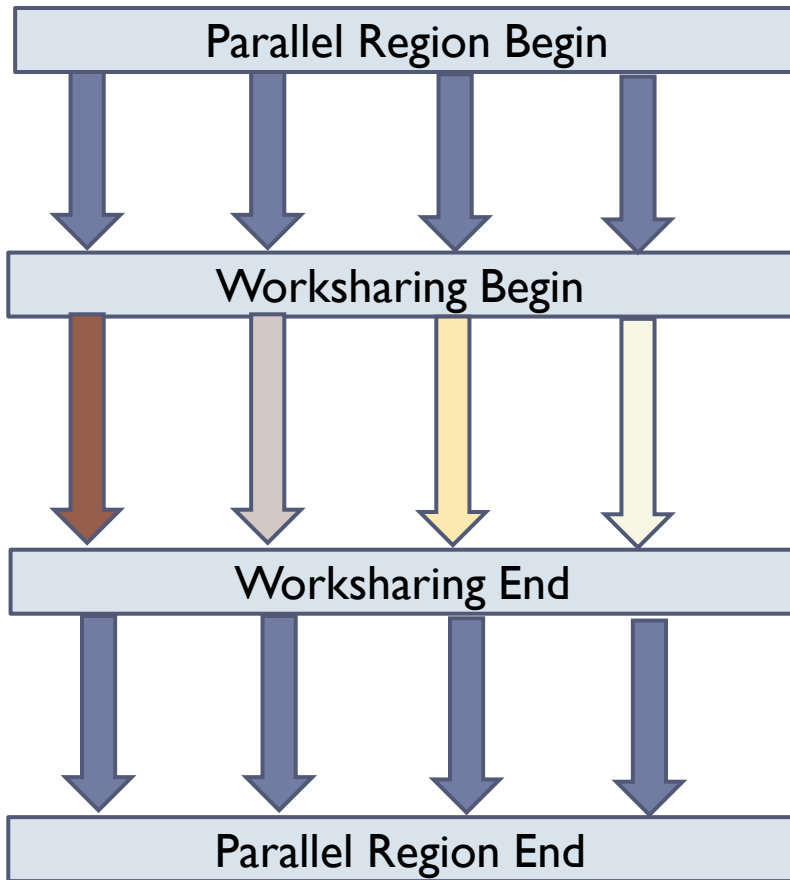
```

void foo()
{
    /* single thread */
    some_work();

    #pragma omp parallel
    {
        /* multiple threads */
        work_in_parallel();
    }

    /* back to single thread */
    finish_work();
}
    
```

Worksharing Constructs



```
#pragma omp parallel
{
  /* all threads execute this */
  work1();

  /* distribute iterations amongst
  threads */
  #pragma omp for
  for (i =0; i < N; i++) {
    work2(&A[i]);
  }

  /* all threads execute this */
  work3();
}
```

Programming Model

- ▶ **Fork-Join Threading Model**
 - ▶ control number of threads through API (directive clauses, runtime routines, environment variables)
 - ▶ support for thread synchronization and mutual exclusion
 - ▶ nested parallel regions for hierarchical parallelism (not currently supported in Open64)
- ▶ **Data Environment**
 - ▶ variable may be shared or private
 - ▶ private variables may be local to a region, or persistent across regions (**threadprivate**)
- ▶ **Relaxed, Weak Ordered Memory Consistency**
 - ▶ reordering allowed between synchronization operations
 - ▶ data synchronization at FLUSH or where its implied

Agenda

- ▶ Brief Overview of OpenMP
- ▶ **OpenMP 3.0 Additions**
- ▶ Supporting OpenMP 3.0
 - ▶ Compiler Front-end and Back-end implementation
 - ▶ Runtime Implementation
- ▶ Experimental Results
- ▶ Status

OpenMP 3.0

- ▶ Tasks added to handle dynamic and unstructured applications
 - ▶ recursion
 - ▶ tree & graph traversals
- ▶ Execution model based on threads was redefined
 - ▶ implicit task: default tasks executed by thread team; each tied to a thread
 - ▶ explicit task: structured block under a **task** directive; executed by some thread in team (may be deferred)
 - ▶ task completion: **taskwait** and **barrier** will ensure previously created tasks have completed
- ▶ Parallelize loop nests using **collapse** clause

Example: Parallelizing a Recursive Algorithm

```
int fib(int n)
{
    int x, y;
    if (n < 2)
        return n;
    else {
        #pragma omp task shared(x)
        x = fib(n-1);

        #pragma omp task shared(y)
        y = fib(n-2);

        #pragma omp taskwait

        return x + y;
    }
}
```

- ▶ Task Scheduling
 - ▶ when a thread may switch execution to a different task
 - ▶ which task it may execute
- ▶ Scheduling or “switching” points during execution of tied tasks:
 - ▶ immediately following generation of explicit task
 - ▶ at taskwait
 - ▶ at implicit and explicit barrier
 - ▶ after last instruction of task region
 - ▶ at taskyield (3.1)

Example: Parallelizing a Recursive Algorithm

```

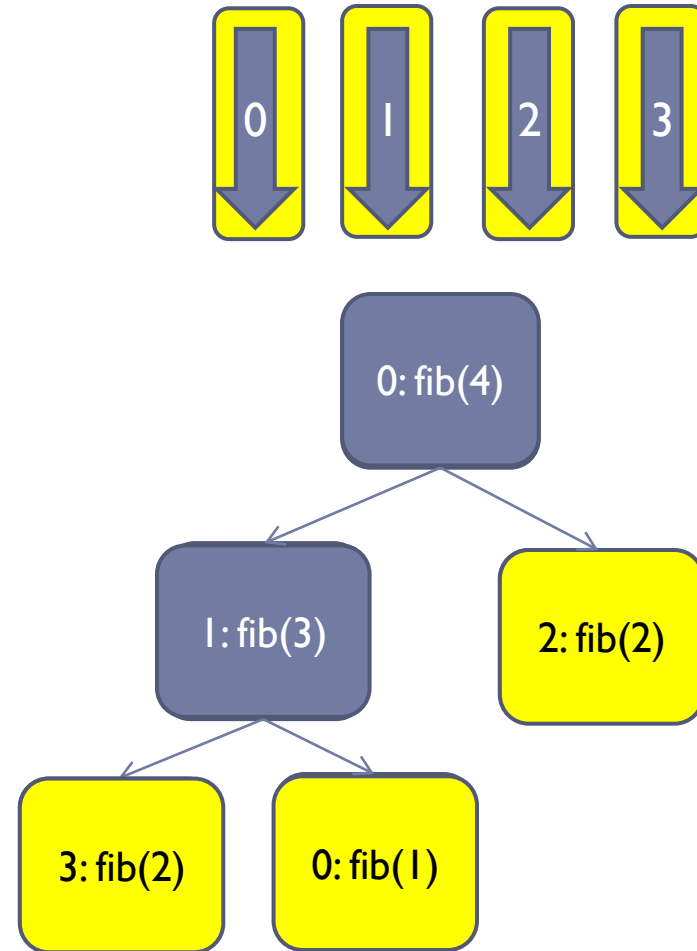
int fib(int n)
{
    int x, y;
    if (n < 2)
        return n;
    else {
        #pragma omp task shared(x)
        x = fib(n-1);

        #pragma omp task shared(y)
        y = fib(n-2);

        #pragma omp taskwait

        return x + y;
    }
}

```



Why not sections?

- ▶ sections are static blocks of work
- ▶ no dependencies between sections
- ▶ code may not appear outside of a section construct
- ▶ recursion accomplished through nested parallelism (more overhead)

```
int fib(int n)
{
    if (n < 2) return n;
    return fib(n-1) + fib(n-2);
}
```

```
int fib(int n)
{
    int x,y;
    if (n < 2) return n;
    #pragma omp parallel sections
    {
        #pragma omp section
        x = fib(n-1);
        #pragma omp section
        y = fib(n-2);
    }
    return x+y;
}
```

Tied vs Untied Tasks

- ▶ **By default, tasks are “tied”**
 - ▶ once it starts executing on a particular thread, it will never suspend and resume execution on a different thread
 - ▶ the executing thread may only switch to a different task at defined scheduling points
 - ▶ tied task may only start executing on a thread once that thread has completed all tasks tied to it which are not in a barrier (unless it descends from all such tasks)
- ▶ **Untied tasks are less restrictive**
 - ▶ helps load balancing for some algorithms
 - ▶ should not be used for codes dependent on thread ID (e.g. use of threadprivate data)

