Tasking in OpenMP

Alejandro Duran

Barcelona Supercomputing Center

Outline

Outline

Why task parallelism?

List traversal

Example

```
void traverse_list ( List l )
{
Element e;

#pragma omp parallel private(e)
    for ( e = l->first; e ; e = e->next )
        #pragma omp single nowait
            process(e);
}
```

Without tasks

- Ackward
- Very poor performance
- Not composable

Why task parallelism?

Tree traversal

Example

```
void traverse (Tree *tree)
{
#pragma omp parallel sections
{
#pragma omp section
    if ( tree->left )
      traverse(tree->left);
#pragma omp section
    if ( tree->right )
      traverse(tree->right);
}

process(tree);
}
```

Without tasks

- Too many parallel regions
  - Extra overheads
  - Extra synchronizations
  - Not always well supported

Task parallelism

- Better solution for those problems
- Main addition to OpenMP 3.0[a]
- Allows to parallelize irregular problems
  - unbounded loops
  - recursive algorithms
  - producer/consumer schemes
  - ...

[a] Ayguadé et al., The Design of OpenMP Tasks, IEEE TPDS March 2009

Outline

Outline

What is an OpenMP task?

- Tasks are work units which execution may be deferred
  - they can also be executed immediately!
- Tasks are composed of:
  - code to execute
  - data environment
    - Initialized at creation time
  - internal control variables (ICVs)

Task directive

```
#pragma omp task [clauses]
    structured block
```

- Each encountering thread creates a task
  - Packages code and data environment
- Highly composable. Can be nested
  - inside parallel regions
  - inside other tasks
  - inside worksharings

List traversal

with tasks

Example

```
void traverse_list ( List l )
{
Element e;
for ( e = l->first; e ; e = e->next )
   #pragma omp task
      process(e);
}
```

List traversal

with tasks

Example

```
void traverse_list ( List l )
{
Element e;
for ( e = l->first; e ; e = e->next )
   #pragma omp task
      process(e);
}
```

What is the scope of e?

Outline

Task data scoping

Data scoping clauses

- shared(list)
- private(list)
- firstprivate(list)
  - data is captured at creation
- default(shared|none)

Task data scoping

When there are no clauses ...

If no clause

- Implicit rules apply
  - e.g., global variables are shared
- Otherwise...
  - firstprivate
  - shared attributed is lexically inherited

Task data scoping

In practice...

Example

```
int a;
void foo() {
  int b,c;
  #pragma omp parallel shared(b)
  #pragma omp parallel private(b)
  {
        int d;
        #pragma omp task
        {
            int e;

            a =
            b =
            c =
            d =
            e =
}}}
```

Task data scoping

In practice...

Example

```
int a;
void foo() {
  int b,c;
  #pragma omp parallel shared(b)
  #pragma omp parallel private(b)
  {
        int d;
        #pragma omp task
        {
            int e;

            a = shared
            b =
            c =
            d =
            e =
}}}
```

Task data scoping

In practice...

Example

```
int a;
void foo() {
  int b,c;
  #pragma omp parallel shared(b)
  #pragma omp parallel private(b)
  {
        int d;
        #pragma omp task
        {
            int e;

            a = shared
            b = firstprivate
            c =
            d =
            e =
}}}
```

Task data scoping

In practice...

Example

```
int a;
void foo() {
  int b,c;
  #pragma omp parallel shared(b)
  #pragma omp parallel private(b)
  {
        int d;
        #pragma omp task
        {
            int e;

            a = shared
            b = firstprivate
            c = shared
            d =
            e =
}}}
```

Task data scoping

In practice...

Example

```
int a;
void foo() {
  int b,c;
  #pragma omp parallel shared(b)
  #pragma omp parallel private(b)
  {
        int d;
        #pragma omp task
        {
            int e;

            a = shared
            b = firstprivate
            c = shared
            d = firstprivate
            e =
}}}
```

Task data scoping

In practice...

