A Static Cut-off for Task Parallel Programs

Shintaro Iwasaki, Kenjiro Taura
Graduate School of Information Science and Technology
The University of Tokyo

September 12, 2016 @ PACT '16
Short Summary

- We focus on a **fork-join task parallel programming model**.

- “**Cut-off**” is an optimization technique for task parallel programs to control granularity.

- Previous cut-off systems have been **dynamic**, and have issues and limitations (detailed later.)

**Keyword:** divide-and-conquer
Short Summary

- We focus on a fork-join task parallel programming model.
  - "Cut-off" is an optimization technique for task parallel programs to control granularity.
- Previous cut-off systems have been dynamic, and have issues and limitations (detailed later.)
- We propose a static cut-off method and further compiler optimization techniques based on it.
- Evaluation shows good performance improvement.
  - 8x speedup on average compared to the original.

Keyword: divide-and-conquer
Index

0. Short Summary
1. Introduction
   - What is task parallelism?
   - What is a “cut-off”?
   - Related work: dynamic cut-off
2. Proposal: Static Cut-off
3. Evaluation
4. Conclusion
Importance of Multi-threading

- The number of CPU cores is increasing.
- **Multi-threading** is an essential idea to exploit modern processors.
  → A task parallel model is one of the most promising parallel programming models.

![Graph showing the increase in the number of CPU cores from 2000 to 2014.](http://cpudb.stanford.edu/)
Task Parallel Programming Models

- Task parallelism is a popular parallel programming model.
  - Adopted by many famous systems/libraries:
    - e.g., OpenMP (since ver. 3.0), Cilk / Cilk Plus, Intel TBB …

- It has two major features:
  - Dynamic load balancing
  - Suitability for divide-and-conquer algorithms

- In this talk, we focus on a “fork-join task parallel model.”
Fork-join Task Parallelism

- We use program examples given in Cilk syntax.
  
  - Two basic keywords are provided to express task parallelism: `spawn` and `sync`.
    - **Spawn (≔ fork)**: create a task as a child, which will be executed concurrently.
    - **Sync (≕ join)**: wait all tasks created (or spawned) by itself.

```c
void vecadd(float* a, float* b, int n){
    for(int i = 0; i < n; i++)
        a[i] += b[i];
}
```

```c
void vecadd(float* a, float* b, int n){
    if(n == 1){
        *a += *b;
    }else{
        spawn vecadd(a, b, n/2);
        spawn vecadd(a+n/2, b+n/2, n-n/2);
        sync;
    }
}
```

Same meaning.
Fork-join Task Parallelism

- We use program examples given in Cilk syntax.
- Two basic keywords are provided to express task parallelism: `spawn` and `sync`.
  - Spawn (≒ fork) : create a task as a child, which will be executed concurrently.
  - Sync (≒ join) : wait all tasks created (or spawned) by itself.
- The main target is a divide-and-conquer algorithm.
  - e.g., sort, FFT, FMM, AMR, cache-oblivious GEMM

```c
void vecadd(float* a, float* b, int n){
    if(n == 1){
        *a += *b;
    }else{
        spawn vecadd(a, b, n/2);
        spawn vecadd(a+n/2, b+n/2, n-n/2);
        sync;
    }
}
```
Overheads of Task Parallel Program

- In general, task parallel runtime is designed to handle fine-grained parallelism efficiently.
- However, extreme fine granularity imposes large overheads, degrading the performance.

```c
void vecadd(float* a, float* b, int n){
    if(n == 1){
        *a += *b;
    }else{
        spawn vecadd(a, b, n/2);
        spawn vecadd(a+n/2, b+n/2, n-n/2);
        sync;
    }
}
```

This `vecadd` is a too fine-grained task; one leaf task only calculates `*a += *b`. 
Overheads of Task Parallel Program

- In general, task parallel runtime is designed to handle fine-grained parallelism efficiently.
- However, extreme fine granularity imposes large overheads, degrading the performance.
- Cut-off has been known as an effective optimization technique.

```c
void vecadd(float* a, float* b, int n){
  if(n == 1){
    *a += *b;
  }else{
    spawn vecadd(a, b, n/2);
    spawn vecadd(a+n/2, b+n/2, n-n/2);
    sync;
  }
}
```