Collapsing Parallel Loops

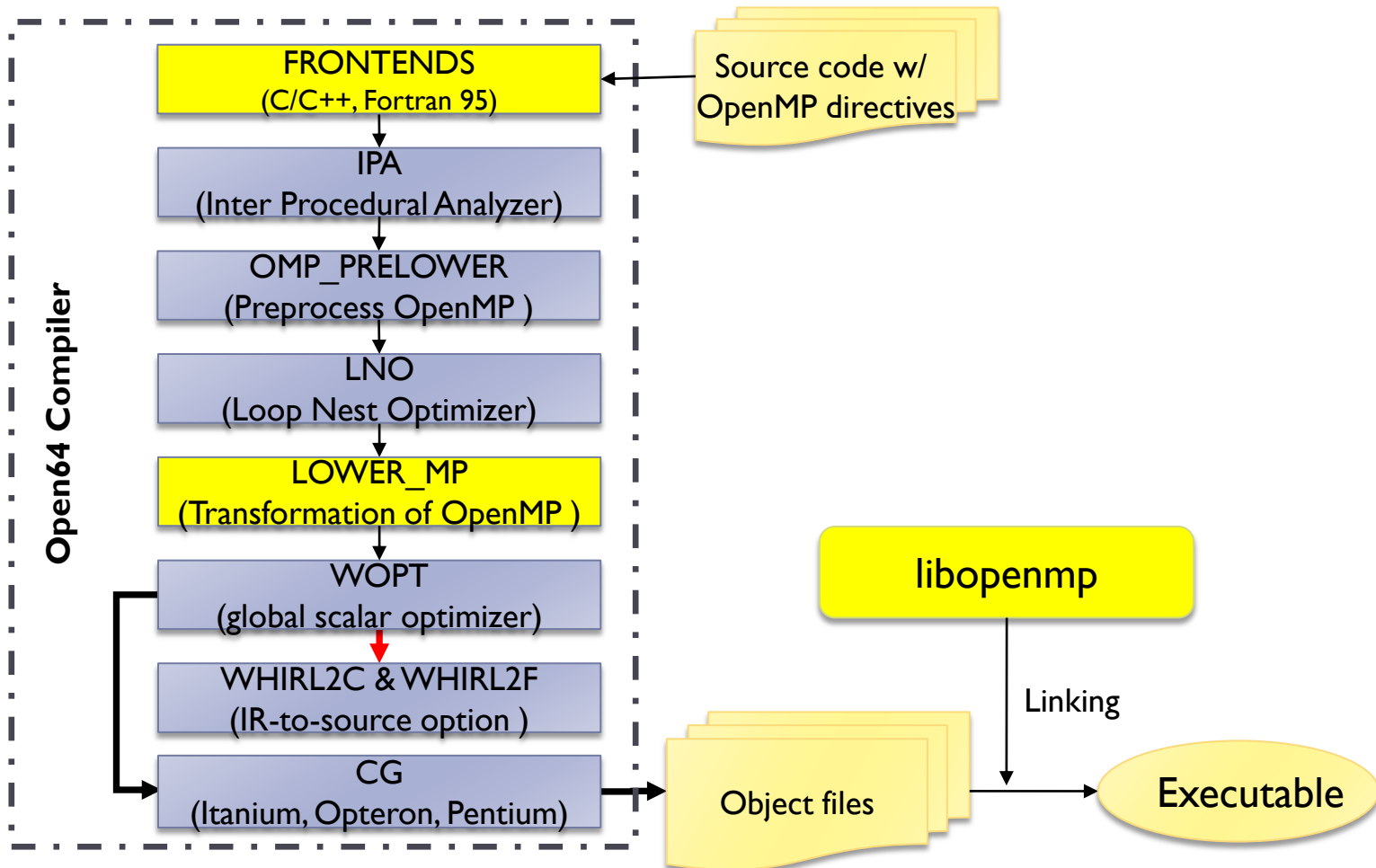
- ▶ The collapse clause is used to parallelize perfectly nested multi-dimensional loops without using nested parallelism
- ▶ Takes a constant positive integer expression as parameter
- ▶ Parameter specifies how many loops should be collapsed by the compiler into one loop before parallelizing the resulting loop

```
#pragma omp for collapse(2)
for( i=0; i<N; i++)
    for( j=0; j<M; j++)
        for( k=0; k<X; k++)
            foo( i, j, k);
```

Agenda

- ▶ Brief Overview of OpenMP
- ▶ OpenMP 3.0 Additions
- ▶ **Supporting OpenMP 3.0**
 - ▶ Compiler Front-end and Back-end implementation
 - ▶ Runtime Implementation
- ▶ Experimental Results
- ▶ Status

Extending Open64 to Support OpenMP 3.0



Front-end: task and taskwait construct

source code

```
#pragma omp task untied shared(x) \  
    if(x < 10)  
  
{  
    x = foo(x);  
}  
  
#pragma omp taskwait
```

- GNU4, wgen, GNU3
- emits REGION for task construct
- REGION PRAGMAS:
 - TASK_BEGIN
 - SHARED, PRIVATE, FIRSTPRIVATE
 - IF
 - UNTIED
- emit pragma for TASKWAIT

VH WHIRL

```
I4STID 0 <2,2,_temp__mp_xpragma0> T<4,.,predef_14,4> {line: 0/0}  
I4I4GT  
I4I4LDID 0 <2,1,x> T<4,.,predef_14,4>  
I4INTCONST 10 (0xa)  
REGION 1 (kind=4) {line: 1/5}  
...  
REGION PRAGMAS  
BLOCK {line: 1/7}  
PRAGMA 2 184 <null-st> 0 (0x0) # TASK_BEGIN {line: 1/5}  
XPRAGMA 2 40 <null-st> # IF {line: 1/5}  
I4I4LDID 0 <2,2,_temp__mp_xpragma0> T<4,.,predef_14,4>  
PRAGMA 2 54 <2,1,x> 0 (0x0) # SHARED {line: 1/5}  
PRAGMA 2 187 <null-st> 0 (0x0) # UNTIED {line: 1/5}  
END_BLOCK  
REGION BODY  
BLOCK {line: 1/5}  
...  
END_BLOCK  
END_REGION 1  
PRAGMA 2 186 <null-st> 0 (0x0) # TASKWAIT {line: 1/10}
```

Task Implementation – Back-end

Translation

- ▶ MP Lowering (wn_mp.cxx)
- ▶ similar translation strategy as for parallel region
- ▶ outline task body into nested PU
- ▶ replaces task REGION with call to **__ompc_task_create**
 - ▶ arg 0: pointer to outlined function
 - ▶ arg 1: frame pointer for shared variables that are on stack
 - ▶ arg 2: may delay?
 - ▶ arg 3: is tied?
- ▶ replaces TASKWAIT pragma with call to **__ompc_task_wait**

WHIRL2C (-CLIST:emit_nested_pu)

```
__ompc_task_create(&__omprg_main_l,  
_w2c_reg3, _temp__mp_xpragma0, 0);  
  
__ompc_task_wait();  
  
/* ... */  
  
static void __omprg_main_l(__ompv_slink_a)  
_UINT64 __ompv_slink_a;  
{  
    register _INT32 _w2c__comma;  
    _UINT64 _temp__slink_sym1;  
    _temp__slink_sym1 = __ompv_slink_a;  
  
    __ompc_task_body_start();  
    _w2c__comma = foo(x);  
    x = _w2c__comma;  
    __ompc_task_exit();  
  
    return;  
} /* __omprg_main_l */
```

Closer Look at Outlined Task

source code

```
#pragma omp task shared(x) private(y)
{
    /* z is firstprivate by default */
    initl(&y);
    x = f(y, z);
}
```

z is **firstprivate**; initialize local z with the original z (through up-level ref)

- body of task is executed by some thread in team, prior to execution of next taskwait/barrier in parent task
- accesses to x occur via up-level ref

WHIRL2C (before CG)

```
static void __omprg_main_l(__ompv_slink_a)
    _UINT64 __ompv_slink_a;
{
    /* declarations */
    register _INT32 _w2c__comma;
    _UINT64 _temp__slink_sym0;
    _INT32 __mplocal_y;
    _INT32 __mplocal_z;
    _temp__slink_sym0 = __ompv_slink_a;
    __mplocal_z = *((_INT32 *) _temp__slink_sym0 + -4LL);
    __ompc_task_body_start();
    initl(&__mplocal_y);
    _w2c__comma = f(__mplocal_y, __mplocal_z);
    *((_INT32 *) _temp__slink_sym0 + -8LL) = _w2c__comma;
    __ompc_task_exit();
    return;
} /* __omprg_main_l */
```

__ompc_task_body_start(); switches back to parent task if `may_delay=1`

__ompc_task_exit(); signals task completion

Agenda

- ▶ Brief Overview of OpenMP
- ▶ OpenMP 3.0 Additions
- ▶ **Supporting OpenMP 3.0**
 - ▶ Compiler Front-end and Back-end implementation
 - ▶ Runtime Implementation
- ▶ Experimental Results
- ▶ Status

OpenMP Runtime Library

- ▶ Provide interfaces to the programmer
 - ▶ Runtime library functions
 - ▶ Environment variables
 - ▶ Internal Control Variables
- ▶ Support compiler translation of OpenMP directives
 - ▶ Creating, managing, and destroying threads
 - ▶ Distributing work among threads
 - ▶ Thread synchronization

Implicit Tasks

- ▶ Compiler creates nested functions from parallel regions
- ▶ **__ompc_fork**: starts up a new thread team
 - ▶ Determine how many slave threads to create
 - ▶ Based on number of cores, **num_threads** clause, and *nthreads_var* internal control variable
 - ▶ uses **pthread_create** to fork slave threads or **pthread_cond_broadcast** to wake them up *
- ▶ Implicit task is initialized with pointer to parallel region's microtask
 - ▶ each thread in team executes an implicit task

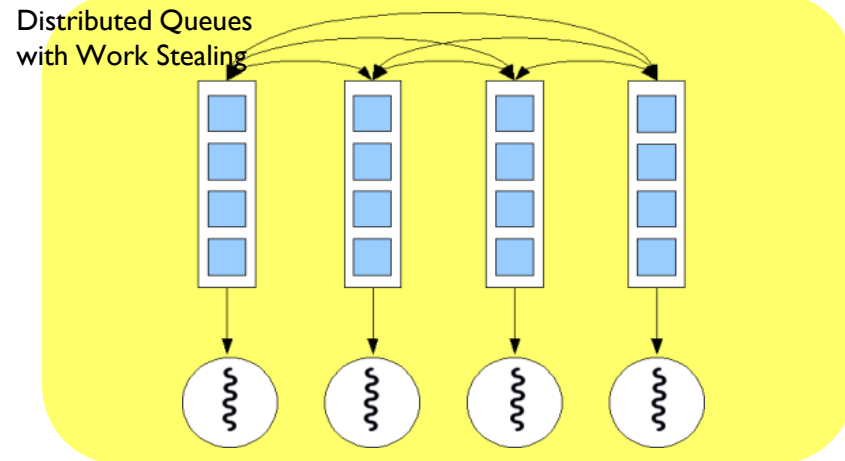
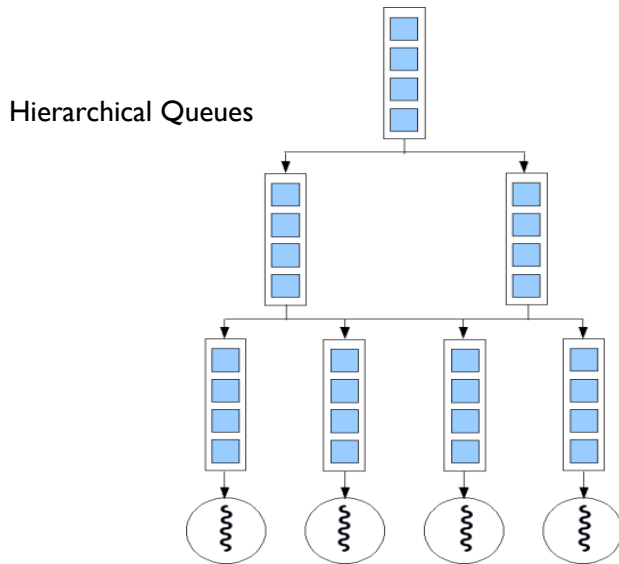
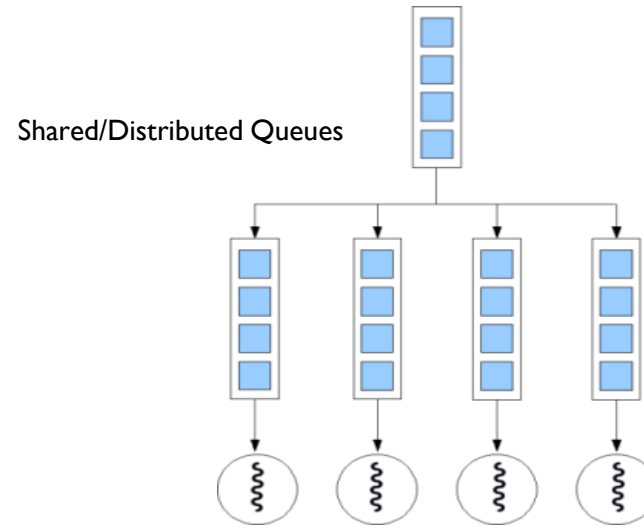
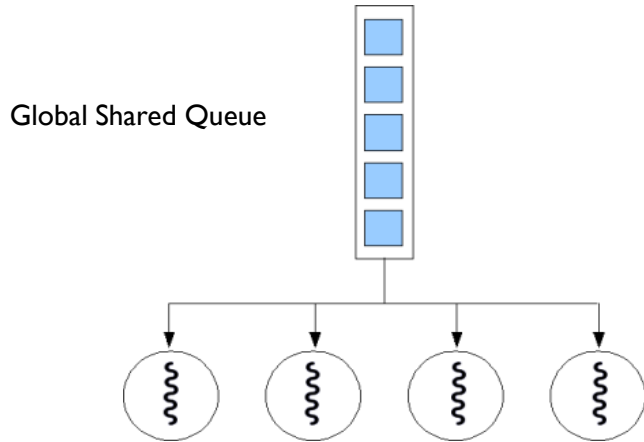
* Note: Threads are maintained across parallel regions

