OpenMP ARB, 2007

- 11 permanent members: AMD, Cray, Fujitsu, HP, IBM, Intel, Microsoft, NEC, PGI, SGI, Sun
- 5 temporary members: ASC/LLNL, cOMPunity, EPCC, NASA, RWTH Aachen
- 5 directors: Josh Simons, Sun; Rupak Biswas, NASA; Sanjiv Shah, Intel; Koh Hotta, Fujitsu; Roch Archambault, IBM
- 3 officers: Larry Meadows, CEO, Intel; Nawal Copty, Secretary, Sun; Dave Poulsen, CFO, Intel

OpenMP 3.0

OpenMP ARB

Mark Bull
SC07, November, 2007, Reno

Tasks

- Adding tasking is the biggest addition for 3.0
- Worked on by a separate subcommittee
  - led by Jay Hoeflinger at Intel
- Re-examined issue from ground up
  - quite different from Intel taskq's

General task characteristics

- 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 (potentially 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**

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!**

`task` Construct

```
#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 on a `task` construct

- **When the `if` clause argument is false**
  - The current task region is suspended.
  - The new task is executed immediately by the encountering thread.
  - The suspended task region is not resumed until the new task is complete.
  - 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 barriers, explicit or implicit
  - applies to all tasks generated in the current parallel region up to the barrier
  - matches user expectation
- At a `taskwait` directive
  - applies only to child tasks of the current task, not to further “descendants”

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) ;
     }
   }
}
```

p is firstprivate by default here

Example – parallel pointer chasing on multiple lists using tasks

```
#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: tree traversal, children before parents

