

An extension to increase tasking control

E. Ayguadé¹, J. Beyer², A. Duran¹, G. Haab³, K. Li⁴, F. Massaioli⁵

¹BSC, ²Cray, ³Intel, ⁴IBM, ⁵CASPUR

June 15th 2010

Outline

- 1 Motivation
- 2 Proposed extensions
- 3 Implementation
- 4 Evaluation
- 5 Proposal 2.0
- 6 Conclusions

Outline

- 1 Motivation
- 2 Proposed extensions
- 3 Implementation
- 4 Evaluation
- 5 Proposal 2.0
- 6 Conclusions

Motivation

Pattern

For recursive problems that perform task decomposition, stop task creation at a certain depth exposes enough parallelism while reducing the overheads to a minimum.

Motivation

Pattern

For recursive problems that perform task decomposition, stop task creation at a certain depth exposes enough parallelism while reducing the overheads to a minimum.

- How good is this pattern in OpenMP?

If clause task creation control

```
1 void nqueens ( int n, int j, char *b, int *sol, int depth ) {
2   ...
3   for ( i = 0; i < n; i++)
4     #pragma omp task untied if(depth < MAX_DEPTH)
5     {
6       /* allocate a temporary array and copy <a> into it */
7       char * b = alloca((j + 1) * sizeof(char));
8       memcpy(b, a, j * sizeof(char));
9       b[j] = i;
10      if (ok(j + 1, b))
11        nqueens(n, j + 1, b, &csols[i], depth+1);
12    }
13   ...
14 }
```

Manual clause task creation control

```
1 void nqueens ( int n, int j, char *b, int *sol, int depth ) {
2     ...
3     for ( i = 0; i < n; i++)
4         if ( depth < MAX_DEPTH ) {
5             #pragma omp task untied
6             {
7                 /* allocate a temporary array and copy <a> into it */
8                 char * b = alloca((j + 1) * sizeof(char));
9                 memcpy(b, a, j * sizeof(char));
10                b[j] = i;
11                if (ok(j + 1, b))
12                    nqueens(n, j + 1, b,&csols[i],depth+1);
13            }
14        } else {
15            a[j] = i;
16            if (ok(j + 1, a))
17                nqueens_ser(n, j + 1, a,&csols[i]);
18        }
19        ...
20 }
```

Manual clause task creation control

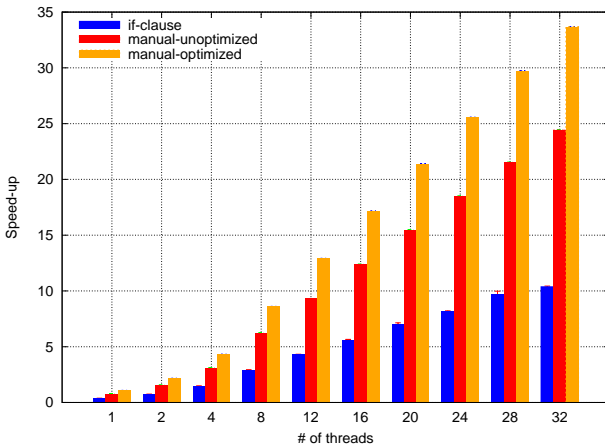
```

1 void nqueens ( int n, int j, char *b, int *sol, int depth ) {
2     ...
3     for ( i = 0; i < n; i++)
4         if ( depth < MAX_DEPTH ) {
5             #pragma omp task untied
6             {
7                 /* allocate a temporary array and copy <a> into it */
8                 char * b = alloca((j + 1) * sizeof(char));
9                 memcpy(b, a, j * sizeof(char));
10                b[j] = i;
11                if (ok(j + 1, b))
12                    nqueens(n, j + 1, b,&csols[i],depth+1);
13            }
14        } else {
15            a[j] = i;
16            if (ok(j + 1, a))
17                nqueens_ser(n, j + 1, a,&csols[i]); ←
18        }
19        ...
20    }

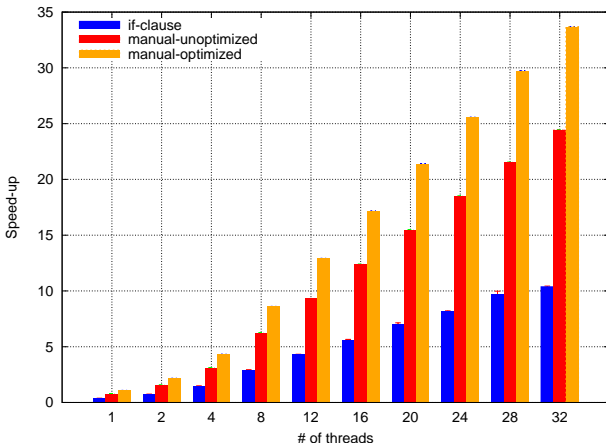
```

