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.

Keyword: divide-and-conquer

- "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.)

Short Summary

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

Keyword: divide-and-conquer

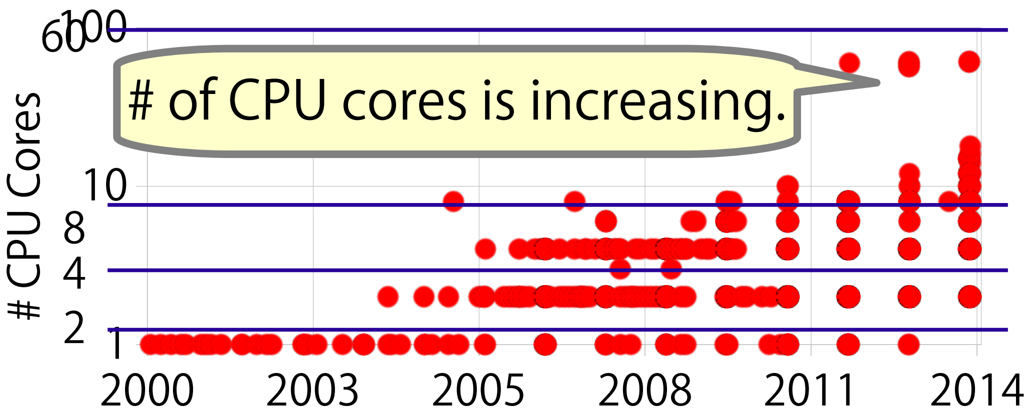
- "Cut-off" is an optimization technique for task parallel programs to control granularity.
- Previous cut-off systems have been dynamic, and have issues and limitations (detailed later.)
- We propose a static cut-off method and further compiler optimization techniques based on it.
- Evaluation shows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

Index

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.

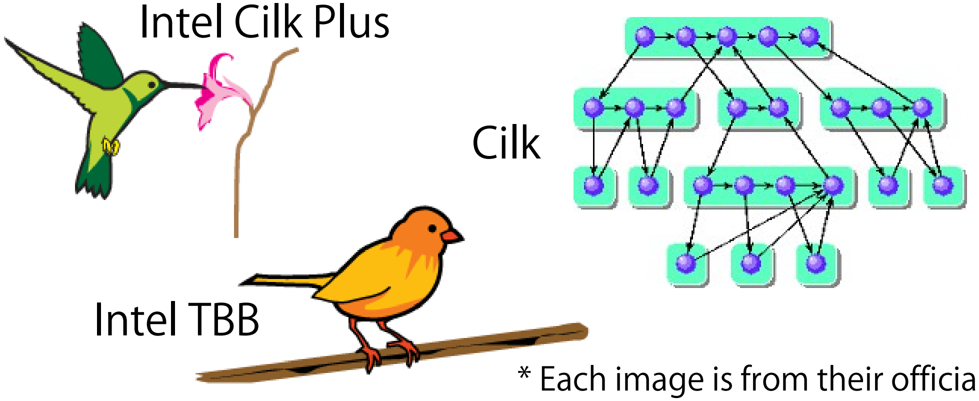


# of CPU cores is increasing.

From CPU DB (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 …



* Each image is from their official pages.

- 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.

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

Same meaning.

```
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;
  }
}
```

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

```
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.

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

```
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.

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

```
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;
  }
}
```

- Cut-off has been known as an effective optimization technique.

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.

```
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;
  }
}
```

Cut-off

```
void vecadd(float* a, float* b, int n){
  if(1<= n && n <=1000){
    vecadd_seq(a, b, n
  }else{
    spawn vecadd(a, b, n/2);
    spawn vecadd(a+n/2, b+n/2, n-n/2);
    sync;
  }
}
//Sequential version of vecadd
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;*/
  }
}
```

A cut-off condition

Call a sequential vecadd
if `1 <= n && n <= 1000`

- Programmers commonly apply it manually.

Cut-off + Further Optimizations

```
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;
  }
}
```

1. Cut-off

2. Transformation

```
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){
  for(int i = 0; i < n; i++)
    a[i] += b[i];
}
```

vecadd_seq() is loopified.

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

Cut-off + Further Optimizations

- 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

… by just writing naïve task parallel programs.

1. Cut-off

```
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){
  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){
  for(int i = 0; i < n; i++)
    a[i] += b[i];
}
```

2. Transformation

vecadd_seq() is loopified.

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

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.

```
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.

Let's say divide-until-trivial task parallel programs.

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

→ Develop optimizations for divide-until-trivial tasks.