- ▶ Created initially during program startup
- ▶ New threads created/destroyed for nested regions only
- ▶ Creating and destroying threads are expensive

Supporting Explicit Tasks

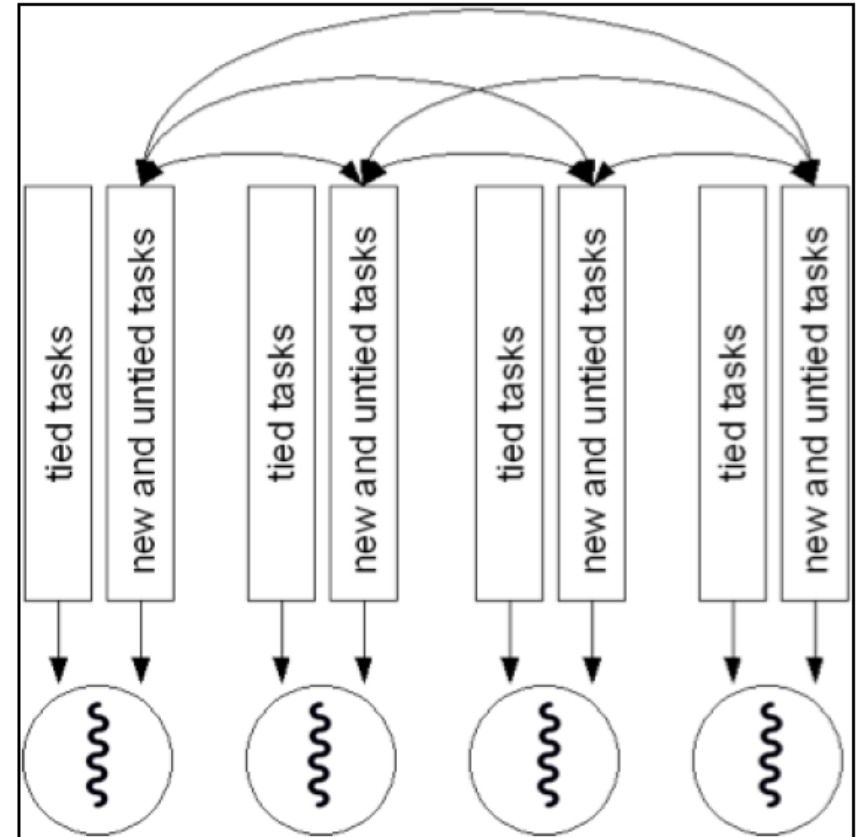
- ▶ The runtime is entirely in charge of scheduling explicit tasks
 - ▶ Explicit tasks are asynchronous
 - ▶ Guaranteed completion only at synchronization points
- ▶ Key Questions:
 - ▶ Storage:
 - ▶ Where/how to store unfinished tasks?
 - ▶ Scheduling
 - ▶ Which task to execute?
 - ▶ Where to execute task?
 - ▶ Load balancing
 - ▶ Are all threads equally busy?
 - ▶ Switching
 - ▶ Tasks do not have to execute to completion!

Task Queue Organization



Task Queues in our Implementation

- ▶ 2 dequeues per thread
 - ▶ Hold unfinished tasks
 - ▶ Private queue
 - ▶ for tied tasks.
 - ▶ can only be accessed by owner thread
 - ▶ Public queue
 - ▶ for work-stealing
 - ▶ can be accessed by all threads in team



Minimize contention to task pool

Tasking API

▶ External API used by compiler

- ▶ `__ompc_task_create()` creates a task and inserts it into a queue
- ▶ `__ompc_task_wait()` suspends a task until all of its children complete
- ▶ `__ompc_task_body_start()` tells the runtime to enqueue the task for scheduling
- ▶ `__ompc_task_exit()` called at the end of a task to perform cleanup and schedule a new task
- ▶ `__ompc_task_create_cond()` returns true if a new task should be created

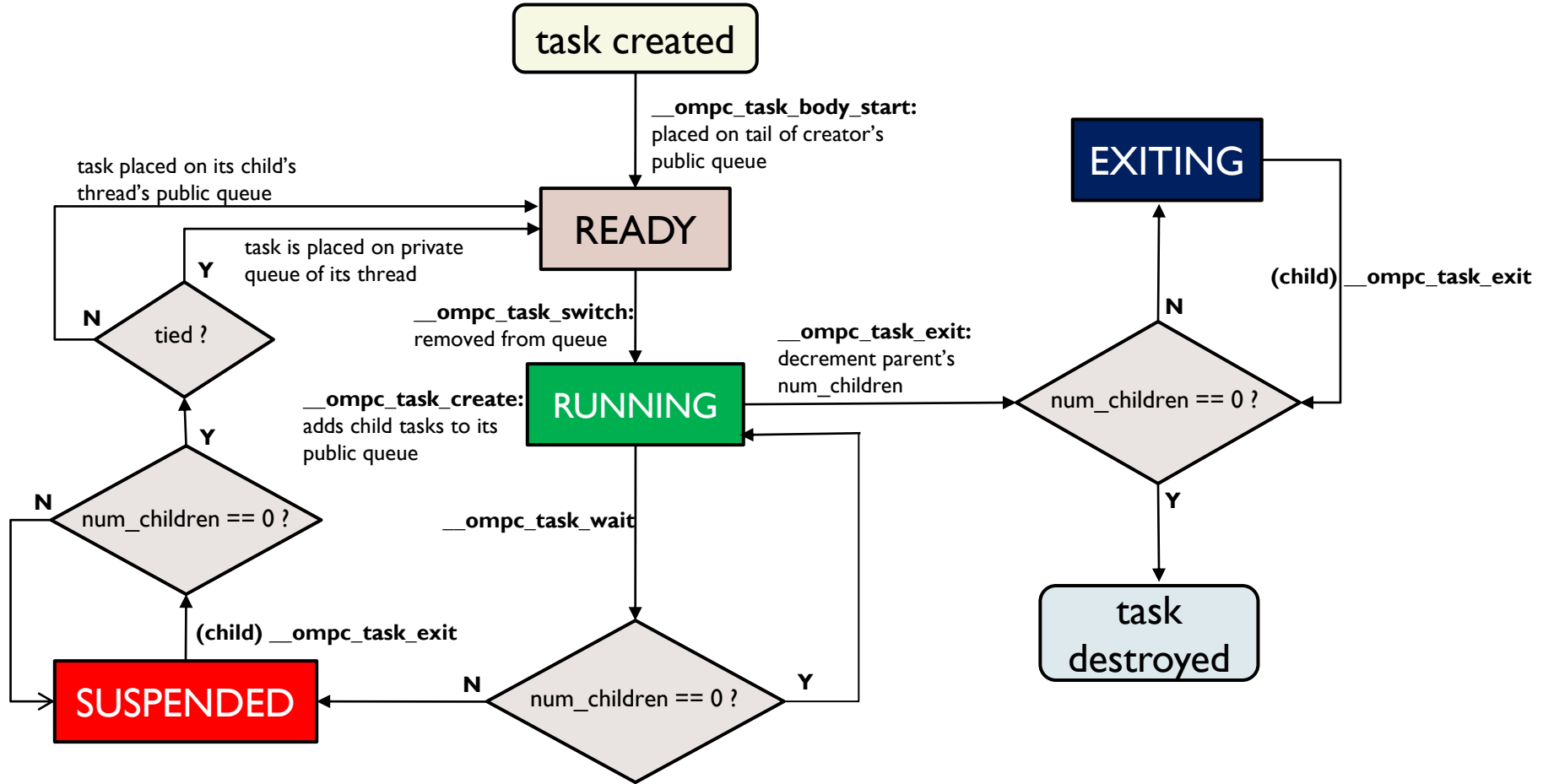
▶ Internal API

- ▶ `omp_task_t` data structure representing a task
- ▶ `__ompc_init_vp()` creates implicit task for thread
- ▶ `__ompc_get_task()` initializes a new coroutine for task
- ▶ `__ompc_task_schedule()` returns a ready task from scheduler
- ▶ `__ompc_delete_task()` destroys a task
- ▶ `__ompc_task_switch()` saves the context of the current task and switches to a new task