It can be optimized

if clause vs manual control



if clause vs manual control



The problem is that the `if` clause still has too much overhead

Outline

- 1 Motivation
- 2 Proposed extensions**
- 3 Implementation
- 4 Evaluation
- 5 Proposal 2.0
- 6 Conclusions

The final clause

Add a new clause to the **task** directive:

```
1 final( expression )
```

When the expression is true

- The task is mark as final
- The **untied** clause is ignored
- All encountered **task** directives are immediately executed
 - with no task created
- The data environment is *inlined*

The final clause

Add a new clause to the **task** directive:

```
1 final( expression )
```

When the expression is true

- The task is mark as final
- The **untied** clause is ignored
- All encountered **task** directives are immediately executed
 - with no task created
- The data environment is *inlined*

Extra API

omp_in_final returns true if called inside the context of a final task

The final clause

The **final** clause can create unexpected side-effects

```
1 void foo (bool arg)
2 {
3     int i = 3;
4     #pragma omp task final(arg) firstprivate(i)
5         i++;
6     printf("%d\n",i); // will print 3 or 4 depending on expr
7 }
```

N Queens

Final version

```

1 void nqueens ( int n, int j, char *b, int *sol, int depth ) {
2   ...
3   for ( i = 0; i < n; i++)
4     #pragma omp task untied final(depth+1 >= MAX_DEPTH)
5     {
6       char *b;
7       int *sols;
8
9       if ( omp_in_final() && depth >= MAX_DEPTH ) {
10        b = a;
11        *sols = sol;
12      } else {
13        /* allocate a temporary array and copy <a> into it */
14        char * b = alloca((j + 1) * sizeof(char));
15        memcpy(b, a, j * sizeof(char));
16        *sols = &csols[i];
17      }
18
19      b[j] = i;
20      if (ok(j + 1, b))
21        nqueens(n, j + 1, b, sols, depth+1);
22    }
23    ...
24 }

```

Outline

- 1 Motivation
- 2 Proposed extensions
- 3 Implementation**
- 4 Evaluation
- 5 Proposal 2.0
- 6 Conclusions

Implementation

We did two different implementations:

- Using a single outline
- Using code cloning

Simple implementation

- If **final** evaluates to true, a flag is set
- Before creating a task,
 - Check the task final flag
 - if false proceed regularly
 - otherwise, just call the outline
- Also, check the flag before doing a **taskwait**

Advanced implementation

- The compiler creates an optimized a serial closure
- if false, proceed regularly
- If the task final flag is true, it invokes the serial closure path

Serial closure

```
1 void foo (int *x)
2 {
3     do_something(x);
4     if ( !omp_in_final() ) {
5         x = allocate()
6     }
7     #pragma omp task
8         foo(x);
9
10    if ( !omp_in_final() ) {
11        x = allocate()
12    }
13    #pragma omp task
14        foo(x);
15
16    #pragma omp taskwait
17 }
18
19 void bar ()
20 {
21     #pragma omp task final(expr)
22         foo(allocate);
23 }
```

Serial closure

```

1 void foo (int *x)
2 {
3     do_something(x);
4     if ( !omp_in_final() ) ← { Always in final, substitute with 1
5         x = allocate()
6     }
7     #pragma omp task ← Won't be created, remove
8         foo(x);
9
10    if ( !omp_in_final() ) {
11        x = allocate()
12    }
13    #pragma omp task
14        foo(x);
15
16    #pragma omp taskwait ← Won't wait for anything, remove
17 }
18
19 void bar ()
20 {
21     #pragma omp task final(expr)
22         foo(allocate);
23 }

```

Serial closure

```
1 void foo__ser (int *x)
2 {
3     do_something(x);
4     if ( !1 ) {
5         x = allocate ()
6     }
7     foo__ser(x);
8
9     if ( !1 ) {
10        x = allocate ()
11    }
12    foo__ser(x);
13 }
14
15 void bar__ser_outline ( int *fp_x )
16 {
17     foo__ser(fp_x);
18 }
```

Serial closure

```
1 void foo__ser (int *x)
2 {
3     do_something(x);
4     foo__ser(x);
5     foo__ser(x);
6 }
7
8 void bar__ser_outline ( int *fp_x )
9 {
10     foo__ser(fp_x);
11 }
```