This vecadd is a too fine-grained task; one leaf task only calculates \(*a += *b\).
Cut-off: An Optimization Technique

- **Cut-off** is a technique to reduce a tasking overhead by stop creating tasks in a certain condition.
  - i.e., execute a task in serial in that condition.

```c
void vecadd(float* a, float* b, int n){
  if(n == 1){
    *a += *b;
  }else{
    spawn vecadd(a, b, n/2);
    spawn vecadd(a+n/2, b+n/2, n-n/2);
    sync;
  }
}
```

Call a **sequential vecadd** if $1 \leq n \&\& n \leq 1000$

```
void vecadd_seq(float* a, float* b, int n){
  if(n == 1){
    *a += *b;
  }else{
    /*spawn*/vecadd_seq(a, b, n/2);
    /*spawn*/vecadd_seq(a+n/2, b+n/2, n-n/2);
    /*sync;*/
  }
}
```

- Programmers commonly apply it manually.
Cut-off + Further Optimizations

1. Cut-off

```c
void vecadd(float* a, float* b, int n){
    if(n == 1){
        *a += *b;
    }else{
        spawn vecadd(a, b, n/2);
        spawn vecadd(a+n/2, b+n/2, n-n/2);
        sync;
    }
}
```

2. Transformation

- In addition to reducing tasking overheads, further transformations are applicable to serialized tasks in some cases.

```c
void vecadd_seq(float* a, float* b, int n){
    for(int i = 0; i < n; i++)
        a[i] += b[i];
}
```
Cut-off + Further Optimizations

1. Cut-off

- **Automatic cut-off** addresses these problems.
  - Find a **cut-off condition** automatically.
  - **Serialize** a task function after a cut-off.
  - And, even **optimize** the serialized function

2. Transformation

- In addition to reducing tasking overheads, **further optimizations** are applicable to serialized tasks in some cases.

... by just writing naïve task parallel programs.
Our Proposal: Static Cut-off

- We propose a compiler optimization technique of an automatic cut-off including further optimizations for task parallel programs without any manual cut-off.

```c
void vecadd(float* a, float* b, int n){
    if(1 <= n && n <= 4096){
        vecadd_seq(a, b, n);
    }else{
        spawn vecadd(a, b, n/2);
        spawn vecadd(a+n/2, b+n/2, n-n/2);
        sync;
    }
}

void vecadd_seq(float* a, float* b, int n){
    // Vectorize the following for-loop,
    // since task keywords implicitly reveal
    // each iteration is independent.
    for(int i = 0; i < n; i++)
        a[i] += b[i];
}
```
Our Proposal: Static Cut-off

- We propose a compiler optimization technique of an automatic cut-off including further optimizations for task parallel programs without any manual cut-off.

Compiler optimizations for simple loops have been well developed.
- Loop blocking, unrolling interchange, etc...

→ Develop optimizations for divide-until-trivial tasks.
Index

0. Short Summary

1. Introduction

2. Proposal: Static Cut-off
   - What cut-off condition is used?
   - How about further optimizations after cut-off?

3. Evaluation

4. Conclusion
Dynamic Cut-off (1/2)

- Most previous studies on automatic cut-off [*1,*2,*3] focus on adaptive cut-off (dynamic cut-off)
  - Dynamic cut-off is a technique not creating tasks when runtime information tells task creation is not beneficial.

- Runtime information:
  a total number of tasks, task queue length, execution time, depth of tasks, frequency of work stealing etc...

State-of-the-art Dynamic Cut-off

- One proposed by Thoman et al. [*] is state-of-the-art.
  - For each spawns, call/create either
    1. an original task
    2. a task inlined some times
    3. a fully serialized task

  which is decided by runtime information.

- e.g., task queue length

If tasks are likely to exist abundantly, it runs a fully serialized task instead.

Dynamic Cut-off (2/2)

- Most previous studies on automatic cut-off [*1,*2,*3] were dynamic cut-off.
  - Dynamic cut-off is a technique serializing tasks when runtime information tells task creation is not beneficial.

- Compared to dynamic cut-off, our static cut-off has two major advantages.
  1. Cost to evaluate a cut-off condition is low.
  2. More aggressive optimizations are likely to be applied.

Key Idea: Cut-off Near Leaves

- Aggregate tasks near leaves.
  - Low risk of serious loss of parallelism.
  - Chance to apply powerful compiler optimizations after cut-off.