```
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

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...

[*1] Bi et al. An Adaptive Task Granularity Based Scheduling for Task-centric Parallelism, HPCC '14, 2014

[*2] Duran et al. An Adaptive Cut-offfor Task Parallelism, SC '08, 2008

[*3] Thoman et al. Adaptive Granularity Control in Task Parallel Programs Using Multiversioning, Euro-Par'13, 2013

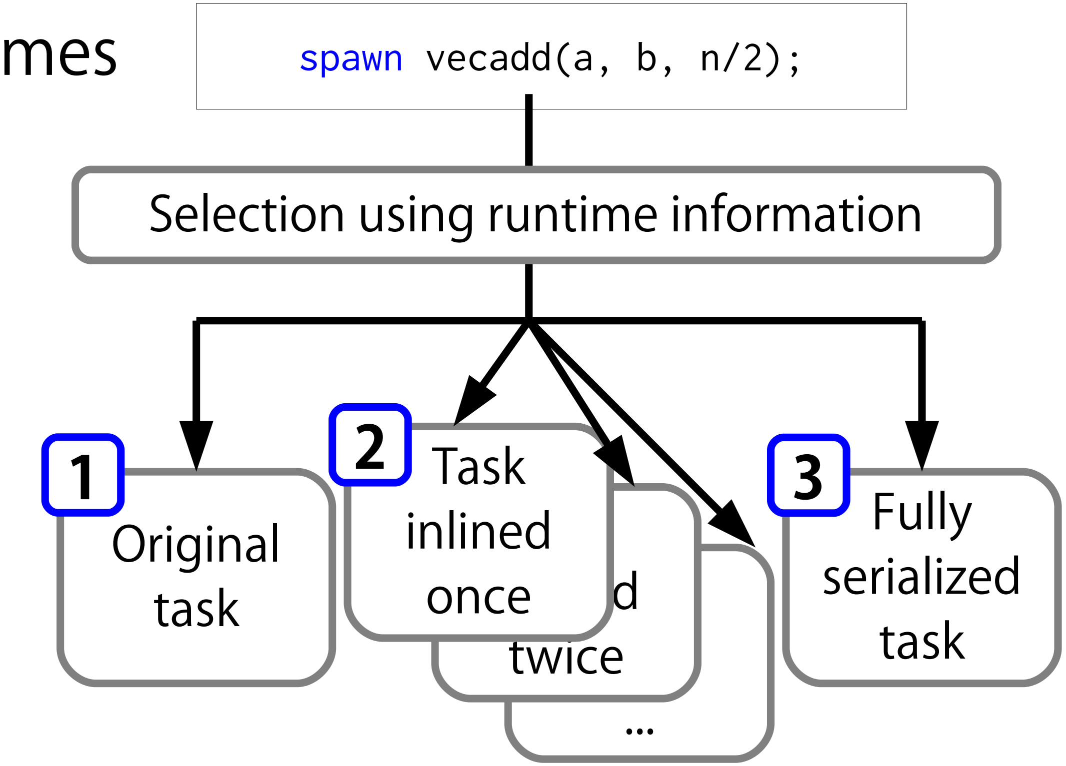
Related Work

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.



```
spawn vecadd(a, b, n/2);
```

[*] P. Thoman et al. Adaptive granularity control in task parallel programs using multiversioning. Euro-Par '13, 2013

Related Work

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.

Dynamic cut-off advantage: wider applicable range.

[*1] Bi et al. An Adaptive Task Granularity Based Scheduling for Task-centric Parallelism, HPCC '14, 2014

[*2] Duran et al. An Adaptive Cut-offfor Task Parallelism, SC '08, 2008

[*3] Thoman et al. Adaptive Granularity Control in Task Parallel Programs Using Multiversioning, Euro-Par'13, 2013