Example

```
int a;
void foo() {
  int b,c;
  #pragma omp parallel shared(b)
  #pragma omp parallel private(b)
  {
        int d;
        #pragma omp task
        {
            int e;

            a = shared
            b = firstprivate
            c = shared
            d = firstprivate
            e = private
}}}
```

Task data scoping

In practice...

Example

```
int a;
void foo() {
  int b,c;
  #pragma omp parallel shared(b)
  #pragma omp parallel private(b)
  {
        int d;
        #pragma omp task
        {
            int e;

            a = shared
            b = firstprivate
            c = shared
            d = firstprivate
            e = private
}}}
```

Tip

default(none) is your friend

- Use it if you do not see it clear

List traversal

Example

```
void traverse_list ( List l )
{
Element e;
for ( e = l->first; e ; e = e->next )
   #pragma omp task
      process(e);

}
```

e is firstprivate

List traversal

Example

```
void traverse_list ( List l )
{
Element e;
for ( e = l->first; e ; e = e->next )
   #pragma omp task
      process(e);
}
```

how we can guarantee here that the traversal is finished?

Outline

Task synchronization

- Barriers (implicit or explicit)
  - All tasks created by any thread of the current team are guaranteed to be completed at barrier exit
- Task barrier

```
#pragma omp taskwait
```

  - Encountering task suspends until child tasks complete
    - Only direct childs not descendants!

List traversal

Example

```
void traverse_list ( List l )
{
Element e;
for ( e = l->first; e ; e = e->next )
   #pragma omp task
      process(e);

#pragma omp taskwait
}
```

All tasks guaranteed to be completed here

Outline

Task execution model

- Task are executed by a thread of the team that generated it
  - Can be executed immediately by the same thread that creates it
- Parallel regions in 3.0 create tasks!
  - One implicit task is created for each thread
    - So all task-concepts have sense inside the parallel region
- Threads can suspend the execution of a task and start/resume another

List traversal

Example

```
List l

#pragma omp parallel
    traverse_list(l);
```

List traversal

Example

```
List l

#pragma omp parallel
    traverse_list(l);
```

Careful!

Multiple traversals of the same list

List traversal

Single traversal

Example

```
List l

#pragma omp parallel
#pragma omp single
    traverse_list(l);
```

Single traversal

- One thread enters single and creates all tasks
- All the team cooperates executing them

List traversal

Multiple traversals

Example

```
List l[N]

#pragma omp parallel
#pragma omp for
for (i = 0; i < N; i++)
    traverse_list(l[i]);
```

Multiple traversals

- Multiple threads create tasks
- All the team cooperates executing them

Task scheduling

How it works?

- Tasks are tied by default
  - Tied tasks are executed always by the same thread
  - Tied tasks have scheduling restrictions
    - Deterministic scheduling points (creation, synchronization, ...)
    - Another constraint to avoid deadlock problems
  - Tied tasks may run into performance problems
- Programmer can use untied clause to lift all restrictions
  - Note: Mix very carefully with threadprivate, critical and thread-ids

And last...

The IF clause

- If the the expression of a if clause evaluates to false
  - The encountering task is suspended
  - The new task is executed immediately
    - with its own data environment
    - different task with respect to synchronization
  - The parent task resumes when the task finishes
  - Allows implementations to optimize task creation

Outline

Search problem

Example

```
void search (int n, int j, bool *state)
{
    int i,res;

    if (n == j) {
      /* good solution, count it */
      solutions++;
      return;
    }

    /* try each possible solution*/
    for (i = 0; i < n; i++)

    {
       state[j] = i;
       if (ok(j+1,state)) {
         search(n,j+1,state);
       }
    }
}
```

Search problem

Example

```
void search (int n, int j, bool *state)
{
    int i,res;

    if (n == j) {
      /* good solution, count it */
      solutions++;
      return;
    }

    /* try each possible solution*/
    for (i = 0; i < n; i++)
    #pragma omp task
    {
       state[j] = i;
       if (ok(j+1,state)) {
         search(n,j+1,state);
       }
    }
}
```

