Compiler Support of the Workqueuing Execution Model for Intel® SMP Architectures

Ernesto Su, Xinmin Tian, Milind Girkar

Intel Compiler Lab, Intel Corporation
{ernesto.su, xinmin.tian, milind.girkar}@intel.com

Grant Haab, Sanjiv Shah, Paul Petersen

KAI Software Lab, Intel Corporation
{grant.haab, sanjiv.shah, paul.petersen}@intel.com

Why Workqueuing?

```
while(p != NULL){
  do_work(p->data);
  p = p->next;
}
```

```
int n=0, i=0;
NODEPTR q=p;
NODEPTR *r;
while(q != NULL){
  n++;
  q = q->next;
}
r=allocate(n *
       sizeof(NODEPTR));
while(p != NULL){
  r[i++]=p;
  p = p->next;
}
#pragma omp parallel for
for (i=0; i<n; i++)
  do_work(r[i]->data);
free(r);
```

Workqueuing

```
#pragma intel omp parallel taskq
{
  while(p != NULL){
    #pragma intel omp task
      do_work(p->data);
    p = p->next;
  }
}
```

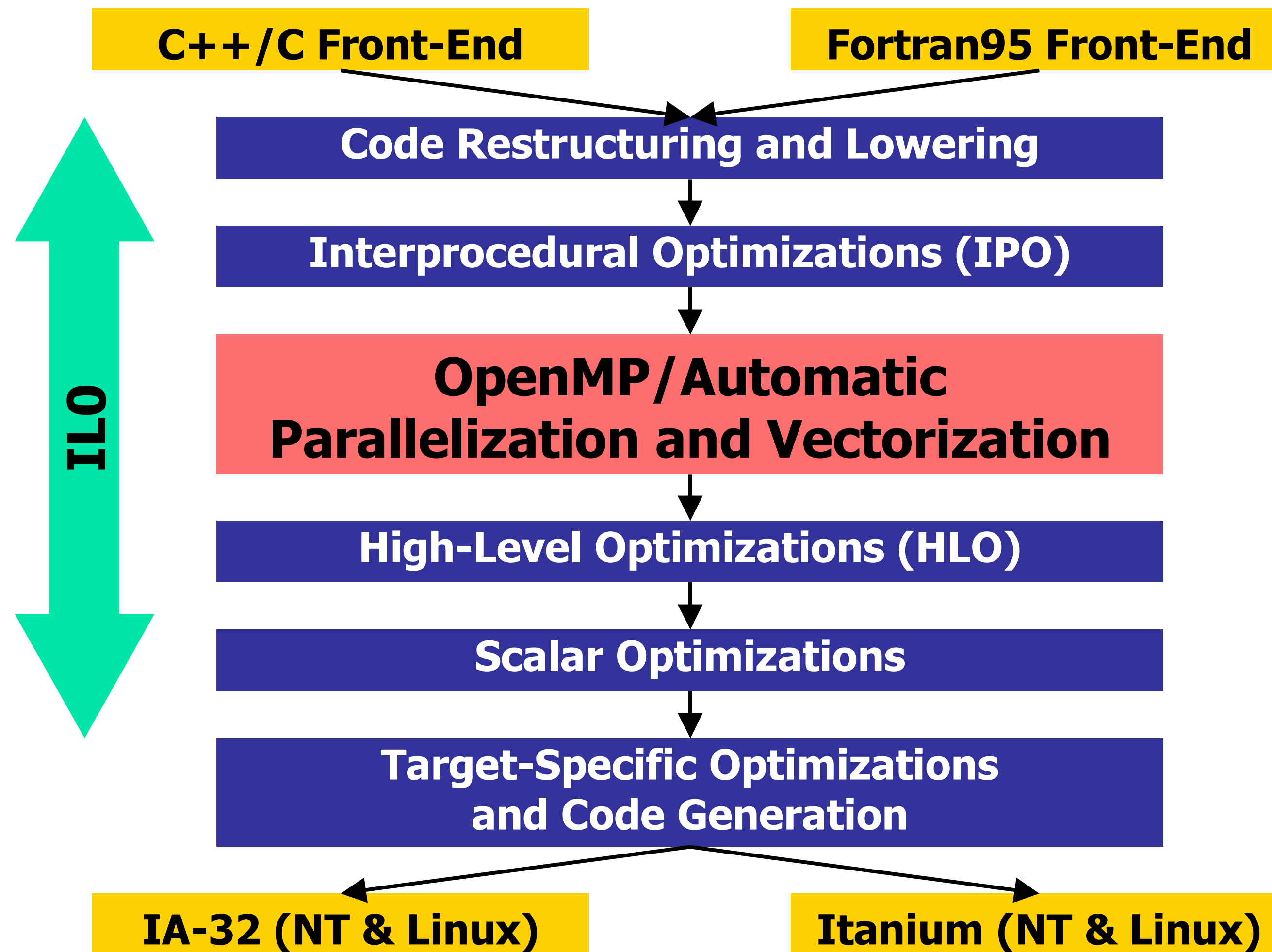
Worksharing vs. Workqueuing

- Worksharing
  - Many common control structures require substantial transformations to sequential code
  - Original sequential semantics lost
  - Can’t handle more complex control structures
- Workqueuing
  - Natural extension to OpenMP
  - Parallelizes more complex control structures (e.g.: recursion) operating on dynamic data
  - Sequential semantics preserved

Outline

- Overview of the Intel Compiler
- Overview of the Workqueuing Model
- Implementation
- Compilation
  - Front End
  - Data Structures
  - Multithreaded Code Generation
- Results
- Conclusions