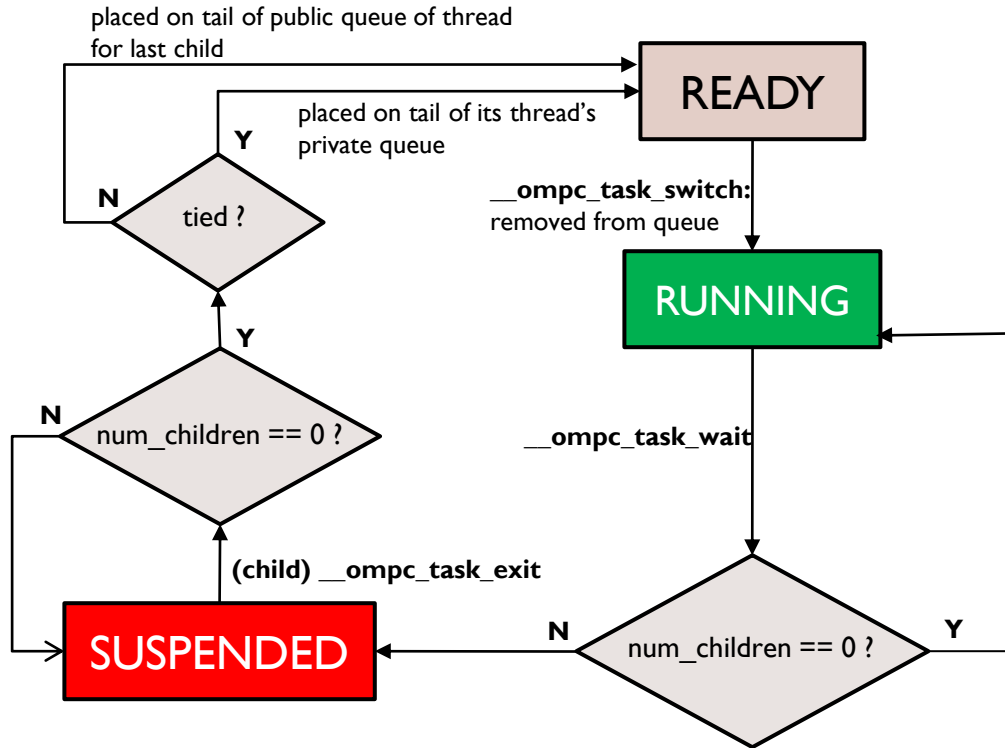
▶ Environment Variables

- | | | | |
|---------------------------------------|-------|-------------------------------------|----|
| ▶ <code>OMP_TASK_STACK_SIZE</code> | 64 KB | ▶ <code>OMP_TASK_LIMIT</code> | 2p |
| ▶ <code>OMP_TASK_CREATE_COND</code> | TRUE | ▶ <code>OMP_TASK_LEVEL_LIMIT</code> | 3 |
| ▶ <code>OMP_TASK_Q_UPPER_LIMIT</code> | 10 | ▶ <code>OMP_TASK_MOD_LIMIT</code> | 3 |
| ▶ <code>OMP_TASK_Q_LOWER_LIMIT</code> | 1 | | |

Task Creation and Execution



While task is suspended ...



- ▶ suspends task while $\text{num_children} > 0$
- ▶ thread may switch to another task while waiting
 - ▶ first checks private queue
 - ▶ then public queue
 - ▶ then public queue of other threads

Change to barrier implementation

- ▶ all tasks created by threads in current team must complete at barrier exit
- ▶ a global counter for entire thread team:
$$\text{num_tasks} = \# \text{ threads} + \# \text{ explicit tasks}$$
- ▶ when thread executes barrier:
 - ▶ decrement num_tasks
 - ▶ if $\text{num_tasks} > 0$, find a new task and execute
 - ▶ if $\text{num_tasks} == 0$, that means all implicit tasks (i.e. threads) have reached barrier and all created tasks are complete
 - ▶ reset $\text{num_tasks} = \# \text{ threads}$, and resume threads

Task Switching

▶ Required functionality

- ▶ creating/deleting *task* objects
- ▶ migrate tasks across threads
- ▶ suspend and resume tasks

▶ Portable Coroutine Library (PCL)

- ▶ coroutine: a subroutine with multiple entry points that can be suspended and resumed
- ▶ context switching via *getcontext*, *makecontext*, *setcontext*, *swapcontext*
- ▶ portable across POSIX-compliant O/s

Summarizing runtime support for tasks

- ▶ Minimal contention for task pool → distributed queue organization
- ▶ Lots of work for threads → breadth-first creation
- ▶ Locality, data reuse → depth-first execution
- ▶ No idle threads → work-stealing
- ▶ Support task switching → coroutines

Agenda

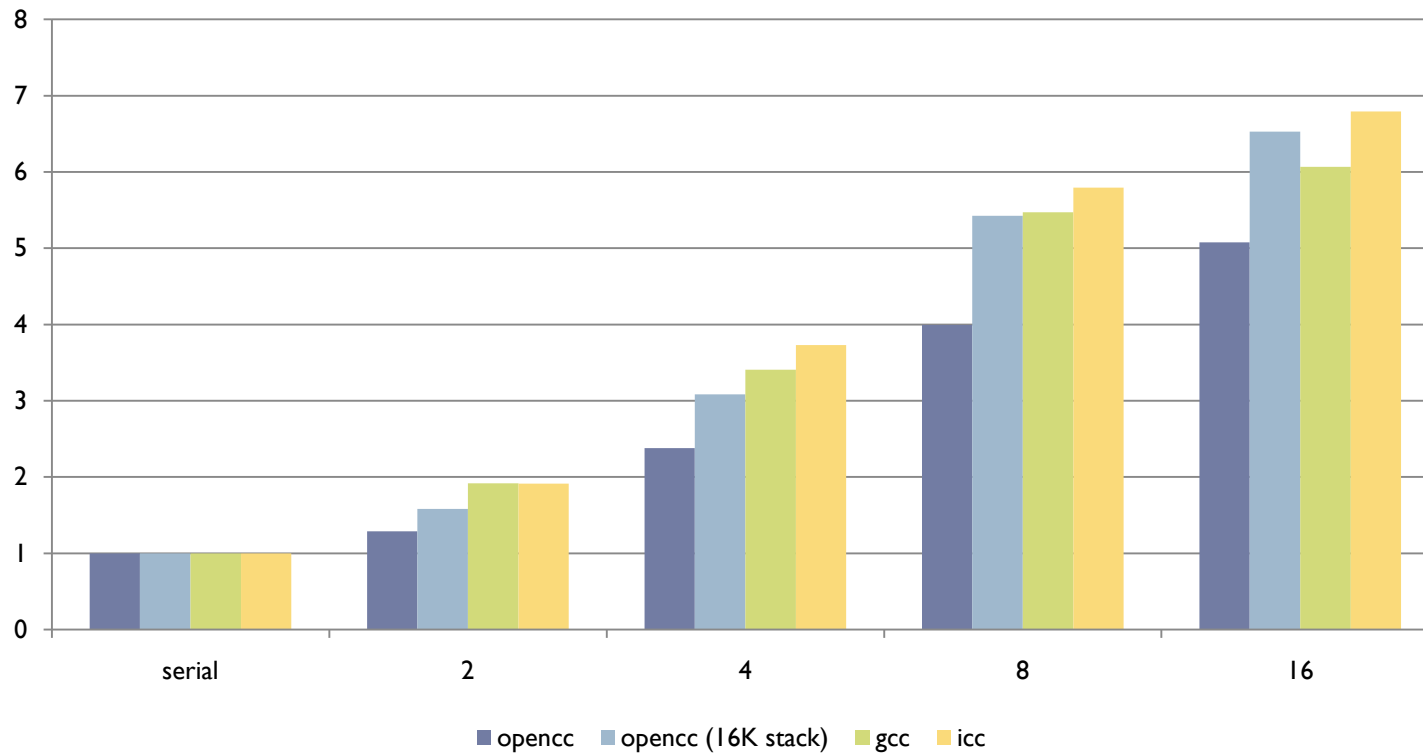
- ▶ Brief Overview of OpenMP
- ▶ OpenMP 3.0 Additions
- ▶ Supporting OpenMP 3.0
 - ▶ Compiler Front-end and Back-end implementation
 - ▶ Runtime Implementation
- ▶ **Experimental Results**
- ▶ Status

Barcelona OpenMP Tasks Suite

- ▶ Task Suite from Barcelona Supercomputing Center:
 - ▶ **sort**: mixture of sorting algorithms to sort a vector
 - ▶ **sparselu**: LU factorization of sparse matrix
 - ▶ **strassen**: matrix multiply using Strassen method
 - ▶ **alignment**: aligns sequence of proteins
 - ▶ **fft**: compute fast fourier transform
 - ▶ **floorplan**: optimal placement of cells in floorplan
 - ▶ **health**: simulates country health system
 - ▶ **nqueens**: find solution to N queens problem
- ▶ System: dual 2.27 GHz 8-core Intel Xeon Nehalem, 32 GB RAM