Search problem

Example

```
void search (int n, int j, bool *state)
{
    int i,res;

    if (n == j) {
      /* good solution, count it */
      solutions++;
      return;
    }

    /* try each possible solution*/
    for (i = 0; i < n; i++)
     #pragma omp task
    {
       state[j] = i;
       if (ok(j+1,state)) {
         search(n,j+1,state);
       }
    }
}
```

Data scoping

Because it's an orphaned task all variables are firstprivate

Search problem

Example

```
void search (int n, int j, bool *state)
{
    int i,res;

    if (n == j) {
      /* good solution, count it */
      solutions++;
      return;
    }

    /* try each possible solution*/
    for (i = 0; i < n; i++)
    #pragma omp task
    {
       state[j] = i;
       if (ok(j+1,state)) {
         search(n,j+1,state);
       }
    }
}
```

Data scoping

Because it's an orphaned task all variables are firstprivate

State is not captured

Just the pointer is captured not the pointed data

Search problem

Example

```
void search (int n, int j, bool *state)
{
    int i,res;

    if (n == j) {
      /* good solution, count it */
      solutions++;
      return;
    }

    /* try each possible solution*/
    for (i = 0; i < n; i++)
     #pragma omp task
    {
       state[j] = i;
       if (ok(j+1,state)) {
         search(n,j+1,state);
       }
    }
}
```

Pitfall #1

Incorrectly capturing pointed data

Pitfall #1

Incorrectly capturing pointed data

Problem

firstprivate does not allow to capture data through pointers

Solutions

1. Capture it manually
2. Copy it to an array and capture the array with firstprivate

Search problem

Example

```
void search (int n, int j, bool *state)
{
    int i,res;

    if (n == j) {
      /* good solution, count it */
      solutions++;
      return;
    }

    /* try each possible solution*/
    for (i = 0; i < n; i++)
    #pragma omp task
    {
       bool *new_state = alloca(sizeof(bool)*n);
       memcpy(new_state,state,sizeof(bool)*n);
       new_state[j] = i;
       if (ok(j+1,new_state)) {
         search(n,j+1,new_state);
       }
    }
}
```

Search problem

Example

```
void search (int n, int j, bool *state)
{
    int i,res;

    if (n == j) {
      /* good solution, count it */
      solutions++;
      return;
    }

    /* try each possible solution*/
    for (i = 0; i < n; i++)
    #pragma omp task
    {
       bool *new_state = alloca(sizeof(bool)*n);
       memcpy(new_state,state,sizeof(bool)*n);
       new_state[j] = i;
       if (ok(j+1,new_state)) {
         search(n,j+1,new_state);
       }
    }
}
```

Caution!

Will new_state still be valid by the time memcpy is executed?

Search problem

Example

```
void search (int n, int j, bool *state)
{
    int i,res;

    if (n == j) {
      /* good solution, count it */
      solutions++;
      return;
    }

    /* try each possible solution*/
    for (i = 0; i < n; i++)
    #pragma omp task
    {
       bool *new_state = alloca(sizeof(bool)*n);
       memcpy(new_state,state,sizeof(bool)*n);
       new_state[j] = i;
       if (ok(j+1,new_state)) {
         search(n,j+1,new_state);
       }
    }
}
```

Pitfall #2

Data can go out of scope!