Compiler Architecture



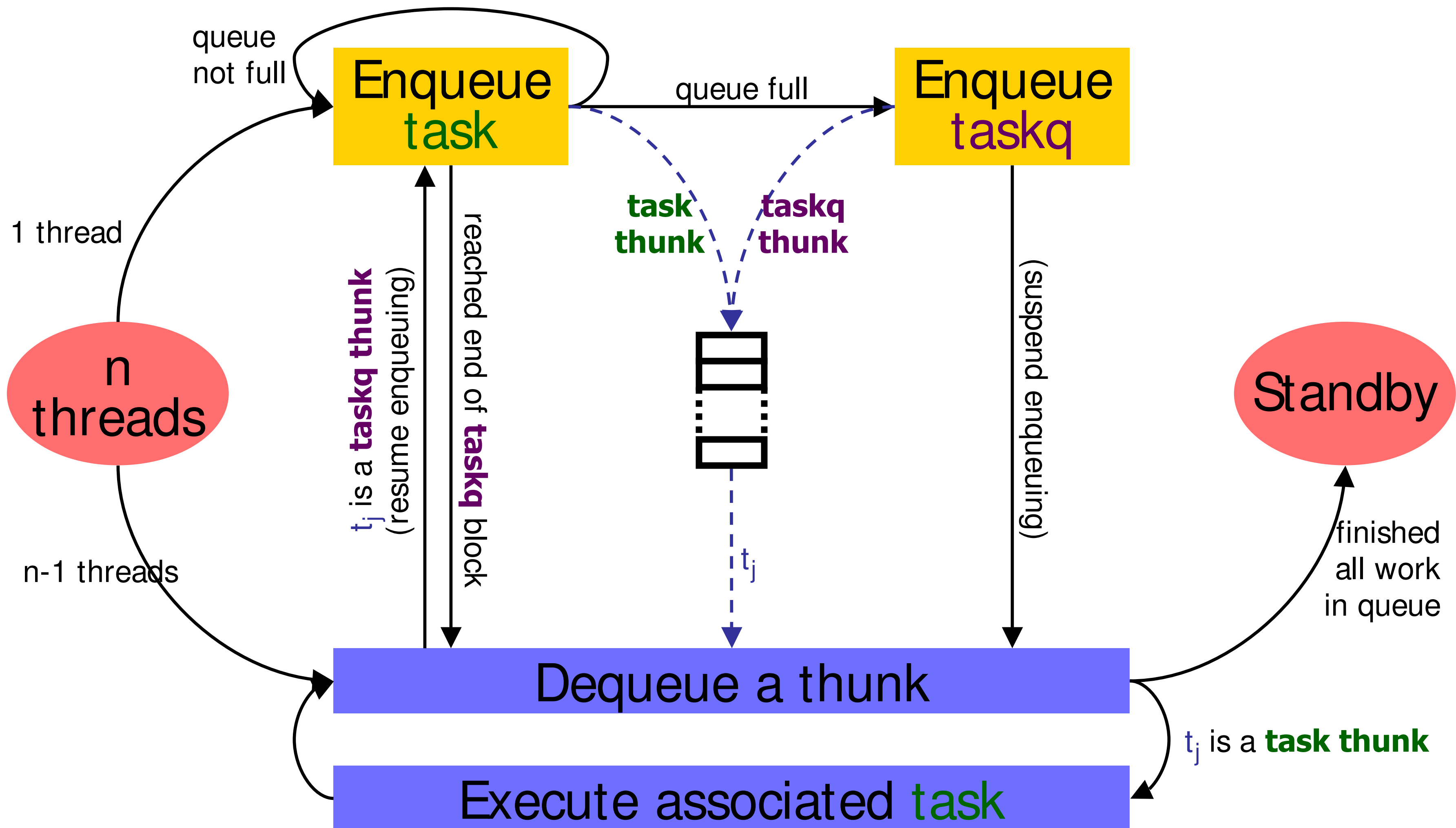
OpenMP Phase

- Pre-pass: **`sections`** $\rightarrow$ **`do`**/**`for`** constructs
- Hierarchical graph builder:
  code regions enclosed in OpenMP constructs
- Loop analyzer: iteration variable, loop bounds
- Variable privatizer
- IL0 code generator
  - Multithreaded code
  - Calls to Intel OpenMP RTL
- Post-pass: thread-local storage

The Workqueuing Model

- Work units need not be all known or pre-computed at the beginning of construct (e.g.: **`while`** loops and recursive functions)
- **`taskq`** specifies an environment (the queue)
  `task` specifies the units of work (dynamic)
- One thread executes the **`taskq`** block, enqueuing each **`task`** it encounters