```
void traverse(node *p) {
    if (p->left)
      #pragma omp task
        traverse(p->left);
    if (p->right)
      #pragma omp task
        traverse(p->right);
    #pragma omp taskwait
    process(p->data);
}
```

Parent task suspended until child 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)

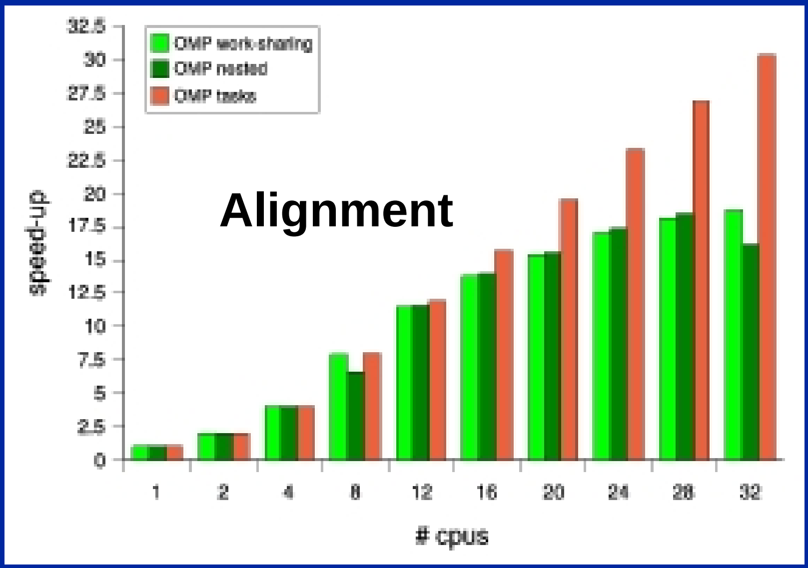
Thread switching

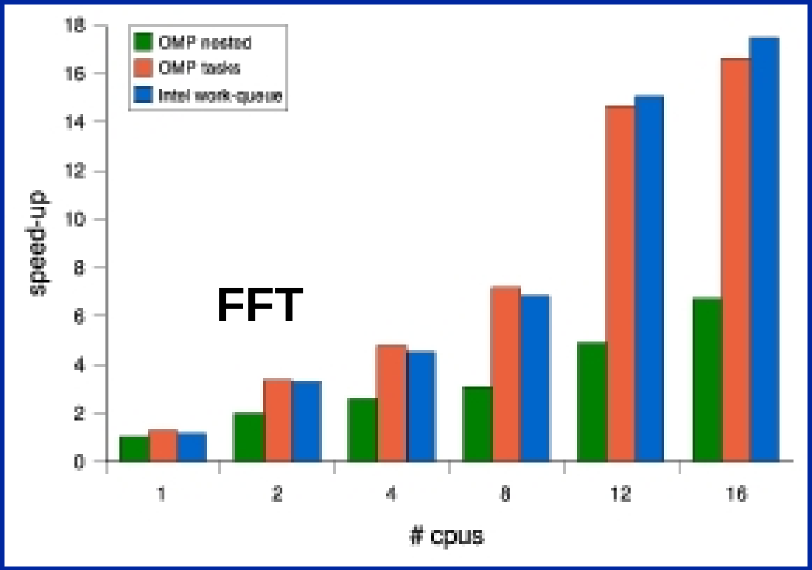
```
#pragma omp single
{
  #pragma omp task untied
    for (i=0; i<ONEZILLION; i++)
      #pragma omp task
        process(item[i]);
}
```

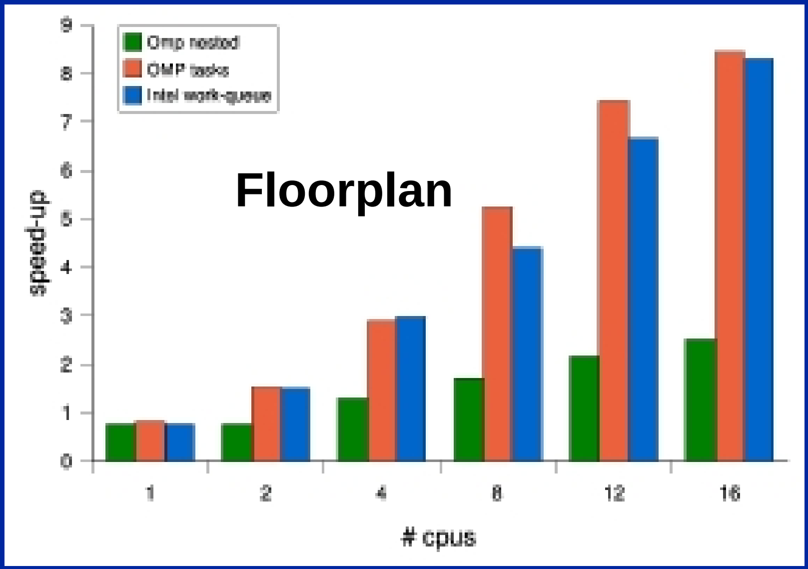
- Eventually, too many tasks are generated
- Generating task is suspended and executing thread switches to a long and boring task
- Other threads get rid of all already generated tasks, and start starving…

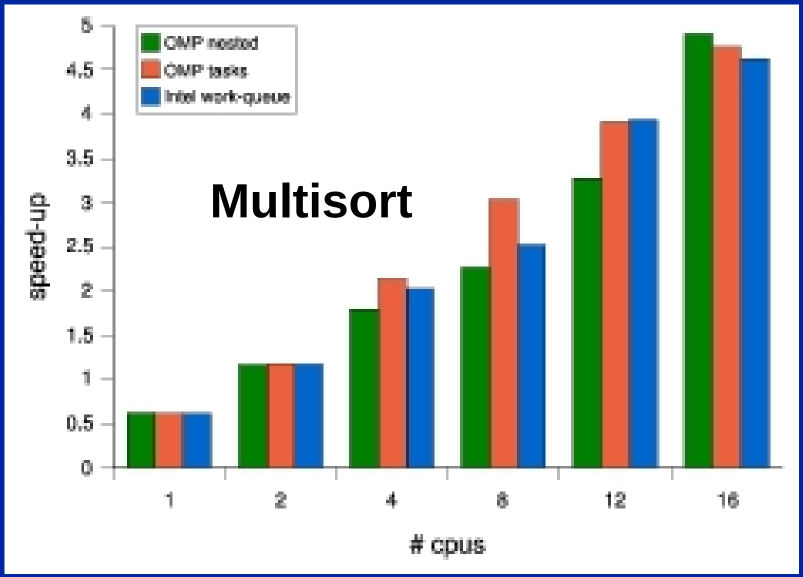
- With thread switching, the generating task can be resumed by a different thread, and starvation is over
- Too strange to be the default: the programmer is responsible!

Performance Results 1