Pitfall #2

Out-of-scope data

Problem

Stack-allocated parent data can become invalid before being used by child tasks

- Only if not captured with firstprivate

Solutions

1. Use firstprivate when possible
2. Allocate it in the heap
   - Not always easy (we also need to free it)
3. Put additional synchronizations
   - May reduce the available parallelism

Search problem

Example

```
void search (int n, int j, bool *state)
{
    int i,res;

    if (n == j) {
      /* good solution, count it */
      solutions++;
      return;
    }

    /* try each possible solution*/
    for (i = 0; i < n; i++)
    #pragma omp task
    {
       bool *new_state = alloca(sizeof(bool)*n);
       memcpy(new_state,state,sizeof(bool)*n);
       new_state[j] = i;
       if (ok(j+1,new_state)) {
         search(n,j+1,new_state);
       }
    }

    #pragma omp taskwait
}
```

Search problem

Example

```
void search (int n, int j, bool *state)
{
    int i,res;

    if (n == j) {
      /* good solution, c
      solutions++;
      return;
    }

    /* try each possible solution */
    for (i = 0; i < n; i++)
    #pragma omp task
    {
       bool *new_state = alloca(sizeof(bool)*n);
       memcpy(new_state,state,sizeof(bool)*n);
       new_state[j] = i;
       if (ok(j+1,new_state)) {
         search(n,j+1,new_state);
       }
    }

    #pragma omp taskwait
}
```

Shared variable needs protected access

Search problem

Example

```
void search (int n, int j, bool *state)
{
    int i,res;

    if (n == j) {
      /* good solution, count it */
      solutions++;
      return;
    }

    /* try each possible solution*/
    for (i = 0; i < n; i++)
    #pragma omp task
    {
       bool *new_state = alloca(sizeof(bool)*n);
       memcpy(new_state,state,sizeof(bool)*n);
       new_state[j] = i;
       if (ok(j+1,new_state)) {
         search(n,j+1,new_state);
       }
    }

    #pragma omp taskwait
}
```

Solutions

- Use omp critical
- Use omp atomic
- Use a reduction operation
  - Not available for 3.0
  - Can be work out manually

Reductions for tasks

Example

```
int solutions=0;
int mysolutions=0;
#pragma omp threadprivate(mysolutions)

void start_search ()
{
  #pragma omp parallel
  {
      #pragma omp single
      {
          bool initial_state[n];
          search(n,0,initial_state);
      }
      #pragma omp critical
          solutions += mysolutions;
  }

}
```

Use a separate counter for each thread

Accumulate them at the end

Search problem

Example

```
void search (int n, int j, bool *state)
{
    int i,res;

    if (n == j) {
      /* good solution, count it */
      solutions++;
      return;
    }

    /* try each possible solution*/
    for (i = 0; i < n; i++)
    #pragma omp task
    {
       bool *new_state = alloca(sizeof(bool)*n);
       memcpy(new_state,state,sizeof(bool)*n);
       new_state[j] = i;
       if (ok(j+1,new_state)){
         search(n,j+1,new_state);
       }
    }

    #pragma omp taskwait
}
```