- All other threads dequeue and execute work from the queue

Multithreaded Execution



Implementation of the WQM

- Seamless integration into existing OpenMP compiler framework
  - Expand IL0 to support new constructs
  - Enable C++ FE to parse new pragmas
  - OpenMP phase
    - Maximize code reuse
      - Unmodified: pre-/post-pass, loop analyzer
      - Slightly modified: graph builder
    - Minimize new code
      - IL0-level multithreaded code generator

Compiling WQ Pragmas

- Example: a simple **`while`** loop
- Parallelized with workqueuing pragmas

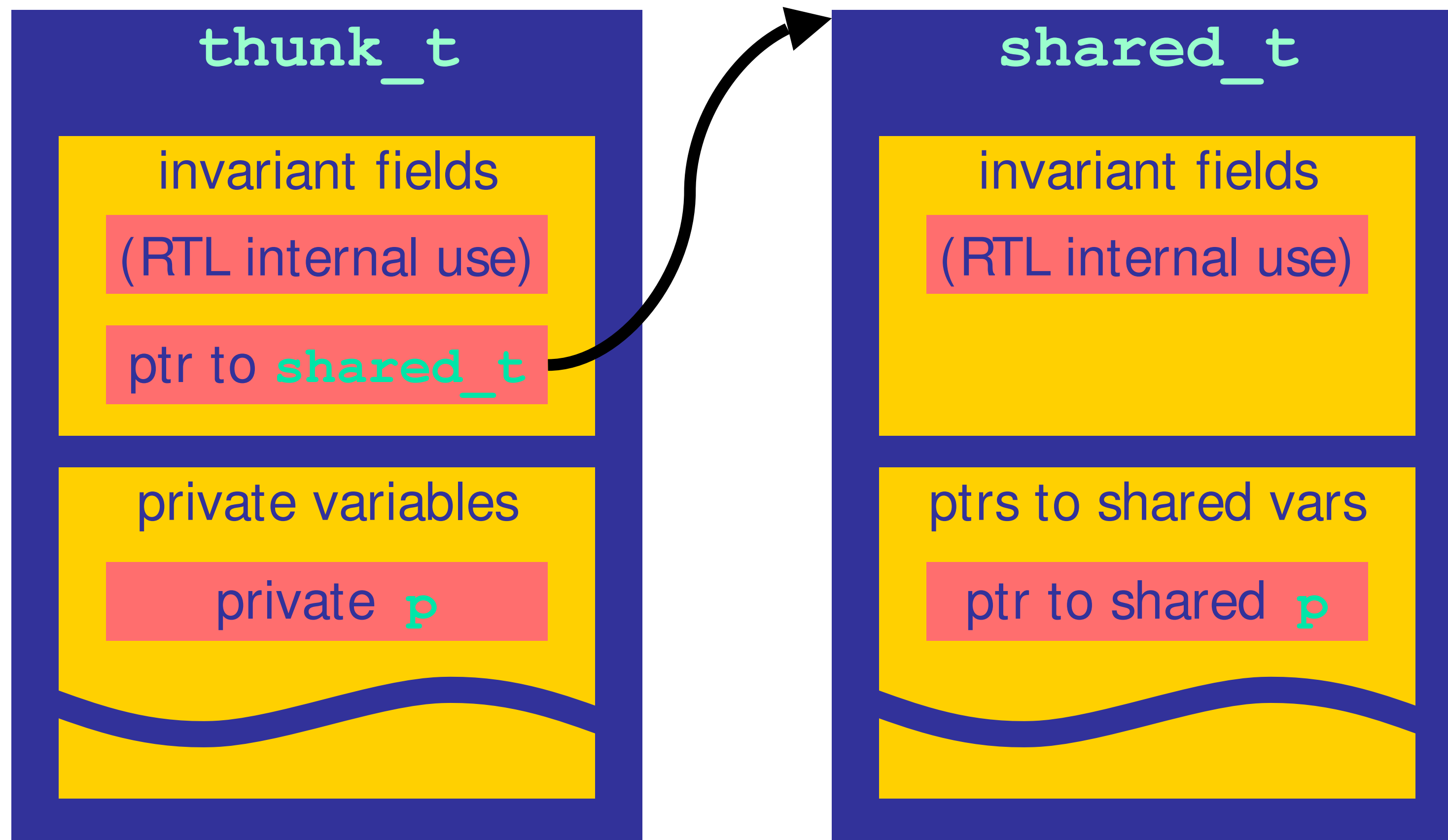
```
#pragma intel omp parallel taskq shared(p)
{
  while (p != NULL)
  {
    #pragma intel omp task captureprivate(p)
      do_work(p);
    p = p->next;
  }
}
```