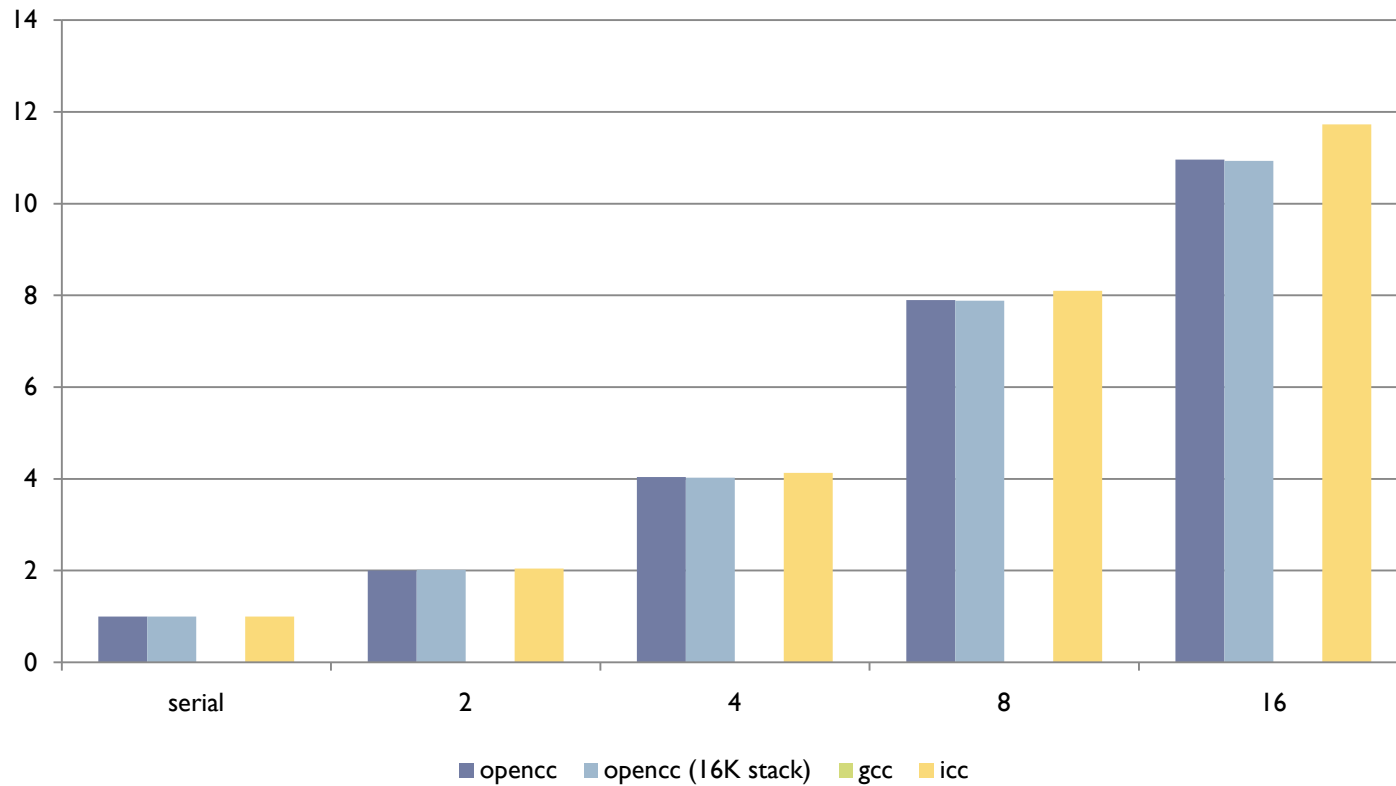
BOTS: sort

BOTS: sort (large)



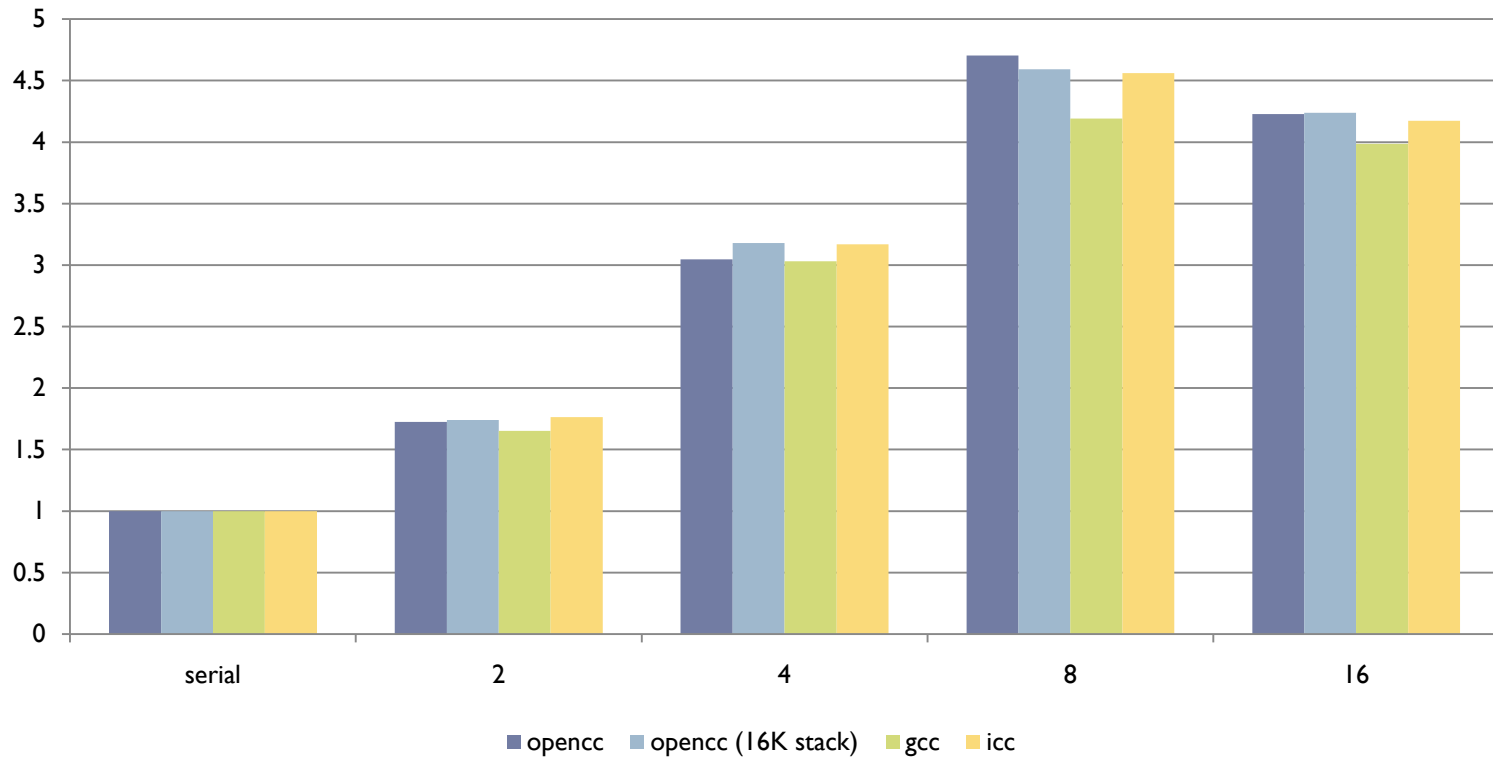
BOTS: sparse LU

BOTS: Sparse LU



BOTS: strassen

BOTS: Strassen



Agenda

- ▶ Brief Overview of OpenMP
- ▶ OpenMP 3.0 Additions
- ▶ Supporting OpenMP 3.0
 - ▶ Compiler Front-end and Back-end implementation
 - ▶ Runtime Implementation
- ▶ Experimental Results
- ▶ **Status**

Implementation Status

- ▶ Currently maintained by Tsinghua University
 - svn co <https://svn.open64.net/svnroot/open64/branches/openmp3.0>
- ▶ Tsinghua University has provided:
 - ▶ GNU4 front-end
 - ▶ backend support
- ▶ Incorporates earlier UH implementation
 - ▶ GNU3 front-end
 - ▶ OpenMP runtime (libopenmp)
- ▶ Supported Features: **task, taskwait, collapse**
- ▶ TODO:
 - ▶ Fortran front-end
 - ▶ Bug Fixing
 - ▶ Performance Tuning

Acknowledgements

▶ Jiangzhou He

- ▶ Institute of High-Performance Computing, Tsinghua U.
- ▶ Created ***openmp3.0*** branch in Open64 SVN repo
- ▶ Provided GNU4 and back-end support for OpenMP 3.0
- ▶ Added support for collapse clause

▶ Cody Addison

- ▶ formerly HPCTools Group, UH
- ▶ Initial implementation for OpenMP 3.0 tasks in OpenUH compiler/runtime
- ▶ Experimental Results

More information

- ▶ OpenMP.org

<http://openmp.org/wp/>

- ▶ HPCTools OpenMP Page

<http://www2.cs.uh.edu/~hpctools/OpenMP/>

Thank You.
Questions?

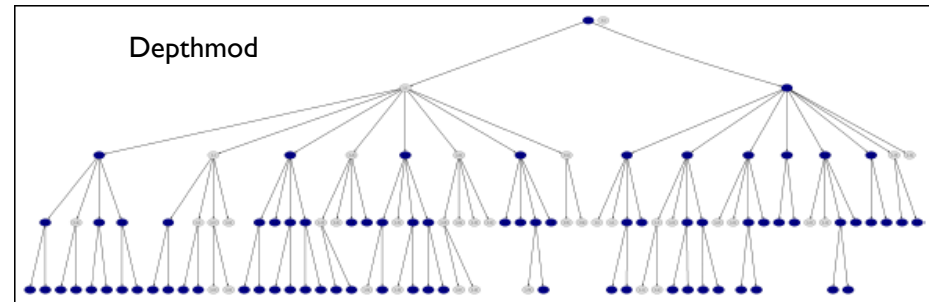
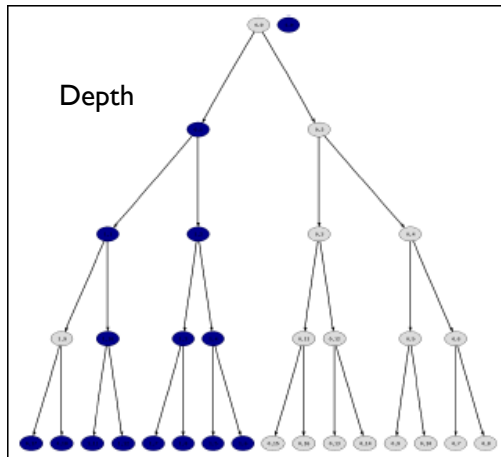
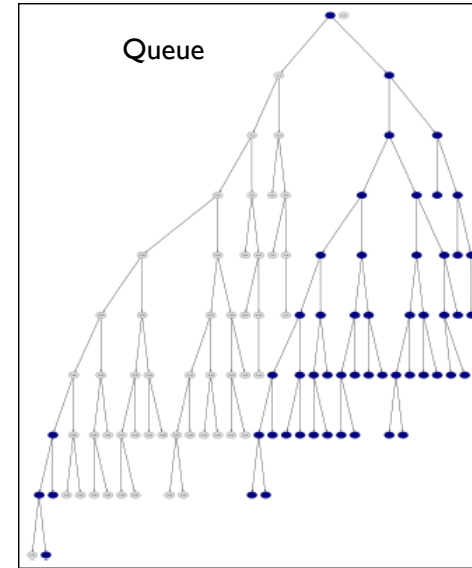
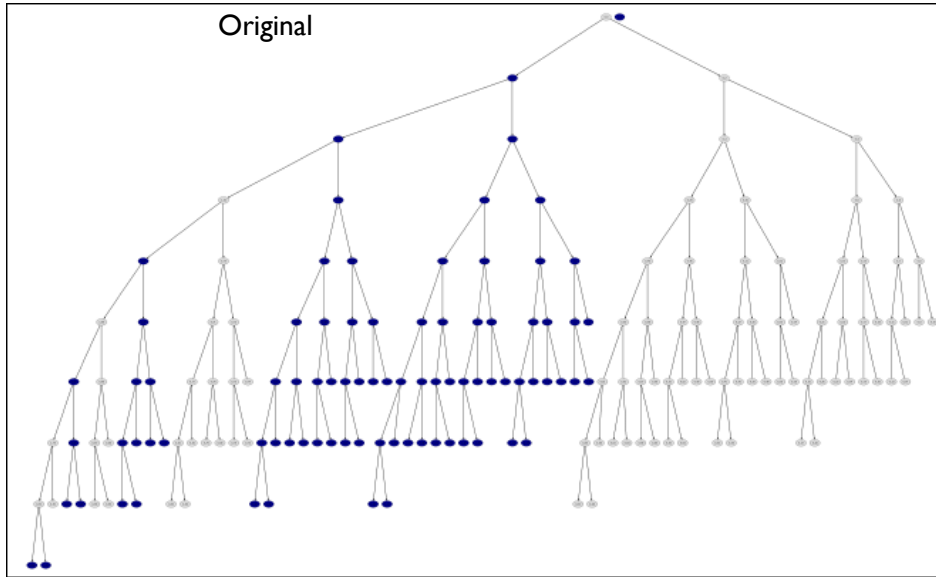
Appendix