Pruning mechanism potentially introduces imbalance in the tree

Search problem

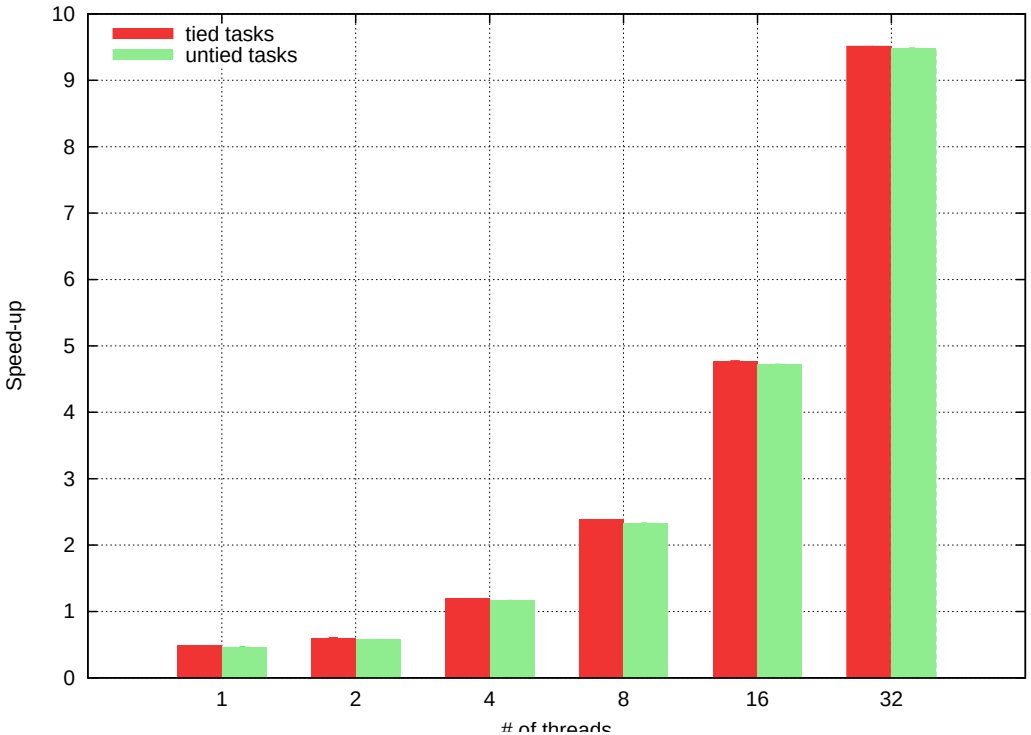
Example

```
void search (int n, int j, bool *state)
{
    int i,res;

    if (n == j) {
      /* good solution, count it */
      solutions++;
      return;
    }

    /* try each possible solution*/
    for (i = 0; i < n; i++)
    #pragma omp task untied
    {
       bool *new_state = alloca(sizeof(bool)*n);
       memcpy(new_state,state,sizeof(bool)*n);
       new_state[j] = i;
       if (ok(j+1,new_state)){
         search(n,j+1,new_state);
       }
    }

    #pragma omp taskwait
}
```

Untied clause

- Allows the implementation to easier load balance

Benefit of untied?



with Intel's icc v11.0

- Don't expect much today.
- But, as implementations are optimized differences may arise

Search problem

Example

```
void search (int n, int j, bool *state)
{
    int i,res;

    if (n == j) {
      /* good solution, count it */
      solutions++;
      return;
    }

    /* try each possible solution */
    for (i = 0; i < n; i++)
    #pragma omp task untied
    {
       bool *new_state = alloca(sizeof(bool)*n);
       memcpy(new_state,state,sizeof(bool)*n);
       new_state[j] = i;
       if (ok(j+1,new_state)) {
         search(n,j+1,new_state);
       }
    }

    #pragma omp taskwait
}
```

Because of untied this is not safe!

Pitfall #3

Unsafe use of untied tasks

Problem

Because tasks can migrate between threads at any point thread-centric constructs can yield unexpected results

Remember

When using untied tasks avoid:

- Threadprivate variables
- Any thread-id uses

And be very careful with:

- Critical regions (and locks)

Simple solution

Create a task tied region with #pragma omp task if(0)

Search problem

Example

```
void search (int n, int j, bool *state)
{
    int i,res;

    if (n == j) {
      /* good solution, count it */
      #pragma omp task if(0)
      solutions++;
      return;
    }

    /* try each possible solution*/
    for (i = 0; i < n; i++)
    #pragma omp task untied
    {
       bool *new_state = alloca(sizeof(bool)*n);
       memcpy(new_state,state,sizeof(bool)*n);
       new_state[j] = i;
       if (ok(j+1,new_state)) {
         search(n,j+1,new_state);
       }
    }

    #pragma omp taskwait
}
```

Now this statement is tied and safe

Task granularity

Granularity is a key performance factor

- Tasks tend to be fine-grained
- Try to “group“ tasks together
  - Use if clause or manual transformations