Key Idea: Cut-off Near Leaves

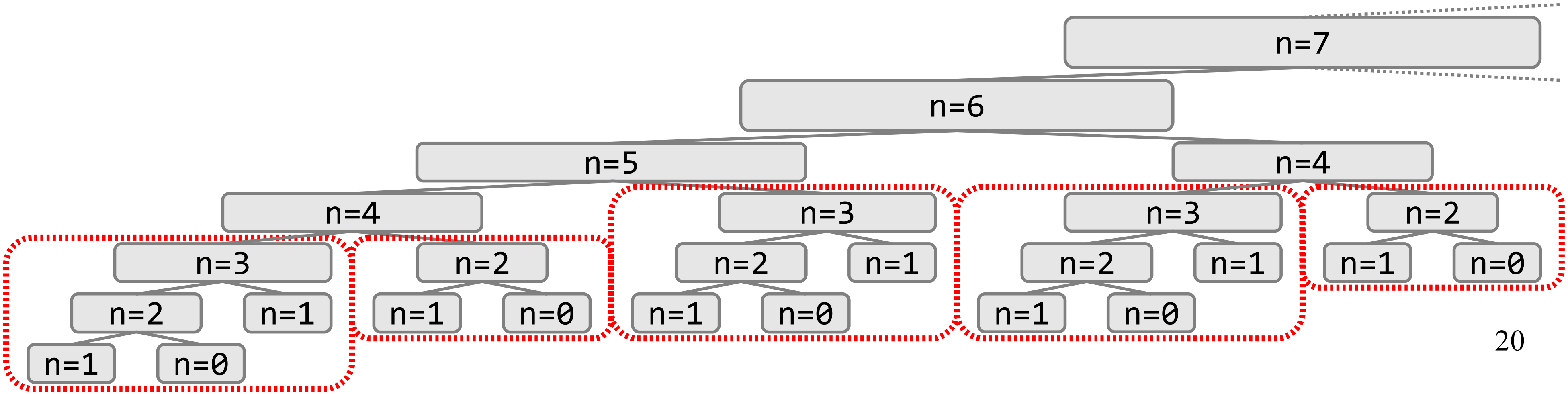
- Aggregate tasks near leaves.

Encircled by ⬚

+ 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.



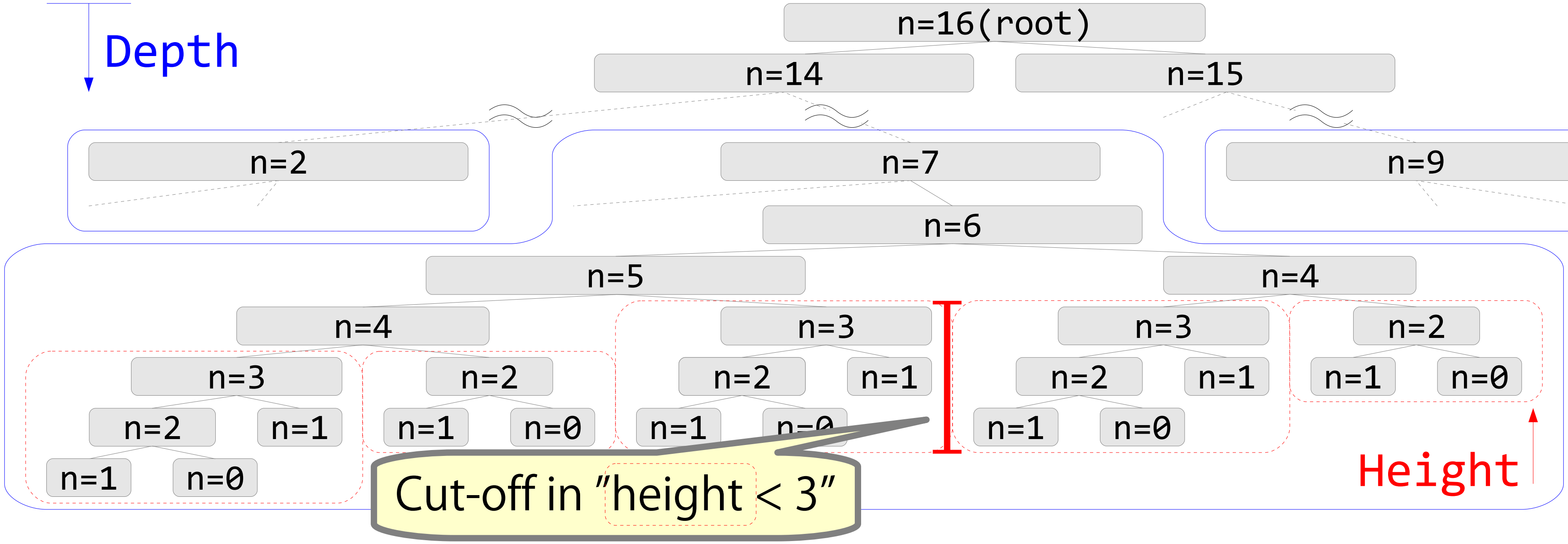
Height of Task

- 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.

```
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;
  }
}
```