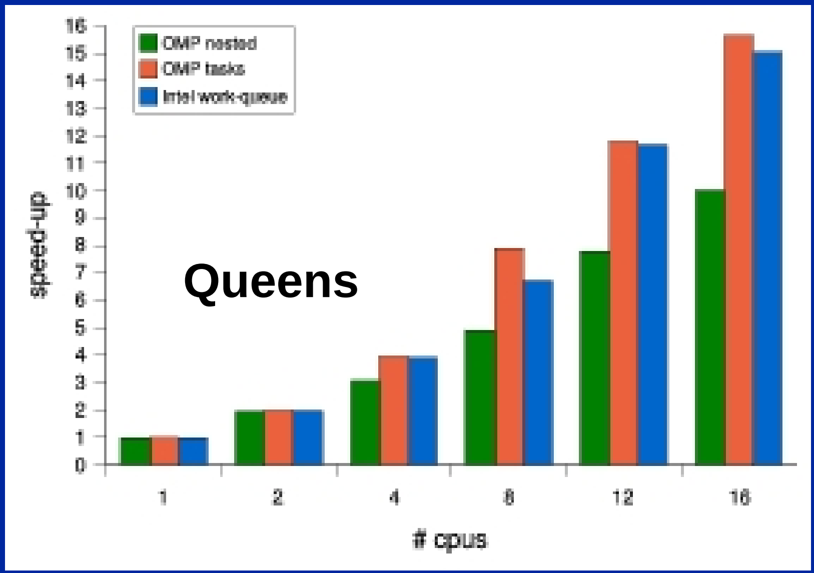


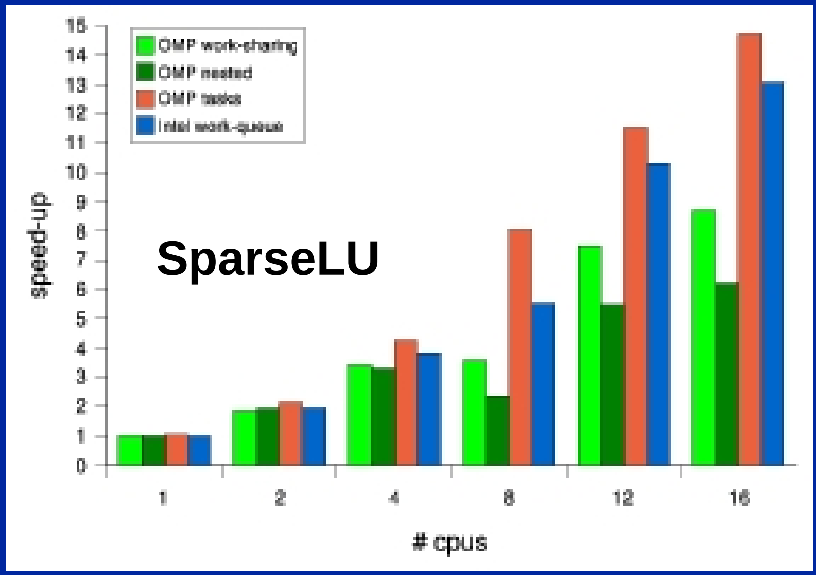


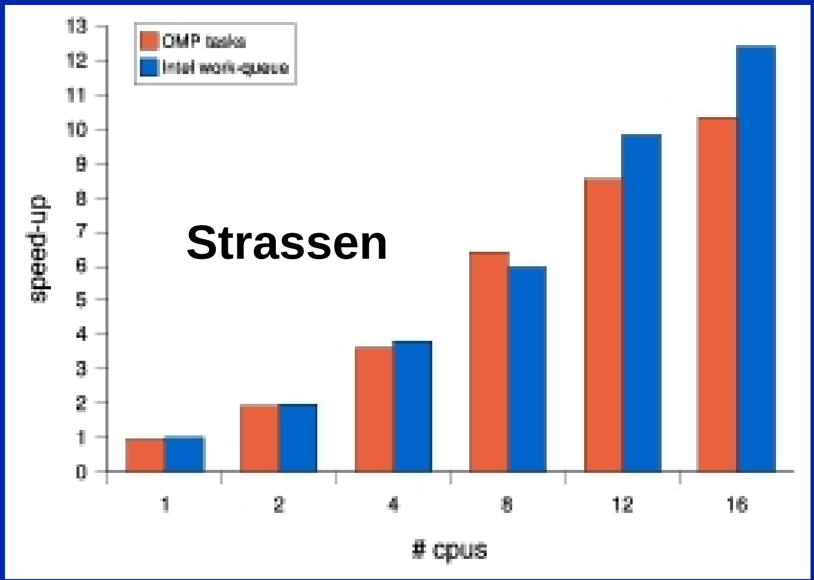


All tests run on SGI Altix 4700 with 128 processors

Performance Results 2







All tests run on SGI Altix 4700 with 128 processors

Reference Implementation

- URL:

  http://mercurium.pc.ac.upc.edu/nanos

- Made by Xavier Teruel, Roger Ferrer, Alex Duran, Eduard Ayguadé, Xavier Martorell

Conclusions on tasks

- Enormous amount of work by many people
- Tightly integrated into 2.5 spec
- Flexible model for irregular parallelism
- Provides balanced solution despite often conflicting goals
- Appears that performance can be reasonable

Better support for nested parallelism

- **Per-thread internal control variables**
  - Allows, for example, calling `omp_set_num_threads()` inside a parallel region.
  - Controls the team sizes for next level of parallelism
- **Library routines to determine depth of nesting, IDs of parent/grandparent etc. threads, team sizes of parent/grandparent etc. teams**