Using the if clause

Example

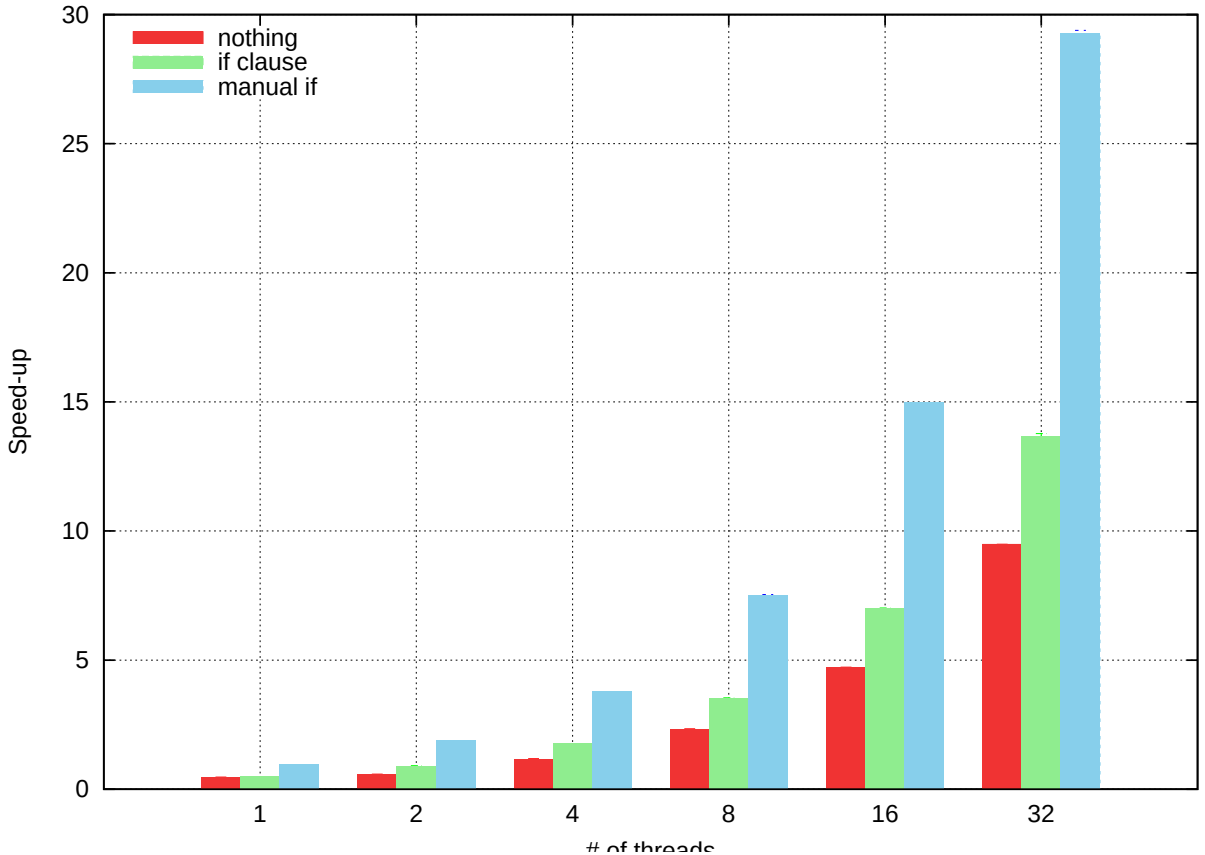
```
void search (int n, int j, bool *state, int depth)
{
    int i,res;

    if (n == j) {
      /* good solution, count it */
      #pragma omp task if(0)
      mysolutions++;
      return;
    }

    /* try each possible solution*/
    for (i = 0; i < n; i++)
    #pragma omp task untied if(depth < MAX_DEPTH)
    {
       bool *new_state = alloca(sizeof(bool)*n);
       memcpy(new_state,state,sizeof(bool)*n);
       new_state[j] = i;
       if (ok(j+1,new_state)) {
         search(n,j+1,new_state,depth+1);
       }
    }
    #pragma omp taskwait
}
```