Front End Support

- IL0 from the C++ FE

```
DIR_PARALLEL_TASKQ SHARED(p)
if (p != 0) {
L1:
  DIR_TASK CAPTUREPRIVATE(p)
    do_work(p)
  DIR_END_TASK
  p = p->next
  if (p != 0) goto L1
}
DIR_END_PARALLEL_TASKQ
```

Data Structures



Parallel Entry

- Starting point executed by all threads

```
test1_par_taskq(p_ptr)
{
  taskq_thunk = __kmpc_taskq(
                test1_taskq, 4, 4, &shareds)
  shareds->p_ptr = p_ptr
  if (taskq_thunk != 0){
    test1_taskq(taskq_thunk)
  }
  __kmpc_end_taskq(taskq_thunk)
}
```

One Thread Enqueues

```
test1_taskq(taskq_thunk) {
 if (*(taskq_thunk->shr->p_ptr) != 0){
L1: task_thunk = __kmpc_task_buffer(
                  taskq_thunk, test1_task)
  task_thunk->p = *(taskq_thunk->shr->p_ptr)
  if (__kmpc_task(task_thunk)!= 0) {
    __kmpc_taskq_task(taskq_thunk,1); return}
  *(taskq_thunk->shr->p_ptr) =
        (*(taskq_thunk->shr->p_ptr))->next
  if (*(taskq_thunk->shr->p_ptr)!= 0) goto L1
 }
 __kmpc_end_taskq_task(taskq_thunk)
}
```