- ▶ Tasking Limiting Strategies in OpenUH
- ▶ Results of Task Limiting Study

Task Limiting Strategies

- ▶ **numtasks**
 - ▶ If the number of tasks greater than N
- ▶ **depth**
 - ▶ If the depth of the task in the task graph is greater than some limit
- ▶ **depthmod**
 - ▶ If the depth modulus N is equal to zero
- ▶ **queue**
 - ▶ Uses the number of tasks in the thread's public queue
 - ▶ Idea: Fill the queue up to an upper limit, then drain it to a lower limit
 - ▶ Provides work for other tasks to steal

Task Limiting Strategies, Fibonacci Example



Experiments

▶ Machines

- ▶ SGI Altix 350
 - ▶ Processors: 16 Itanium-2 processors
 - ▶ Memory: 16 GB per node = 128 GB total
 - ▶ GCC 4.1.2
- ▶ PSSC Labs Octo
 - ▶ Processors: 8 dual core Opterons (16 cores total)
 - ▶ Memory: 64 GB
 - ▶ GCC 4.2.3

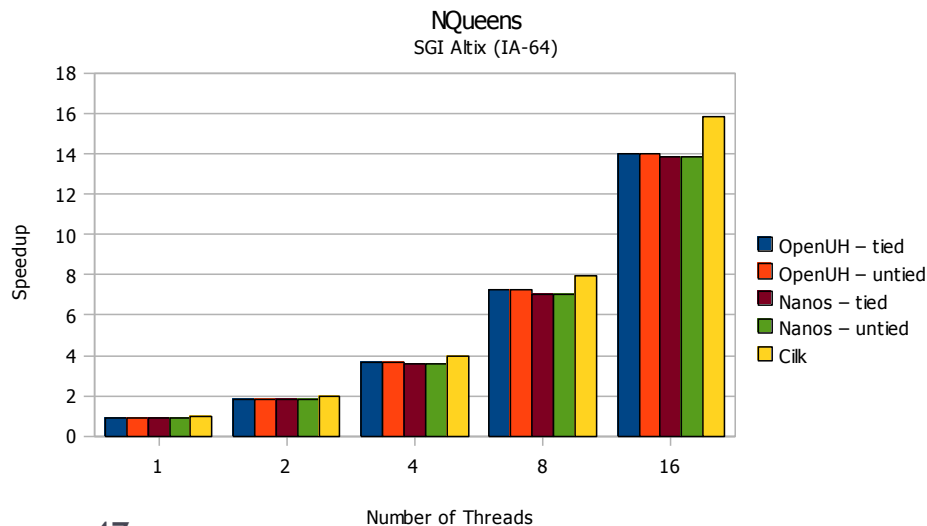
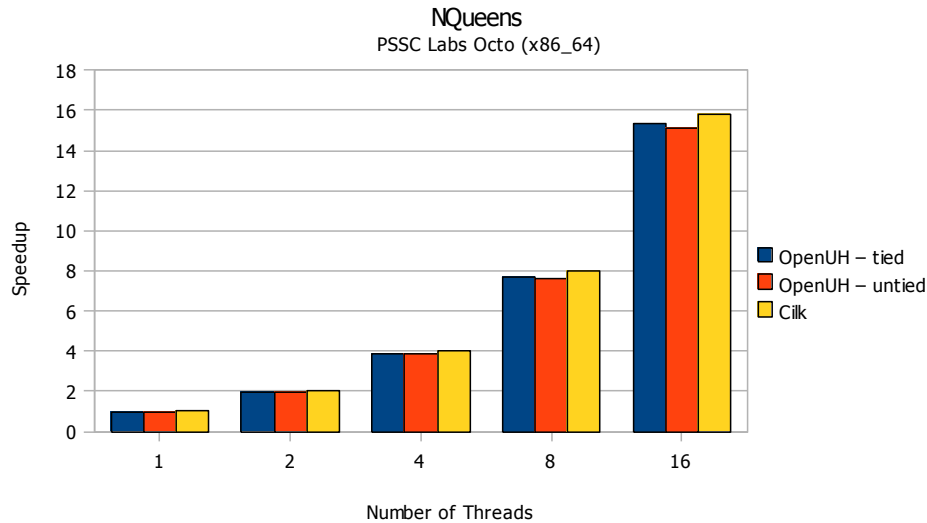
▶ Runtime Library Comparison

- ▶ Compare Cilk 5.4.6, Nanos 4.2.0 and OpenUH
- ▶ Libraries compiled with GCC -O2
- ▶ Cilk used GCC as backend
- ▶ Nanos used OpenUH
- ▶ Speedups taken based on serial execution with no OpenMP translation
- ▶ Applications compiled using -O3

▶ Task Create Condition Comparison

Runtime Library Comparison

NQueens



Algorithm

- Recursive
- Lots of tasks created
- After a few levels, little branching occurs in the task graph
- No communication between tasks
- Data set is small
- Good measure of library overheads

Results

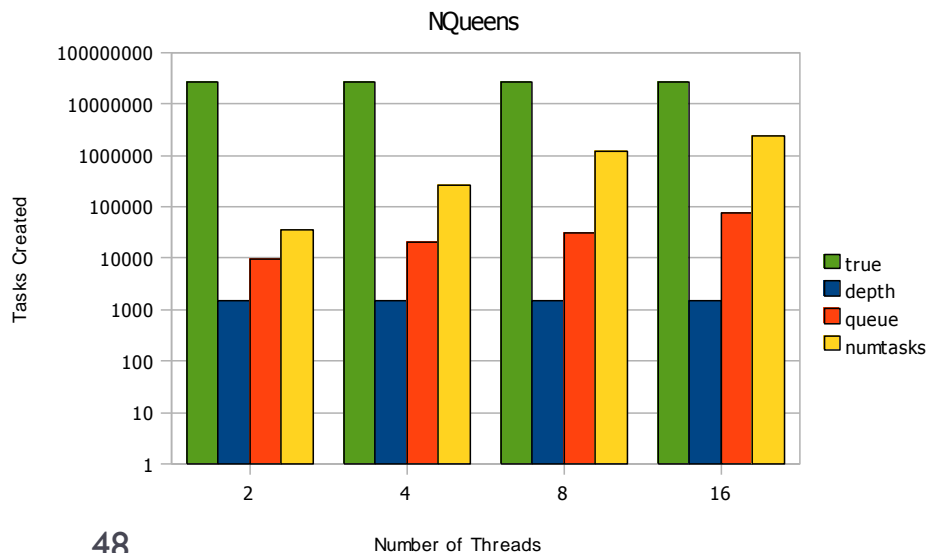
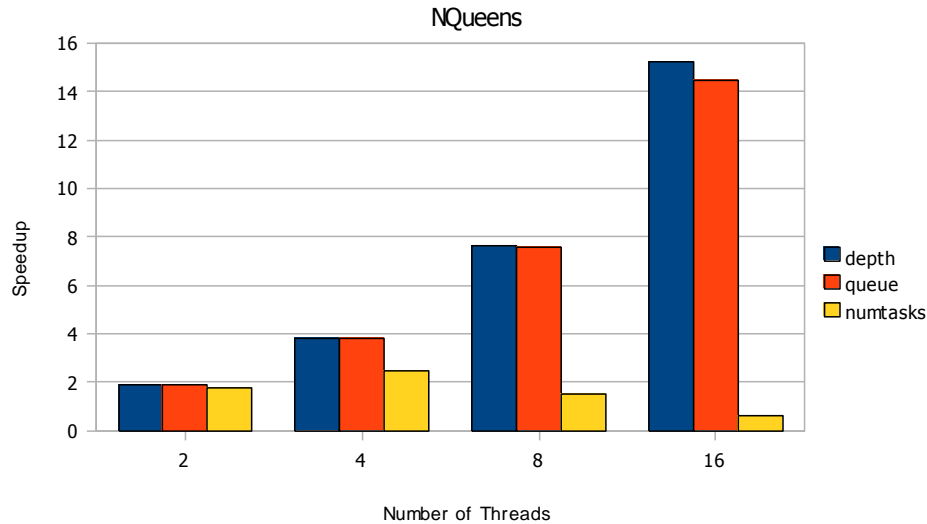
- Cilk achieves best performance on both architectures
- tied tasks slightly outperform untied

Conclusions

- Cilk's better performance is a result of faster task creation and lack of parallel and barrier regions
- The better performance of tied tasks is as a result of reduced queue contention

