Autotuning of a Cut-off for Task Parallel Programs

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

September 22, 2016
@ ATMG '16 (special session in MCSoC '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.

- We had developed a series of compiler **optimization techniques** for **automatic cut-off** ("**static cut-off**"[*])

---

[*] Iwasaki et al., A Static Cut-off for Task Parallel Programs, PACT '16, 2016.
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.
- We had developed a series of compiler optimization techniques for automatic cut-off ("static cut-off"[*])
- This study proposes an automatic cut-off technique with an autotuning method to obtain the best combination of these techniques for higher performance.

[*] Iwasaki et al., A Static Cut-off for Task Parallel Programs, PACT '16, 2016.
Index

0. Short Summary
1. Introduction
2. Static Cut-off and its Limitations
3. Our Proposal: Cut-off with Autotuning
4. Evaluation
5. Conclusion
Index

0. Short Summary

1. Introduction
   - What is task parallelism?
   - What is a “cut-off”?

2. Static Cut-off and its Limitations

3. Our Proposal: Cut-off with Autotuning

4. Evaluation

5. Conclusion
Importance of Multi-threading

• The number of CPU cores gets larger and larger.

Intel Xeon Phi (Knights Corner) is a typical example: it has 60 cores, supporting over 200 hardware threads. [http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html](http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html)

• Multi-threading is essential to exploiting modern processors.

→ A task parallel model is one of the most promising parallel programming models.
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){  
  if(n == 1){  
    *a += *b;  
  }else{  
    spawn vecadd(a, b, n/2);  
    spawn vecadd(a+n/2, b+n/2, n-n/2);  
    sync;  
  }  
}

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

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 $1 \leq n \&\& n \leq 1000$

• Programmers commonly apply it manually.

```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;
  }
}
```

//Sequential version of vecadd

```c
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*/;
  }
}
```
In addition to reducing tasking overheads, further transformations are applicable to serialized tasks in some cases.

vecadd_seq() is loopified.
Dynamic Cut-off

- Most previous studies on automatic cut-off [*1,*2,*3] focused 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...

- Problems:
  Cost to evaluate a cut-off condition is large, Optimizations after the cut-off are less applicable.

Our Goal: Automatic Cut-off

- Our goal is developing automatic cut-off including further optimizations automatically 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 Goal: Automatic Cut-off

- Our goal is developing automatic cut-off including further optimizations automatically 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.

```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];
}
```
Index

0. Short Summary
1. Introduction
2. Static Cut-off and its Limitations
   - Our previous work: static cut-off
   - Limitations
3. Our Proposal: Cut-off with Autotuning
4. Evaluation
5. Conclusion
What we've proposed: Static Cut-off

- **Static cut-off** is an automatic cut-off method including a series of compile-time optimization techniques for task parallel programs.

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

[*] Iwasaki et al., A Static Cut-off for Task Parallel Programs, PACT '16, 2016.
What we've proposed: Static Cut-off

- **Static cut-off** is an automatic cut-off method including a series of compile-time optimization techniques for task parallel programs.

- It tries to aggregate **tasks near leaves**.

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

**Key idea:** use a **height** instead of a **depth**.
Depth/Height of Tasks

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

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

  ```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;
  }
  }
  ```
Depth/Height of Tasks

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

  \[
  F_n = \begin{cases} 
  n & \text{if } n < 2 \\
  F_{n-1} + F_{n-2} & \text{otherwise}
  \end{cases}
  \]

  - Depth is easy to obtain.

  - e.g., increment a variable from the root.

  **Cut-off in “depth > 6”**

```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;
    }
}
```
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 calculate, 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 "depth > 6"

Cut-off in "height < 3"
Static Cut-off Flow

1. Try to calculate a height-based cut-off condition.
   - If the height-based cut-off condition is calculable ...
     2. Decide a height parameter H.
     3. Apply one of the following:
        • Static task elimination
        • Code-bloat-free inlining
        • Loopification
   - Otherwise...
     2. Apply the dynamic cut-off [*]

Examples of Static Cut-off

2. Decide a height parameter $H$.

3. Apply one of the following:
   - Static task elimination
   - Code-bloat-free inlining
   - Loopification

Use heuristics.

$H = 10$ in this case.

```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;
    }
}

void vecadd_seq(float* a, float* b, int n){
    if(1 <= n && n <= 1024){
        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){
    ???
}
```
Examples of Static Cut-off

2. Decide a height parameter H.

3. Apply one of the following:
   - Static task elimination
   - Code-bloat-free inlining
   - Loopification

Just remove spawn & sync to reduce the overheads.
Examples of Static Cut-off

2. Decide a height parameter H.

3. Apply one of the following:
   - Static task elimination
   - Code-bloat-free inlining
   - Loopification

```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;
  }
}
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 10 times here.
      vecadd_seq(a2,b2,n2);
    }
  }
}
```

Apply inlining to reduce function call overheads w/o exponential code growth.
Examples of Static Cut-off