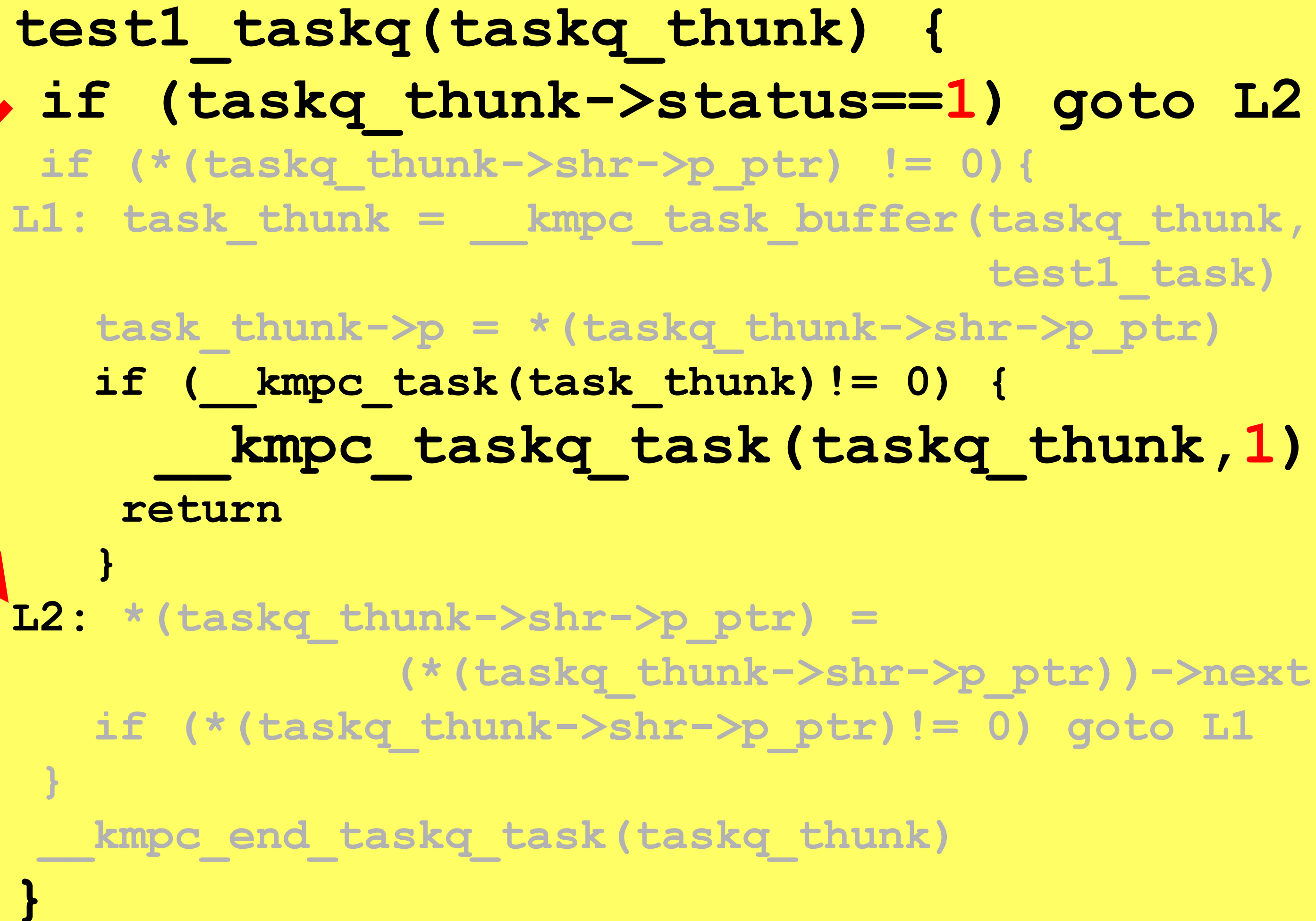
All Other Threads Dequeue

- If dequeued a **`task_thunk`**, perform associated work

```
test1_task(task_thunk)
{
  do_work(task_thunk->p)
}
```

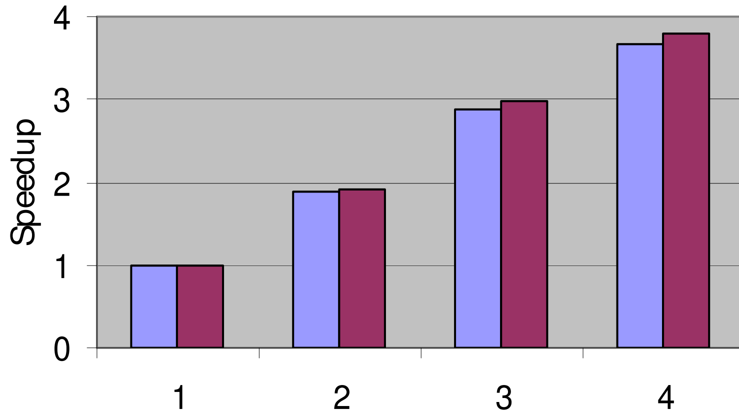
- If dequeued a **`taskq_thunk`**, resume enqueuing
  - Run **`test1_taskq(taskq_thunk)`**
  - May be a different thread