- Our compiler tries to determine a condition under which the recursion stops within a certain height.
Height of Task

- Consider a task tree of fib(16) below.

  fib calculates $F_n = \begin{cases} 
  n & \text{if } n < 2 \\
  F_{n-1} + F_{n-2} & \text{otherwise} 
  \end{cases}$

- Height is difficult to obtain, but it is suitable for a cut-off condition.

```c
void fib(int n, int* r){
    if(n < 2){
        *r = n;
    }else{
        int a, b;
        spawn fib(n-1, &a);
        spawn fib(n-2, &b);
        sync;
        *r = a + b;
    }
}
```

Cut-off in “height < 3”
Transformation Flow

1. Try to obtain a cut-off condition.
2. Optimize a task after cut-off.
Transformation Flow

1. Try to obtain a cut-off condition.

2. Optimize a task after cut-off.
How to Implement it?

1. Try to obtain a cut-off condition.

→ Try to calculate “the $H$th termination condition”
the condition in which a task ends within a height $H$.

Key idea.

For example, the 2nd termination condition of fib is “n <= 3”
How to Implement it?

1. Try to obtain a cut-off condition.
   → Try to calculate “the $H$th termination condition”
   the condition in which a task ends within a height $H$.

2. Optimize task after cut-off.
   → Compiler optimizations:
   apply one of them.
   1. Static task elimination
   2. Code-bloat-free inlining
   3. Loopification

   $H$: Height

   Key idea.

   Reduce tasking overheads.
   + Reduce function-calling overheads.
   + Simplify control flow.
**Static Cut-off Flow**

- Our developed system...
  1. calculates the $H$th termination condition.
  2. decides a height $H$ using heuristics.
  3. applies one of the compiler optimizations:
    - 3a Static task elimination
    - 3b Code-bloat-free inlining
    - 3c Loopification

![Diagram showing the static cut-off flow with decision points for identifying the $H$th termination condition, applying loopification, and code-bloat-free inlining.](image-url)
Termination Condition

- Consider a fibonacci task.
  - Compute as $F_n = \begin{cases} n & \text{if } n < 2 \\ F_{n-1} + F_{n-2} & \text{otherwise} \end{cases}$

- (0th) termination condition is a condition in which tasks never create a child task.

These tasks themselves never create a child, so $n < 2$ is a termination condition.

```c
void fib(int n, int* r){
    if(n < 2){
        *r = n;
    }else{
        int a, b;
        spawn fib(n-1, &a);
        spawn fib(n-2, &b);
        sync;
        *r = a + b;
    }
}
```
**H**th Termination Condition

- Consider a fibonacci task.
  - Compute as \( F_n = \begin{cases} n & \text{if } n < 2 \\ F_{n-1} + F_{n-2} & \text{otherwise} \end{cases} \)
- **H**th termination condition is a condition in which tasks only create a child task within a height \( H \).

The tasks create a task at most within height 2, so \( n < 4 \) is a 2nd termination condition.

```c
void fib(int n, int* r){
    if (n < 2){
        *r = n;
    }else{
        int a, b;
        spawn fib(n-1, &a);
        spawn fib(n-2, &b);
        sync;
        *r = a + b;
    }
}
```
Termination Condition Analysis

• A 0th termination condition is a condition in which tasks never create children.
  – A simple basic block analysis tells \( n < 2 \) is such a condition for \( \text{fib} \) example.

• An \( H \)th termination condition is recursively calculated by using an \((H-1)\)th termination condition.
  – It requires a simple algebra solver.

```c
void fib(int n, int* r){
    if(n < 2){
        *r = n;
    }else{
        int a, b;
        spawn fib(n-1, &a);
        spawn fib(n-2, &b);
        sync;
        *r = a + b;
    }
}
```
Determining Cut-off Height $H$

- Basically, choose the larger $H$.
  - a. a height which makes the number of cycles after cut-off is less than 5000 cycles.
  - Task creation takes approximately 100 cycles.