2. Decide a height parameter \( H \).

3. Apply one of the following:
   - Static task elimination
   - Code-bloat-free inlining
   - Loopification

```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;
    }
}

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

Simplify the control flow and also promote vectorization.
Summary of Static Cut-off

First, try to calculate a height-based cut-off condition.

– If it is calculable, determine $H$ and apply one of them:
  
  ● Static task elimination: reduce tasking overheads.
  
  ● Code-bloat-free inlining: + reduce function call overheads.
  
  ● Loopification: + convert recursion into a loop.

– Otherwise, apply the dynamic cut-off [*]

Limitations of Static Cut-off

- The evaluation had shown our static cut-off enhanced performance, yet there are room for further tuning to achieve best performance.
  
  1. Heuristics-based decision on cut-off threshold does not always return the optimal ones.
  
  2. Optimization for serialized tasks can be improved more.
     - e.g., combining multiple transformations
  
  3. Dynamic cut-off is not so efficient.
     - However, our static cut-off cannot be applied to all.
Index

0. Short Summary
1. Introduction
2. Static Cut-off and its Limitations
3. Our Proposal: Cut-off with Autotuning
   - Autotuning framework
4. Evaluation
5. Conclusion
Cut-off with Autotuning

- Decide a cut-off strategy using an autotuning way.
- There are three possible elements for tuning:
  1. Cut-off thresholds (\(\equiv\) a cut-off condition)
     - Especially for loopification, the cut-off condition has an impact on cache-blocking effect.
  2. Combination of transformations.
     - e.g., inlining & parallel + loopification & serial
  3. Whether depth or height is used.
**Autotuning Flow**

- **Input:** original code + script to compile & run
- **Output:** autotuned configuration file
  - Our compiler generates an autotuned program with that file.
- **We adopt an autotuning strategy similar to that of PetaBricks[*.**

[*. J. Ansel et al., PetaBricks: A Language and Compiler for Algorithmic Choice. PLDI '09, 2009.}
Related Work

PetaBricks

- **PetaBricks** [*], proposed by Ansel et al., is an autotuning framework for parallel divide-and-conquer algorithms.
  - It focuses on algorithmic choice.
  - e.g., for sorting, we can combine mergesort, quicksort, insertionsort together, by switching at each “conquer” phase.

- Users need to write multiple versions of the algorithm.

Basic Idea: Connecting Tasks

- Similar to the approach of PetaBricks, we optimize cut-off by connecting various tasks with appropriate conditions.

- The simplest cut-off is represented as follows:

Original task → Transformed task1 → Transformed task2 → Transformed task3

Condition1 → Condition2 → Condition3

: switch

Original task → Serialized function (Static task elimination)

Cut-off Condition
Example: Fibonacci

First, run the original task to **ensure parallelism**, then switch to the serialized to **reduce a overhead**. The leaf is inline-expanded for **serial performance**.

```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;
    }
}
```
Example: Fibonacci

First, run the original task to ensure parallelism, then switch to the serialized to reduce overhead. Leaf tasks are inline-expanded for serial performance.

```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;
    }
}
```

Run in parallel (11 < n)
(an original task parallel function)

Run in serial (3 < n < 12)
(a normal recursive function)

Run in serial (n < 4)
(a recursive function inline-expanded twice)
Example: Final Code

Original task

if n < 11

Serialized function

if n < 4

Twice-inlined serialized function

```c
void fib(int n, int* r){
    if(n < 11){
        fib2(n, r);
    }else{
        int a, b;
        spawn fib(n-1, &a);
        spawn fib(n-2, &b);
        sync;
        *r = a + b;
    }
}

void fib2(int n, int* r){
    if(n < 4){
        fib3(n, r);
    }else{
        int a, b;
        fib2(n-1, &a);
        fib2(n-2, &b);
        *r = a + b;
    }
}