Transformation Flow

```
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;
  }
}
```

Input

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

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

Transformation Flow

```
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;
  }
}
```

Input

```
void vecadd(float* a, float* b, int n){
  if(1 <= n && n <= 10000){
    vecadd_opt(a, b, n);
  }else{
    spawn vecadd(a, b, n/2);
    spawn vecadd(a+n/2, b+n/2, n-n/2);
    sync;
  }
}
void vecadd_opt(float* a, float* b, int n){
  for(int i = 0; i < n; i++)
    a[i] += b[i];
}
```

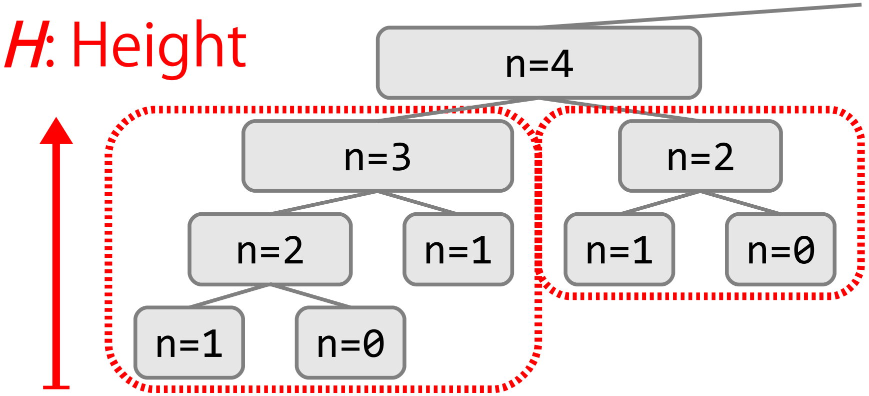
1. Try to obtain a cut-off condition.
2. Optimize a task after cut-off.

How to Implement it?

1. Try to obtain a cut-off condition.

Key idea.

→ Try to calculate "the *H*th termination condition"
the condition in which a task ends within a height ***H***.



For example, the 2nd termination condition of fib is "n <= 3"

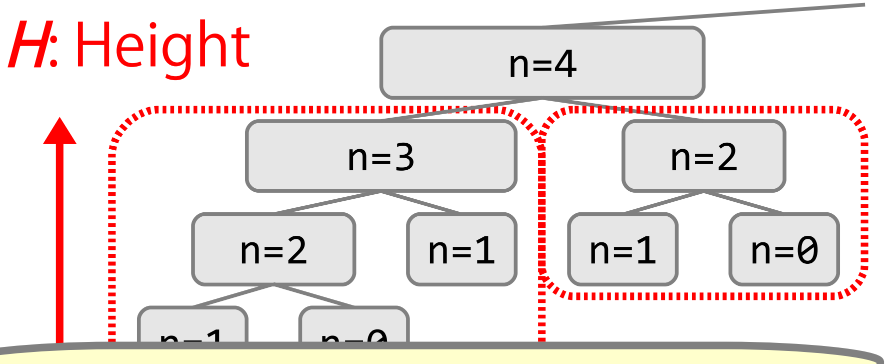
How to Implement it?

1. Try to obtain a cut-off condition.

Key idea.

→ 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

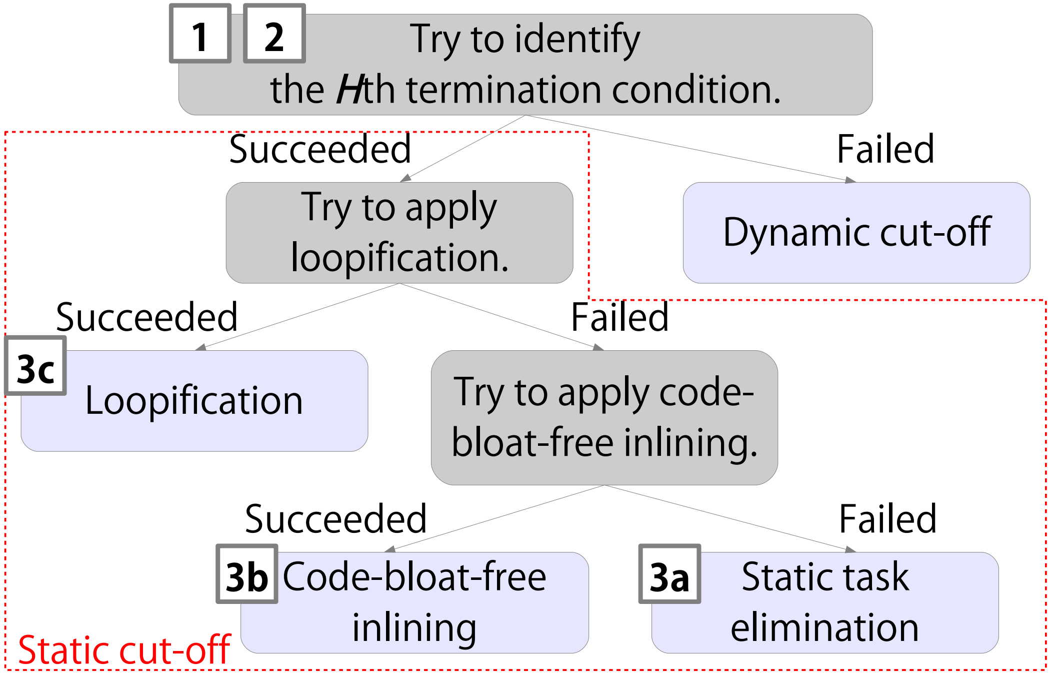
Reduce tasking overheads.