<table>
<thead>
<tr>
<th>CPU</th>
<th>Frequency</th>
<th>Task Creation Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Xeon E7540</td>
<td>2.0GHz</td>
<td>36.0 [ns/task]</td>
</tr>
<tr>
<td>AMD Opteron 6380</td>
<td>2.5GHz</td>
<td>44.9 [ns/task]</td>
</tr>
<tr>
<td>Intel Xeon E5-2695 v2</td>
<td>2.4GHz</td>
<td>21.5 [ns/task]</td>
</tr>
<tr>
<td>Intel Xeon E5-2699 v3</td>
<td>2.3GHz</td>
<td>33.8 [ns/task]</td>
</tr>
</tbody>
</table>

- We use the LLVM's cost function for estimation, which is not so accurate, but seems sufficient for this use.
  - b. 4 (constant)

It is designed for very fine-grained tasks.

It is a minimum cut-off height.

[MassiveThreads](https://github.com/massivethreads/massivethreads)
Static Task Elimination

- If a compiler identifies $H$ and calculates an $H$th termination condition, the simplest cut-off is applicable.

```c
void vecadd(float* a, float* b, int n){
  if(n == 1){
    *a += *b;
  }else{
    spawn vecadd(a, b, n/2);
    spawn vecadd(a+n/2, b+n/2, n-n/2);
  }
}
```

```c
void vecadd_seq(float* a, float* b, int n){
  if(n == 1){
    *a += *b;
  }else{ /*spawn*/
    vecadd_seq(a, b, n/2);
    vecadd_seq(a+n/2, b+n/2, n-n/2);
    sync; /*sync;*/
  }
}
```

Just remove spawn and sync in the $H$th termination condition.
General Inlining

- General inlining incurs code bloat.
  - Divide-and-conquer tasks often have more than one recursive calls.

```c
void vecadd(float* a, float* b, int n){
  if(Hth Termination Condition){
    vecadd_seq(a, b ,n);
  }else{
    spawn vecadd(a, b, n/2);
    spawn vecadd(a+n/2, b+n/2, n-n/2);
    sync;
  }
}
void vecadd_seq(float* a, float* b, int n){
  if(n == 1){
    *a += *b;
  }else{
    /*spawn*/vecadd_seq(a, b, n/2);
    /*spawn*/vecadd_seq(a+n/2, b+n/2, n-n/2);
    /*sync;*/
  }
}
```

Inlining vecadd_seq() almost doubles the code size.
void vecadd_seq(float* a, float* b, int n){
    if(n == 1){
        *a += *b;
    }else{
        for(int i = 0; i < 2; i++){
            float *a2, *b2; int n2;
            switch(i){
                case 0:
                    a2=a; b2=b ; n2=n/2; break;
                case 1:
                    a2=a+n/2; b2=b+n/2; n2=n-n/2; break;
            }
            vecadd_seq(a2,b2,n2);
        }
    }
}
Code-bloat-free Inlining (2/2)

1. Delay execution of spawned tasks to corresponding sync.

2. In the $H$th termination condition, inlining $H$ times can remove the innermost recursive calls.

```c
void vecadd_seq(float* a, float* b, int n){
    if(n == 1){
        *a += *b;
    }else{
        for(int i = 0; i < 2; i++){
            float *a2, *b2; int n2;
            switch(i){
                case 0:
                    a2=a;     b2=b    ; n2=n/2;    break;
                case 1:
                    a2=a+n/2; b2=b+n/2; n2=n-n/2; break;
            }
            //Inline vecadd_seq(a2,b2,n2)
            if(n2 == 1){
                *a2 += *b2;
            }else{
                //Never executed in the 1st condition.
                /* for(int i2 = 0; i2 < 2; i2++){
                    float *a3, *b3; int n3;
                    [...];
                    vecadd_seq(a3,b3,n3);
                } */
            }
        }
    }
}
```

These recursive calls are never called in the 1st termination condition.
Loopification: Goal

- Try to convert recursion into a loop.

```c
void vecadd(float* a, float* b, int n){
    if(n == 1){
        *a += *b;
    }else{
        spawn vecadd(a, b, n/2);
        spawn vecadd(a+n/2, b+n/2, n-n/2);
        sync;
    }
}
```

Desired final result.

```c
void vecadd(float* a, float* b, int n){
    if(Hth Termination Condition){
        vecadd_seq(a, b ,n);
    }else{
        spawn vecadd(a, b, n/2);
        spawn vecadd(a+n/2, b+n/2, n-n/2);
        sync;
    }
}
void vecadd_loop(float* a, float* b, int n){
    for(int i=0; i<n; i++)
        a[i] += b[i];
}
```
Loopification: Idea (1/2)