Task Create Condition

NQueens



- **Conditions**

- depthmod and true were omitted due to lengthy execution times

- **Results**

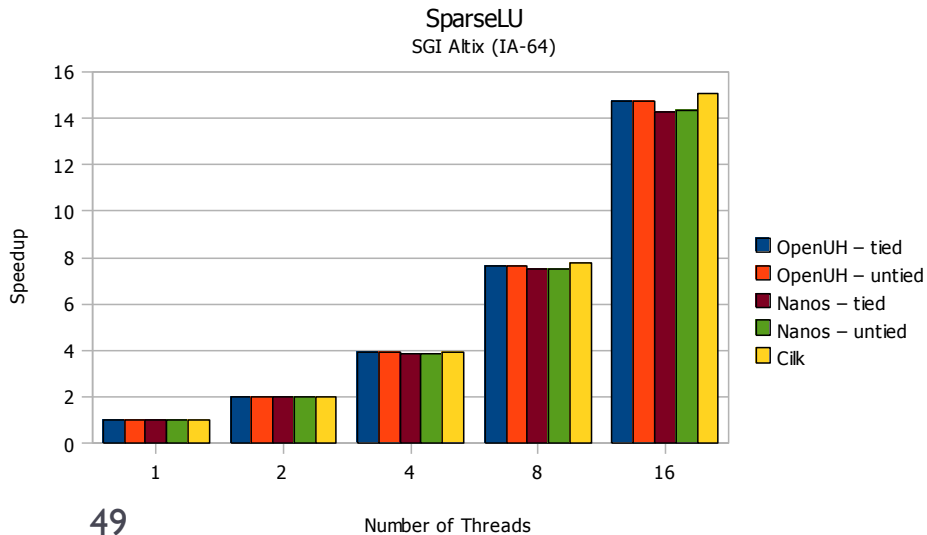
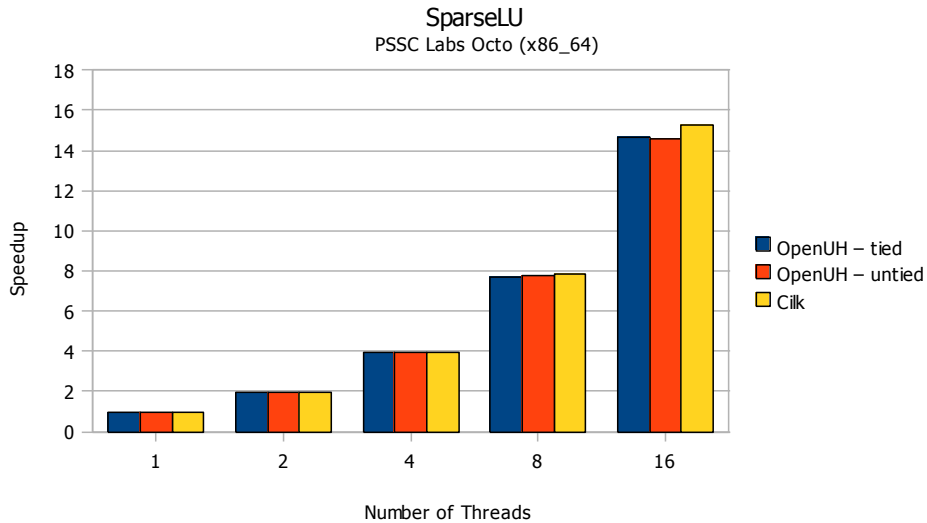
- depth performs better than others

- **Conclusions**

- The performance corresponds directly with the number of tasks created

Runtime Library Comparison

SparseLU



Algorithm

- ▶ One task creates all other tasks
- ▶ One level of depth in the task graph
- ▶ Data is arranged in blocks of 100x100
- ▶ Little communication between tasks

Results

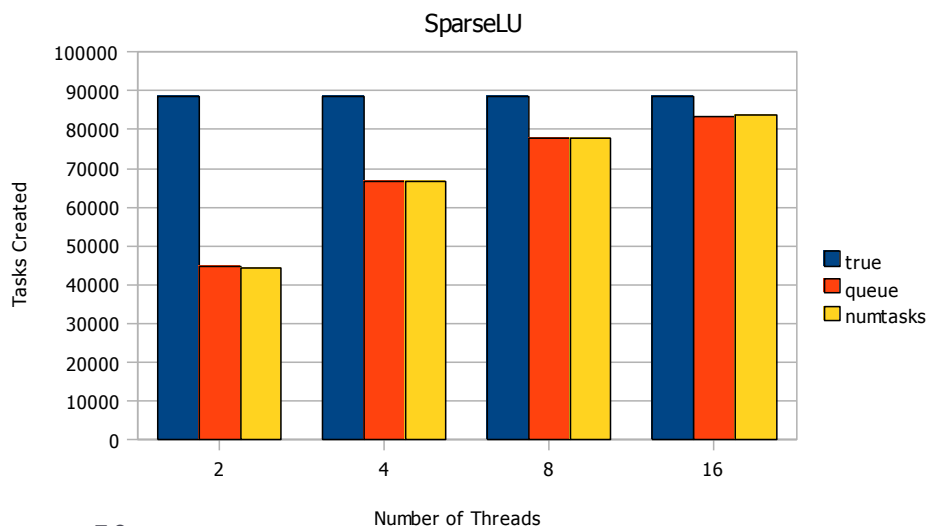
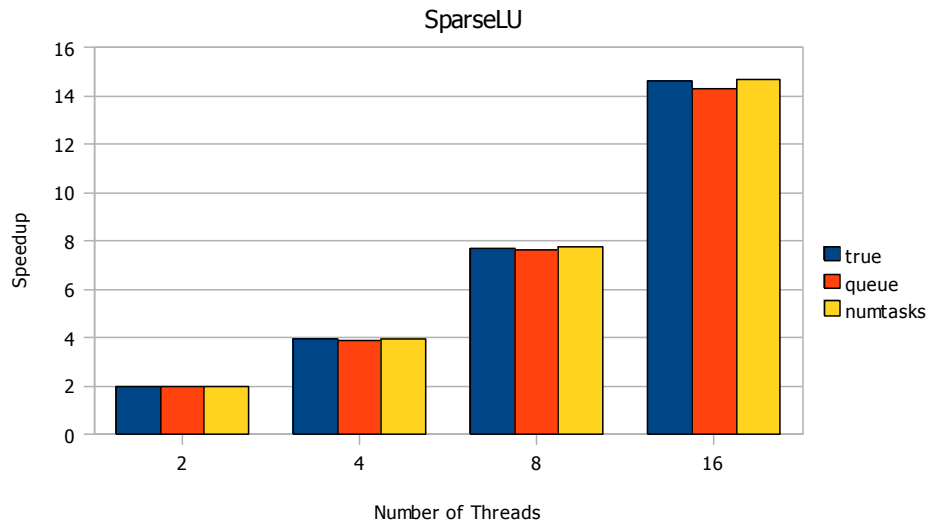
- ▶ Cilk achieves best performance on both architectures
- ▶ tied & untied tasks perform the same

Conclusions

- ▶ Benchmark measures implementation overheads
- ▶ ~85,000 tasks created
- ▶ Cilk's better performance is a result of faster task creation and lack of parallel and barrier regions

Task Create Condition

SparseLU



• Conditions

- depth and depthmod were not tested, since only a single level exists in the task graph

• Results

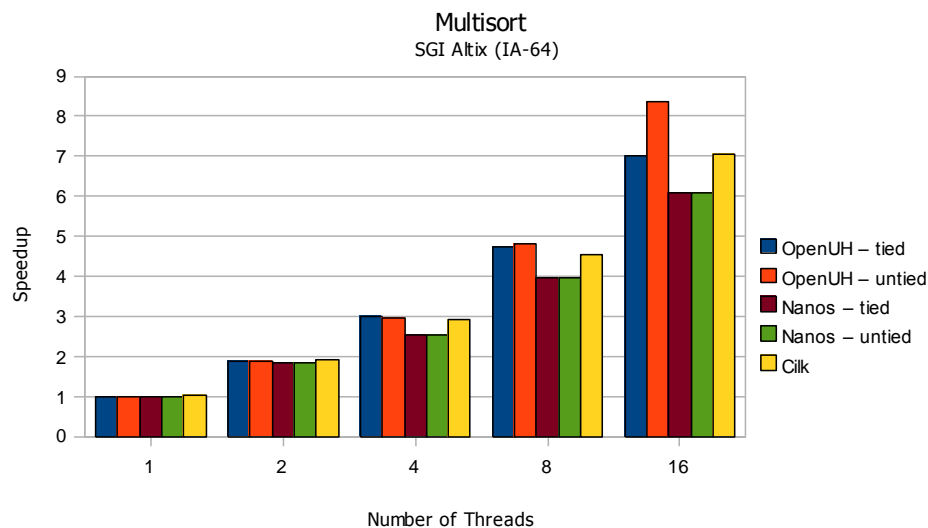
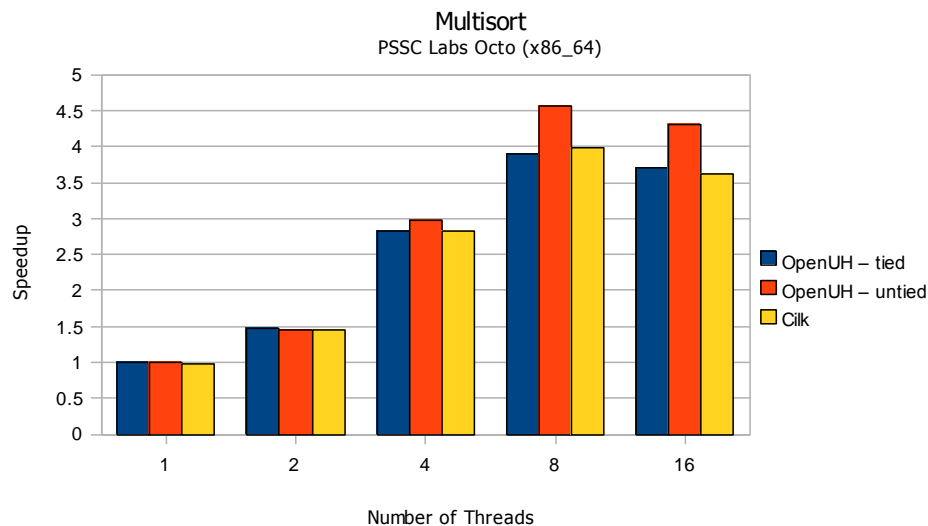
- numtasks performs the same as true

• Conclusions

- It appears that most tasks are needed to achieve parallelism
- the number of tasks created by the queue method is an indication of the amount of tasks needed in a system
- queue method requires three comparisons and one write for every invocation