```
omp_get_level()
omp_get_active_level()
omp_get_ancestor_thread_num(level)
omp_get_team_size(level)
```

N.B. new defn. of active parallel region: a parallel region executed by more than one thread

Parallel loops

- Guarantee that this works:

```
!$omp do schedule(static)
do i=1,n
   a(i) = ....
end do
!$omp end do nowait
!$omp do schedule(static)
do i=1,n
  .... = a(i)
end do
```

Loops (cont.)

- Allow collapsing of perfectly nested loops

```
!$omp parallel do collapse(2)
do i=1,n
    do j=1,n
         .....
    end do
end do
```

- Will form a single loop and then parallelise that

Loops (cont.)

- Made `schedule(runtime)` more useful
  - can get/set it with library routines

    `omp_set_schedule()`

    `omp_get_schedule()`
  - allow implementations to implement their own schedule kinds
- Added a new schedule kind AUTO which gives full freedom to the runtime to determine the scheduling of iterations to threads.
- Allowed unsigned ints and C++ RandomAccessIterators as loop control variables in parallel loops

Portable control of threads

- Added environment variable to control the size of child threads' stack

  `OMP_STACKSIZE`

- Added environment variable to hint to runtime how to treat idle threads

  `OMP_WAIT_POLICY`

  `ACTIVE` keep threads alive at barriers/locks

  `PASSIVE` try to release processor at barriers/locks

- Added environment variable and runtime routines to get/set the maximum number of active levels of nested parallelism

  `OMP_MAX_NESTED_LEVELS`

  `omp_set_max_nested_levels()`

  `omp_get_max_nested_levels()`

- Added environment variable to set maximum number of threads in use

  `OMP_THREAD_LIMIT`

  `omp_get_thread_limit()`

Odds and ends

- Disallowed use of the original variable as master thread's private variable
- Made it clearer where/how private objects are constructed/destructed
- Relaxed some restrictions on allocatable arrays
- Plugged some minor gaps in memory model
- Allowed C++ static class members to be threadprivate
- Minor fixes and clarifications to 2.5

Summary

- OpenMP 3.0 is almost ready
- Been a lot of hard work by a lot of people
- We hope you like it: let us know via the public comment process what you think!

Acknowledgements

Bronis de Supinski
Greg Bronevetsky
Dieter an Mey
Christian Terboven
Lei Huang
Barbara Chapman
Alex Duran
Eduard Ayguade
Michael Suess
Gabriele Jost
Mark Bull
Randy Meyer
Kelvin Li
Guansong Zhang

Michael Wong
Priya Unnikrishnan
Diana King
Ernesto Su
Judy Ward
Tim Mattson
Xinmin Tian
Grant Haab
Jay Hoeflinger
Larry Meadows
Sanjiv Shah
Jeff Olivier
Henry Jin
Michael Wolfe

Eric Duncan
Nawal Copty
Yuan Lin
James Beyer
Federico Massaoli
Brian Bliss