- The target task needs to have a recursion block in non-termination condition.
  - A recursion block is required to have no side-effects but creating tasks.

```c
void vecadd(float* a, float* b, int n){
    if (n == 1){
        *a += *b;
    } else {
        spawn vecadd(a, b, n/2);
        spawn vecadd(a+n/2, b+n/2, n-n/2);
        sync;
    }
}
```

- **: leaf function
- **: recursion block

Blocks executed in a termination condition.
Loopification: Idea (2/2)

1. Generate loop candidates by assigning a certain termination condition and estimating the loop form.
   - The loop element is assumed to be a leaf function.

```c
void vecadd_candidate1(float* a, float* b, int n){
    for(int i=0; i<n; i++){
        leaf_function(a + i, b + i, /**/);
    }
}
```

2. Then, check the equivalence of a loop candidate and recursion (induction)

   This verification is valid only in a *th termination condition.

Please check our paper for details.
Why Loopification?

Why don't you use loop-parallelism in the first place?

→ We believe there are two merits:

- A divide-and-conquer strategy can be written as cache-oblivious style, suitable for modern hierarchical memory.
  - e.g., matrix multiplication, and stencil computation
- Our loopification also vectorizes a loop utilizing dependency information revealed by task keywords.
  
```c
void heat2d(array2d a, array2d b) {
    [...];
    if (sizex(a)==1 && sizey(b)==1) {
        ax = a[i-1,j]-2*a[i,j]+a[i+1,j];
        ay = a[i,j-1]-2*a[i,j]+a[i,j+1];
        b[i,j] = a[i,j]+K*(ax+ay);
    } else {
        spawn heat2d(div11(a), div11(b));
        spawn heat2d(div12(a), div12(b));
        spawn heat2d(div21(a), div21(b));
        spawn heat2d(div22(a), div22(b));
        sync;
    }
}
```

2D divide-and-conquer achieves better cache locality.
If Analysis Fails → Dynamic Cut-off

• Termination condition analysis sometimes fails for various reasons.
  - e.g., Pointer-based tree traversal.
  
  It's difficult to identify the simple “Hth termination condition”

• In that case, our system applies the dynamic cut-off as a fallback strategy.
  
  - We adopted the state-of-the-art dynamic cut-off proposed by Thoman et al. [*]

---

void treetraverse(TREE* tree){
  if(tree->left==0&&tree->right==0){
    calc(tree);
  }else{
    if(tree->left)
      spawn(treetraverse(tree->left));
    if(tree->right)
      spawn(treetraverse(tree->right));
    sync;
  }
}

Summary of Static Cut-off

- Our developed system...
  - Calculates an $H$th termination condition.
  - Decides a height $H$ using heuristics.
  - Applies one of the compiler optimizations:
    - Static task elimination
    - Code-bloat-free inlining
    - Loopification
  - Adopts dynamic cut-off if analysis (1) fails.

1. Try to identify the $H$th termination condition.
2. Try to apply loopification.
3a. Static task elimination
3b. Code-bloat-free inlining
3c. Loopification
4. Dynamic cut-off

Failed

Succeeded
Index

0. Short Summary
1. Introduction
2. Proposal: Static Cut-off
3. Evaluation
   - Benchmarks & Environment
   - Performance Evaluation
4. Conclusion
Implementation & Environment

• We implemented it as an optimization pass on LLVM 3.6.0.

  Modified MassiveThreads[*1], a lightweight work-stealing based task parallel system adopting the child-first scheduling policy[*2].

• Experiments were done on dual sockets of Intel Xeon E5-2699 v3 (Haswell) processors (36 cores in total).
  - Use `numactl --interleave=all` to balance physical memory across sockets

[*2] Mohr et al., Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs, LFP '90, 1990
Benchmarks

- 15 benchmarks were prepared for evaluation.
  - All are divide-until-trivial task parallel programs.