Runtime Library Comparison

Multisort



Algorithm

- Recursive merge sort
- Irregular task execution times
- Data intensive

Results

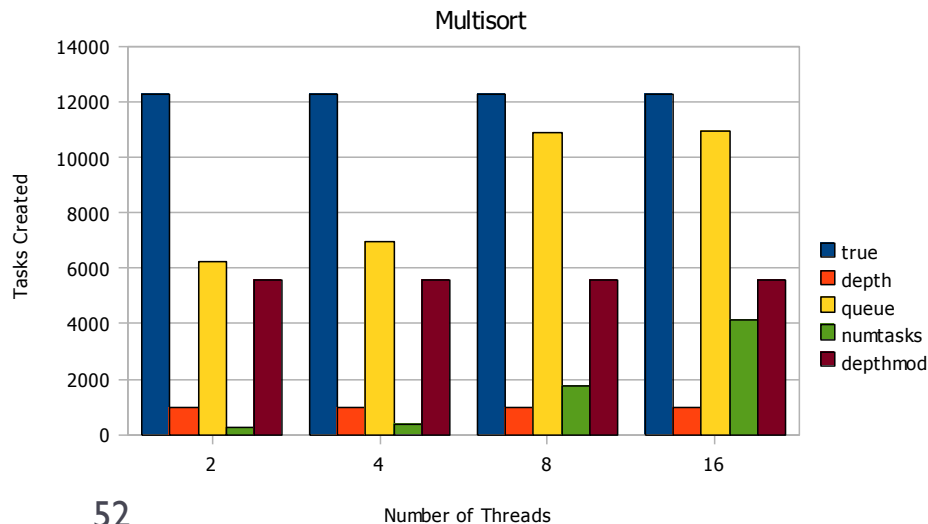
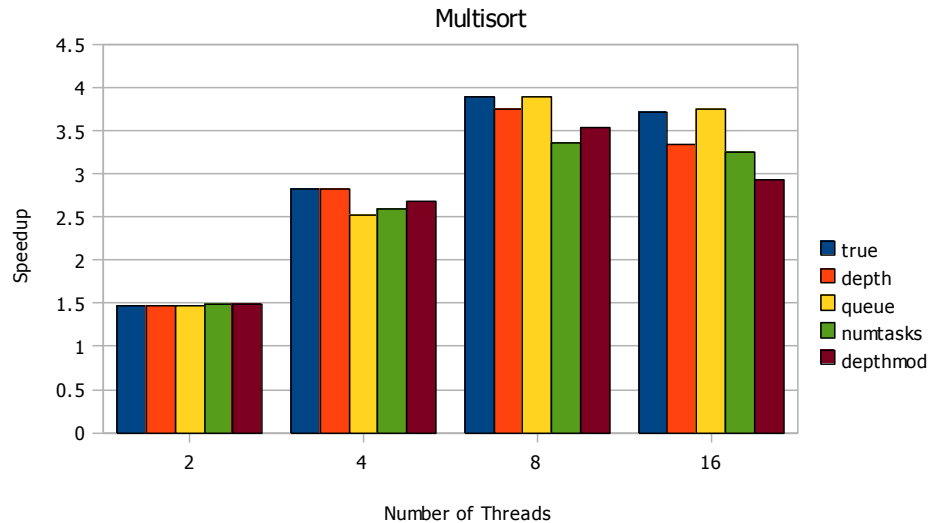
- OpenUH untied tasks perform best on both architectures
- Performance degradation from 8 to 16 threads on Octo server
 - Not seen on Altix

Conclusions

- untied tasks are needed to deal with load imbalance
- With more tasks being created in OpenUH, we provide more opportunity for load balancing
- Performance degradation points to inefficiencies in depth-first schedulers
 - resource contention

Task Create Condition

Multisort



Results

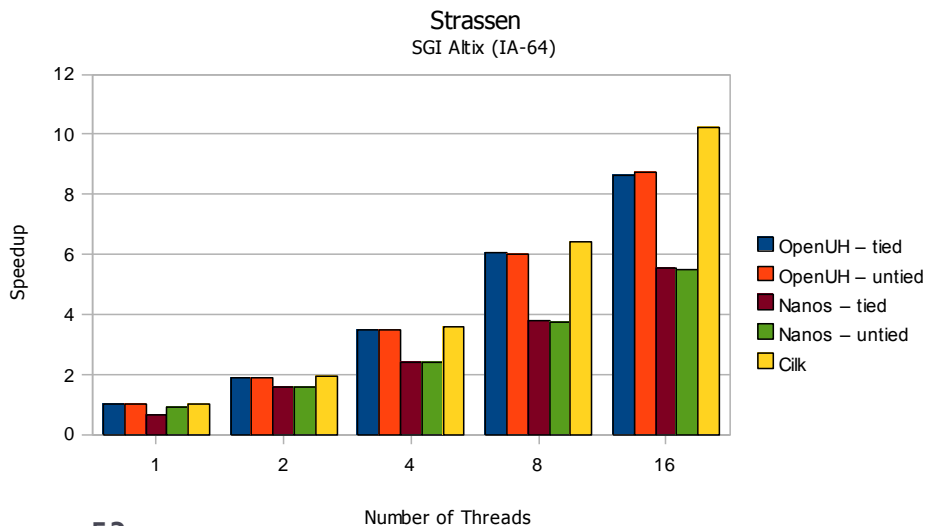
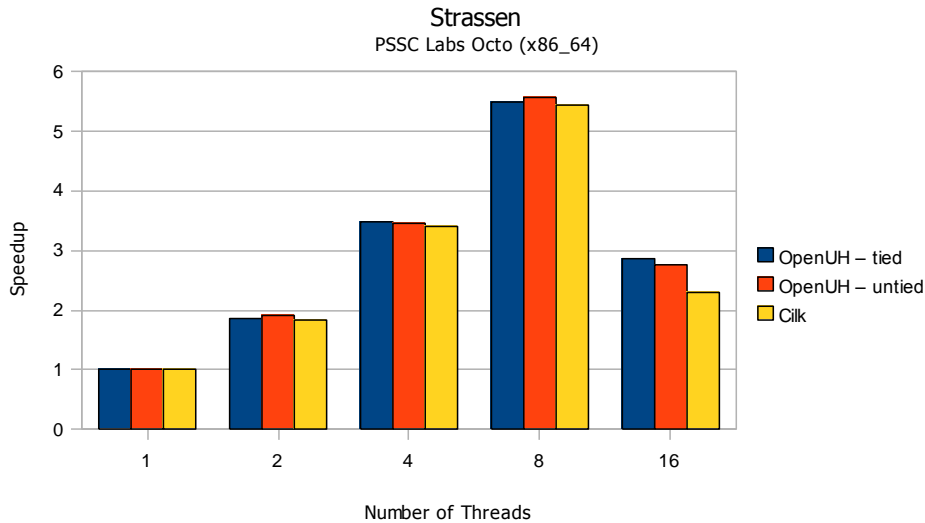
- queue and true perform best for 8 and 16 threads
- queue is worst for 4 threads

Conclusions

- For 8 and sixteen threads, we need lots of tasks to increase load balance
- depth does reasonably well for the number of tasks created
 - good for reducing memory consumption
- Investigate depthmod more closely
 - creates more threads than depth, yet results in less speedup
 - cache behavior or overhead of modulus operation?

Runtime Library Comparison

Strassen



Algorithm

- Recursive matrix multiplication
- Regular in the sense that each task creates seven children tasks
- Data intensive

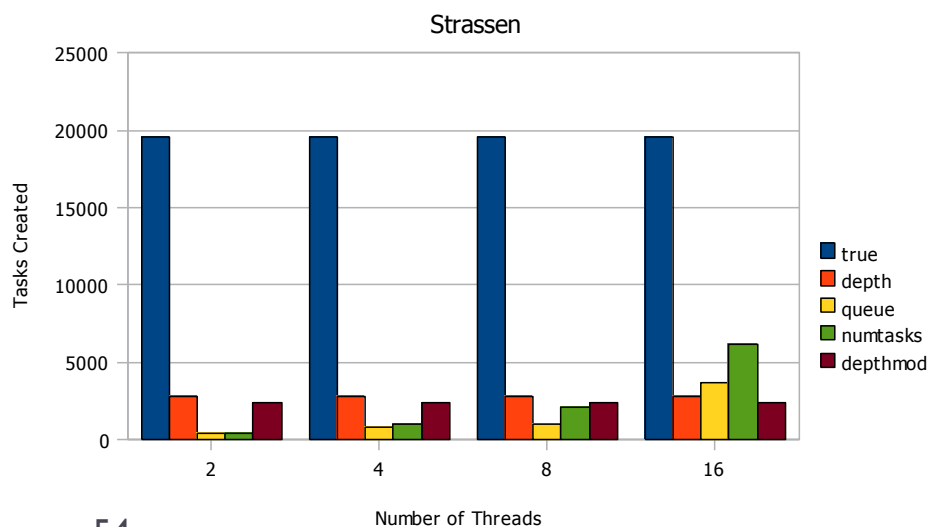
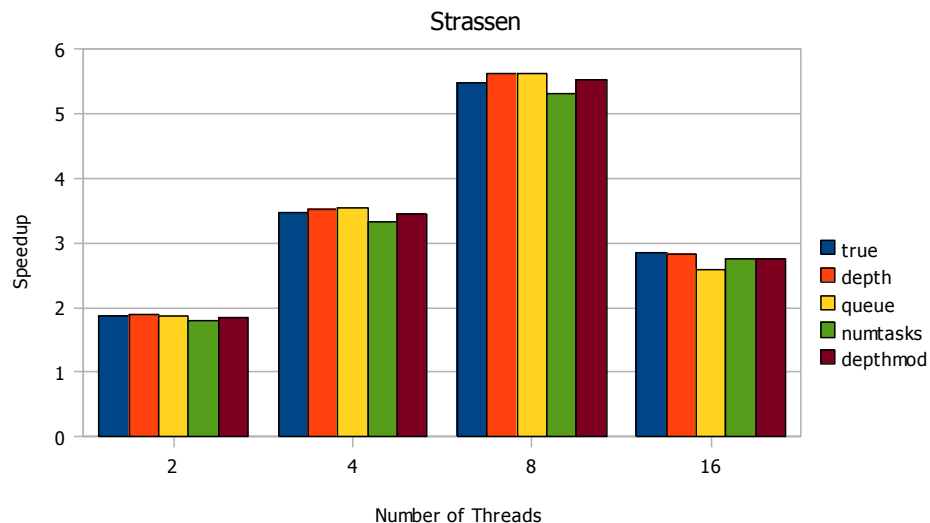
Results

- untied tasks perform slightly better on Octo server
- Cilk performs significantly better on the Altix server
- Large degradation in performance on Octo server

Conclusions

- Performance degradation on Octo is attributed to resource contention between cores on the same chip
- It appears, based on these results and Multisort, that Cilk does not perform well in the presence of multicore architectures where contention comes into play.
- Waiting to gather performance data to help shed light on results

Task Create Condition Strassen



• Results

- until 16 threads, depth and queue perform equally well

• Conclusions

- while true performs slightly less, it creates many more tasks than other methods
 - system can handle lots of tasks, just not millions
- number of tasks needed is relatively small
 - load balancing is not much of an issue for this application