Outline

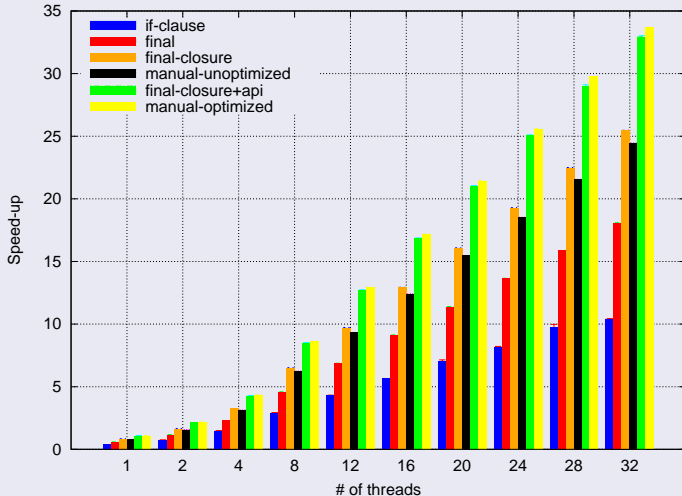
- 1 Motivation
- 2 Proposed extensions
- 3 Implementation
- 4 Evaluation**
- 5 Proposal 2.0
- 6 Conclusions

Experimental setup

- SGI Altix 4700
 - cpuset partition with 32 cpus
- Implemented with the Mercurium/Nanos environment
 - gcc as backend with -O3
- Applications
 - nqueens, fib, floorplan

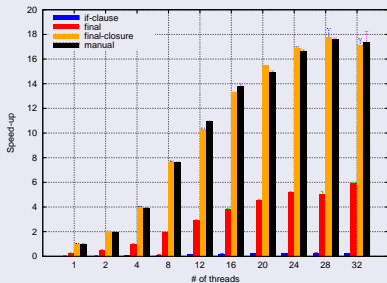
Results

Nqueens

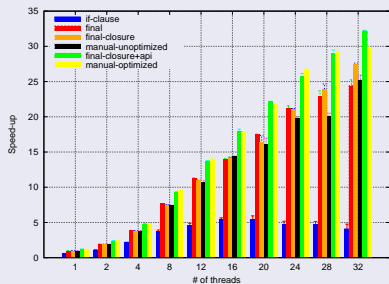


Results

Fib



Floorplan



Outline

- 1 Motivation
- 2 Proposed extensions
- 3 Implementation
- 4 Evaluation
- 5 Proposal 2.0**
- 6 Conclusions

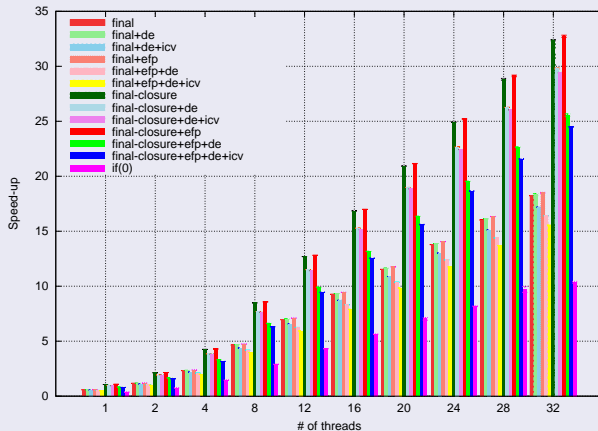
Final is dangerous

We had many concerns about the danger of the final clause

- Virical in nature
- Creates side-effects

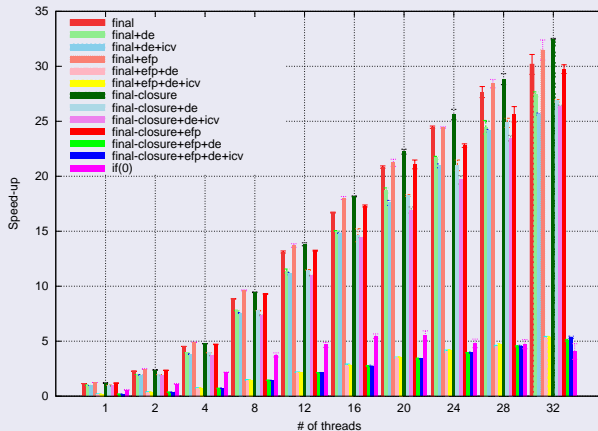
Extra results

Nqueens



Extra results

Floorplan



The inlineable clause

Avoid the dangerous side-effects by allowing to turn them on/off on each task

The inlinable clause

Avoid the dangerous side-effects by allowing to turn them on/off on each task

- The **final** preserves the data environment and ICVs by default
- A task with the **inlinable** clause can get its environment and ICVs not created
 - if executed immediately
 - **final,if**

Outline

- 1 Motivation
- 2 Proposed extensions
- 3 Implementation
- 4 Evaluation
- 5 Proposal 2.0
- 6 Conclusions**

Conclusions

- **final** allows to control task creation efficiently in recursive codes
 - particularly if serial closure is used
- **inlineable** allows to control separately the creation of the data environment
- Under discussion for OpenMP 3.1

The End

Thanks for your attention!