- `fib`
- `nqueens`
- `fft`
- `sort`
- `nbody`
- `strassen`
- `vecadd`
- `heat2d`
- `heat3d`
- `gaussian`
- `matmul`
- `trimul`
- `treeadd`
- `treesum`
- `uts`

<table>
<thead>
<tr>
<th>Function</th>
<th>Dynamic Cut-off</th>
<th>Termination Condition</th>
<th>Code-bloat-free Inlining</th>
<th>Loopification</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fib</code></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td><code>nqueens</code></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td><code>fft</code></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td><code>sort</code></td>
<td>✓</td>
<td>✓</td>
<td>(1/2)</td>
<td>(1/2)</td>
</tr>
<tr>
<td><code>nbody</code></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td><code>strassen</code></td>
<td>✓</td>
<td>✓</td>
<td>(4/5)</td>
<td>(4/5)</td>
</tr>
<tr>
<td><code>vecadd</code></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td><code>heat2d</code></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td><code>heat3d</code></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td><code>gaussian</code></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td><code>matmul</code></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td><code>trimul</code></td>
<td>✓</td>
<td>✓</td>
<td>(1/4)</td>
<td>(1/4)</td>
</tr>
<tr>
<td><code>treeadd</code></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td><code>treesum</code></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td><code>uts</code></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

Only dynamic cut-off is applicable to them.
How to Read?

- Y-Axis: **Relative performance** over base (divide-until-trivial)

- **dynamic**: dynamic cut-off proposed by Thomans et al.
- **static**: all - loopification - code-bloat-free inlining
- **cbf**: all - loopification
- **loop**: all
- **proposed**: the total performance
- **seq**: sequential (not task-parallelized)

Only show the results if static / cbf / loop is applicable.
How to Read?

- Y-Axis: Relative performance over base (divide-until-trivial)
  - dynamic: dynamic cut-off proposed by Thomans et al.
  - static: all - loopification - code-bloat-free inlining
  - cbf: all - loopification
  - loop: all
  - proposed: the total performance
  - seq: sequential (not task-parallelized)

Only show the results if static / cbf / loop is applicable.
Roughly speaking, How to Read?

- **Y-Axis**: Relative performance over **base** (divide-until-trivial)
  - **dynamic**: dynamic cut-off proposed by Thomans et al.
  - **static**: static task elimination if applicable
  - **cbf**: code-bloat-free inlining if applicable
  - **loop**: loopification if applicable
  - **proposed**: our proposal (the right chart ➔)
  - **seq**: not task-parallelized
Single-threaded Performance (1/3)

- Cut-off (■ & ■) improved performance overall.
- Compared to ■ dynamic cut-off, ■ our proposed cut-off achieved higher performance.
Performance of ■ static was better than ■ dynamic if termination condition analysis succeeded.

- Evaluation of a cut-off condition inserted at compile time is less expensive than that of dynamic cut-off.

- ■ static achieved comparable performance of ■ seq.

Static task elimination successfully reduced tasking overheads in most cases.
Single-threaded Performance (3/3)

- Performance was furthermore improved if cbf / loop was applicable.
- As a result, our proposal achieved 11.2x speedup (from 1.1x to 333x) on average over original task parallel programs.
Multi-threaded Performance

- Multi-threaded performance (36 cores) is similar to single-threaded one.

- Our proposal achieved 8.0x speedup (from 1.1x to 220x) on average over original task parallel programs.
vs. Loop Parallel Programs

- Compared to loop parallel programs.
  - task: task parallel programs optimized by our proposal.
  - omp: programs just inserted `#omp parallel for`.
  - omp_optimized: OpenMP ones hand-tuned carefully. Tuning attributes (collapse, chunk size, scheduling etc) and loop blocking.
  - polly: programs automatically parallelized by Polly [*].

[*] Grosser et al., Polly - Polyhedral optimization in LLVM., IMPACT ’11, 2011.
vs. Loop Parallel Programs

- Performance of □ task was comparable to that of ■ omp and □ polly.
- ■ Optimized OpenMP version was fastest, however.
  - One reason is that the recursive cache blocking is not so flexible as to fit the exact cache size.
Index

0. Short Summary
1. Introduction
2. Proposal: Static Cut-off
3. Evaluation
4. Conclusion
Conclusion

- We propose a compiler optimizing divide-until-trivial task parallel programs using the $H$th termination condition analysis.
  - Further optimizations are developed based on the analysis.

- The evaluation shows the efficacy of the proposed automatic cut-off.

Future work:
- Widen the applicable range of loopification.
- Adopt better heuristics (or totally new methods) to determine a height $H$.  