Using an if statement

Example

```
void search (int n, int j, bool *state, int depth)
{
    int i,res;

    if (n == j) {
      /* good solution, count it */
      #pragma omp task if(0)
      mysolutions++;
      return;
    }

    /* try each possible solution*/
    for (i = 0; i < n; i++)
    #pragma omp task untied
    {
       bool *new_state = alloca(sizeof(bool)*n);
       memcpy(new_state,state,sizeof(bool)*n);
       new_state[j] = i;
       if (ok(j+1,new_state)) {
         if ( depth < MAX_DEPTH )
            search(n,j+1,new_state,depth+1);
         else
            search_serial(n,j+1,new_state);
       }
    }
    #pragma omp taskwait
}
```

If clause vs If statement



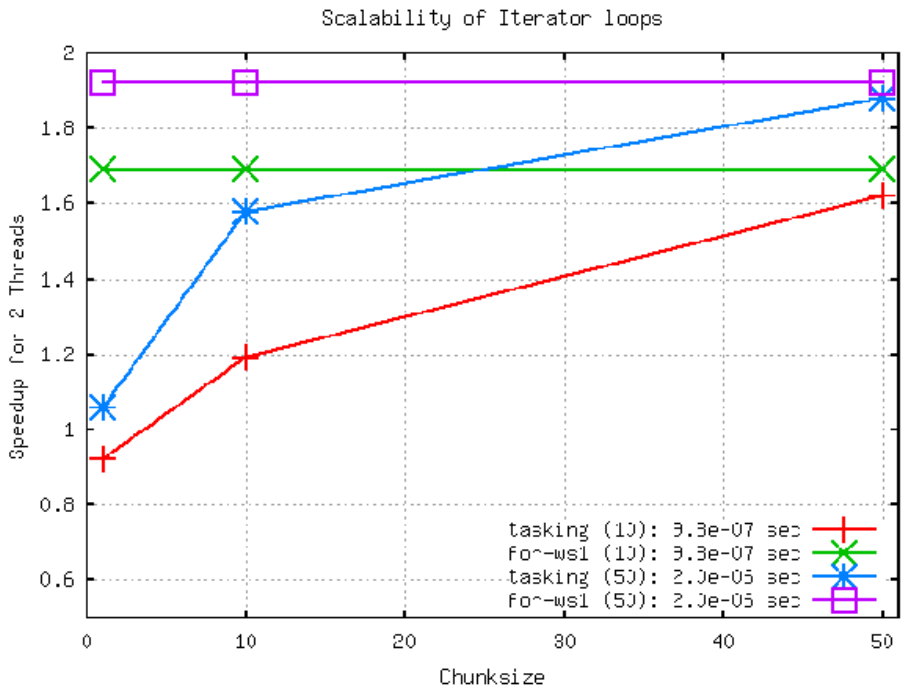
with Intel's icc v11.0

- If clause reduces overheads
  - without modifying the code
- but if granularity is very small is not enough

Don’t abuse tasks

Tasks are nice but...

- They are not the answer to everything
  - They are more costly than other OpenMP mechanisms
- Use other OpenMP constructs when appropriate
  - Particularly for/do worksharing and sections



Courtesy of Christian Terboven

Outline

Summary

Tasks in 3.0

- #pragma omp task [clauses] creates a task
  - shared,firstprivate,private data clauses
    - firstprivate is usually the default (but shared inherited)
  - untied allows tasks to move between threads
  - if allows to dynamically control task creation
- #pragma omp taskwait waits for children completation

Summary

Pitfalls & tips

1. Use default(none) if unsure of data scoping
2. Careful when using firstprivate on pointers
3. Careful with Out-of-scope data
4. Use untied tasks carefully
5. Control granularity
6. Do not abuse of tasks

Tasks after 3.0

In the works

- Support for task reductions
- Task dependences
- Scheduling hints

The End

Thanks for your attention!