void fib3(int n, int* r){
    if(n < 2){
        *r = n;
    }else
        [inlined twice];
}
```
Search Space for Autotuning

- There are two tuning parameters:
  1. **Switching conditions**
  2. **Optimizations** for each task (task)
     - + Optimization parameters (e.g., # of times of inlining)

- The number of patterns are potentially **countless**.
Basic Cut-off Strategy

1. Use **height** rather than depth if possible.

2. # of task versions is at most 3.

   - An original task: no optimization is applied
     → fine-grained & parallel
   - A middle task: optimization may be applied
     → fine~coarse-grained & serial
   - A leaf task: optimization may be applied
     → coarse-grained & serial

To limit the search space.
Three Typical Patterns

- We defined three typical patterns to limit the search space.
  - Pattern 1: depth-based cut-off
    - Target examples: tree traversals
  - Pattern 2: height-based cut-off without loopification
    - Target examples: fibonacci, nqueens
  - Pattern 3: height-based cut-off with loopification
    - Target examples: vector addition, matrix multiplication
Three Typical Patterns

- We defined three typical patterns to limit the search space.
  - Pattern 1: depth-based cut-off
    - Target examples: tree traversals
  - Pattern 2: height-based cut-off without loopification
    - Target examples: fibonacci, nqueens
  - Pattern 3: height-based cut-off with loopification
    - Target examples: vector addition, matrix multiplication
Three Typical Patterns

- We defined three typical patterns to limit the search space.
  - Pattern 1: depth-based cut-off
    - Target examples: tree traversals
  - Pattern 2: height-based cut-off without loopification
    - Target examples: fibonacci, nqueens
  - Pattern 3: height-based cut-off with loopification
    - Target examples: vector addition, matrix multiplication
Three Typical Patterns

- We defined three typical patterns to limit the search space.
  - Pattern 1: depth-based cut-off
    - Target examples: tree traversals
  - Pattern 2: height-based cut-off without loopification
    - Target examples: fibonacci, nqueens
  - Pattern 3: height-based cut-off with loopification
    - Target examples: vector addition, matrix multiplication
Three Typical Patterns

- We defined three typical patterns to limit the search space.
  - Pattern 1: depth-based cut-off
    - Target examples: tree traversals
  - Pattern 2: height-based cut-off without loopification
    - Target examples: fibonacci, nqueens
  - Pattern 3: height-based cut-off with loopification
    - Target examples: vector addition, matrix multiplication
Pattern 1: Depth-based Cut-off

- It is designed for tasks to which it is difficult to apply static cut-off.
  - e.g., tree traversal programs, unbalanced tree search
Pattern 2: Height-based Cut-off

It is designed for **tasks to which static cut-off is applicable**, but loopification is not.

- e.g., fib, nqueens

It preserves recursive call sites.

They completely remove recursion by inlining H times.
Pattern 3: Height-based Cut-off

- It is designed for loopifiable tasks.
  - e.g., vecadd, matmul, heat2d

It significantly affects loop-blocking.
Avoid Loss of Parallelism

- More parallelism is better if the performance is the same in terms of dynamic load balancing.
- Our autotuning adapt the switching condition preserving most parallelism, which can accomplish 99% of the optimal performance measured.

- In this example, we choose \( n < 2000 \) even if \( n < 10000 \) performs slightly better.

```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;
    }
}
```

- Not fastest, though.
Autotuning: Summary

- Our autotuning searches for the best combination of differently transformed tasks.
  - It contains a cut-off concept.
- It employs three patterns to limit the search space.
  - Depth-based one
  - Height-based ones (w/ & w/o loopification)
Index

0. Short Summary
1. Introduction
2. Static Cut-off and its Limitations
3. Our Proposal: Cut-off with Autotuning
4. Evaluation
   – Benchmarks & Environment
   – Performance Evaluation
5. 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].

- An autotuning driver is written in Python.

- 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

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

<table>
<thead>
<tr>
<th></th>
<th>Dynamic Cut-off</th>
<th>Autotuning Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>fib</td>
<td>✓</td>
<td>2. Height-based without loopification</td>
</tr>
<tr>
<td>nqueens</td>
<td>✓</td>
<td>2. Height-based without loopification</td>
</tr>
<tr>
<td>nbody</td>
<td>✓</td>
<td>2. Height-based without loopification</td>
</tr>
<tr>
<td>vecadd</td>
<td>✓</td>
<td>3. Height-based with loopification</td>
</tr>
<tr>
<td>heat2d</td>
<td>✓</td>
<td>3. Height-based with loopification</td>
</tr>
<tr>
<td>heat3d</td>
<td>✓</td>
<td>3. Height-based with loopification</td>
</tr>
<tr>
<td>gaussian</td>
<td>✓</td>
<td>3. Height-based with loopification</td>
</tr>
<tr>
<td>matmul</td>
<td>✓</td>
<td>3. Height-based with loopification</td>
</tr>
<tr>
<td>treeadd</td>
<td>✓</td>
<td>1. Depth-based</td>
</tr>
<tr>
<td>treesum</td>
<td>✓</td>
<td>1. Depth-based</td>
</tr>
<tr>
<td>uts</td>
<td>✓</td>
<td>1. Depth-based</td>
</tr>
</tbody>
</table>

Static cut-off is not applicable to them.
Multi-threaded Performance

- Optimization including *dynamic* (dynamic cut-off[\*]) improved performance over *original* (no cut-off)
- *autotuning* (proposal) was faster than *dynamic* and *static* (static cut-off) overall.

vs. Loop Parallel Programs

- **autotuning** (proposed autotuned one) was
  - comparable to *polly* (Polly) and *omp* (OpenMP)
  - defeated by *omp_optimized* (hand-optimized OpenMP).

  • Hand-tuned OpenMP can employ flexible cache-blocking.
  • div-and-conq divides the axis only by a constant integer.
Index

0. Short Summary
1. Introduction
2. Static Cut-off and its Limitations
3. Our Proposal: Cut-off with Autotuning
4. Evaluation
5. Conclusion
We developed an autotuning framework for divide-until-trivial task parallel programs. It achieved significant speedup over the original naïve task parallel programs.