Resuming Execution of **taskq**

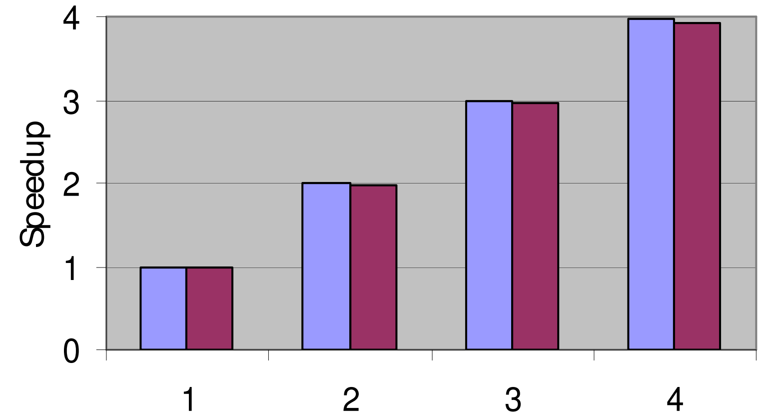


Results

Fibonacci



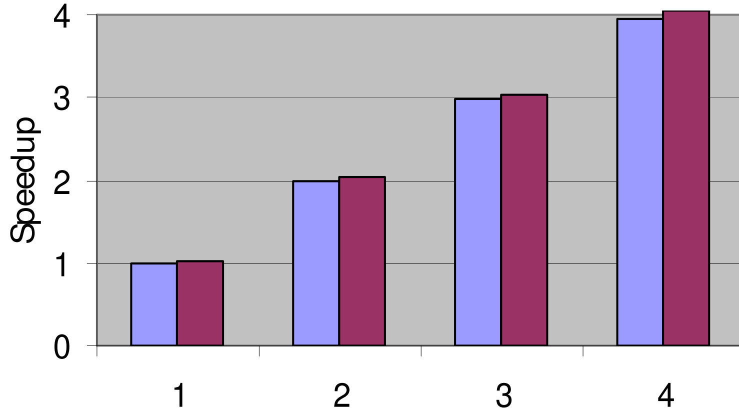
Permanent



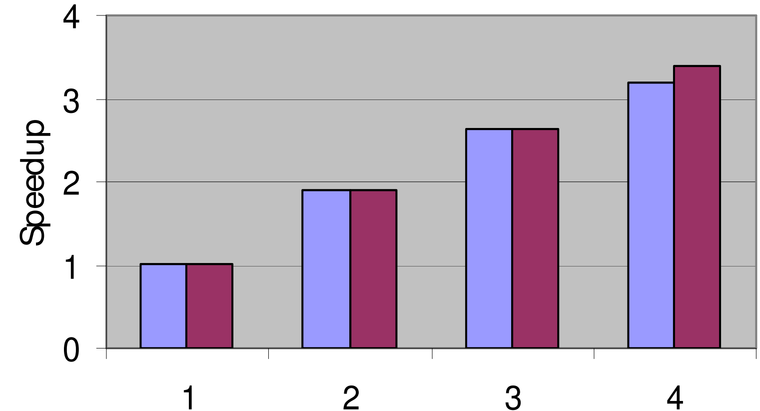
Intel Xeon® 550 MHz

Intel Xeon 1.6 GHz

Queens

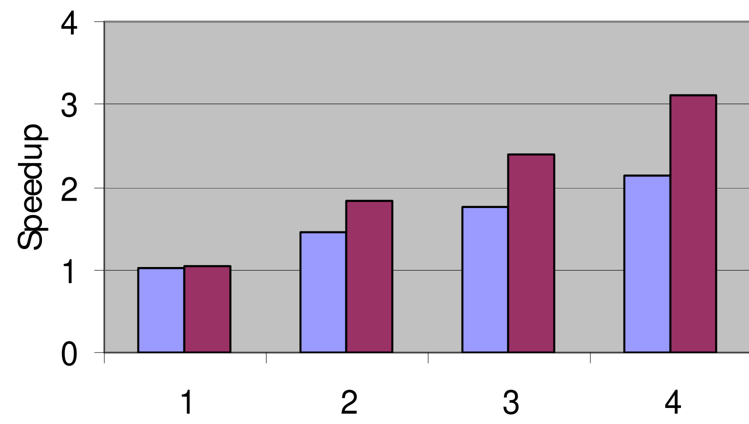


Strassen

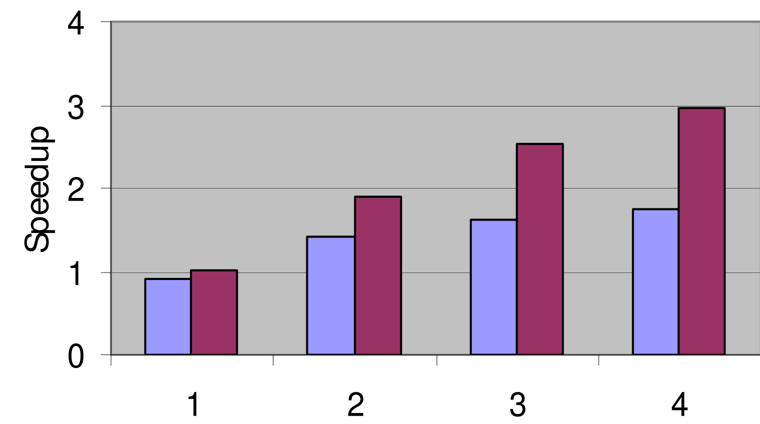


Results (cont.)

FFT



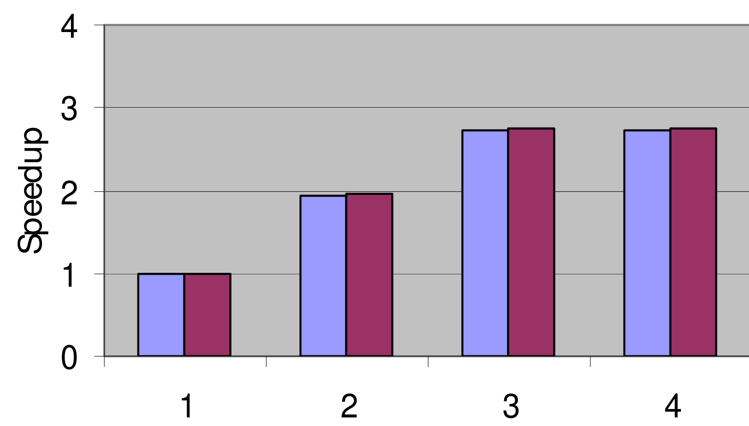
Multisort



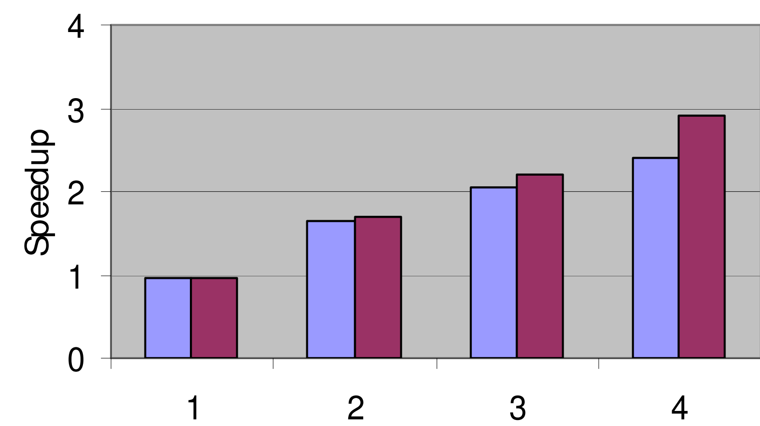
Intel Xeon® 550 MHz

Intel Xeon 1.6 GHz

Pipeline



Quicksort



Conclusions

- Simple extension to augment the scope of applications parallelizable with OpenMP
- Implemented in the Intel Compiler & RTL
  - Seamlessly integrated with OpenMP framework
  - Available in version 7.0
- Good preliminary empirical results with a variety of benchmark applications

Future Work

- More compilers
  - Intel Fortran95 Compiler
- More experiments
- More features
  - An **`if`** clause to control queue depth in recursions
  - New functions to query
    - Nesting level of a queue
    - Number of queues