+ 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



1 **2** Try to identify the *H*th termination condition.

Succeeded → Try to apply loopification.

Failed → Dynamic cut-off

Try to apply loopification: Succeeded → **3c** Loopification

Try to apply loopification: Failed → Try to apply code-bloat-free inlining.

Try to apply code-bloat-free inlining: Succeeded → **3b** Code-bloat-free inlining

Try to apply code-bloat-free inlining: Failed → **3a** Static task elimination

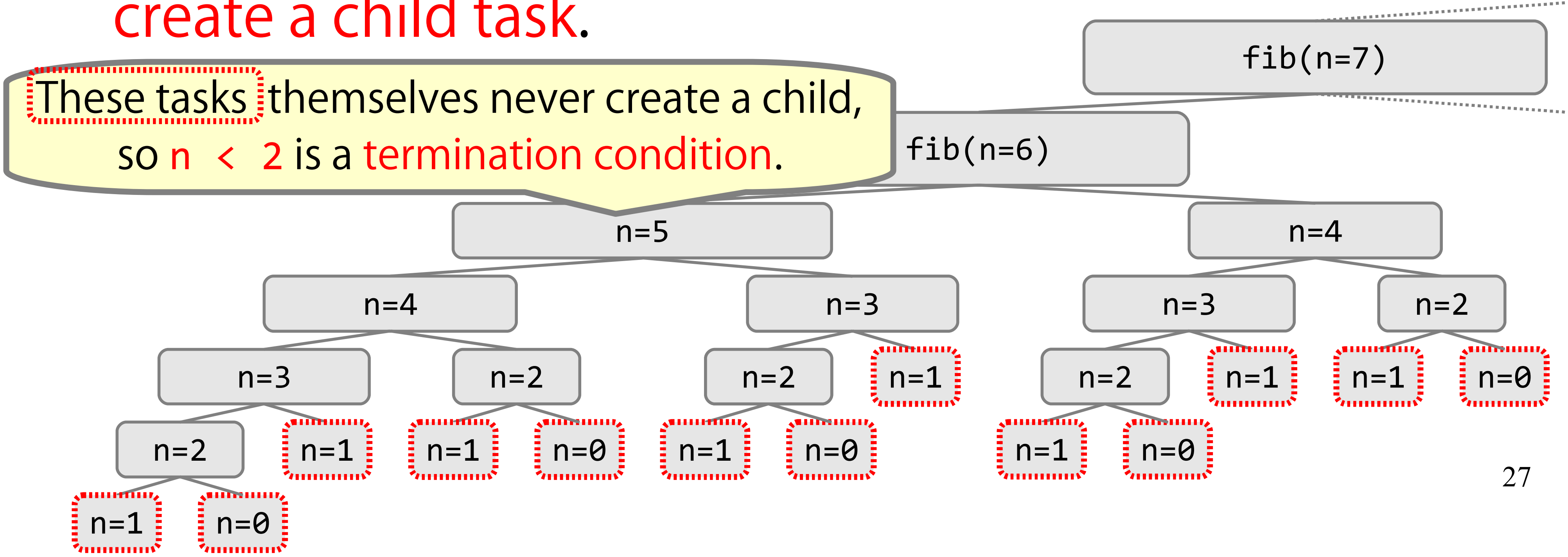
Static cut-off

1

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.

```
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;
  }
}
```



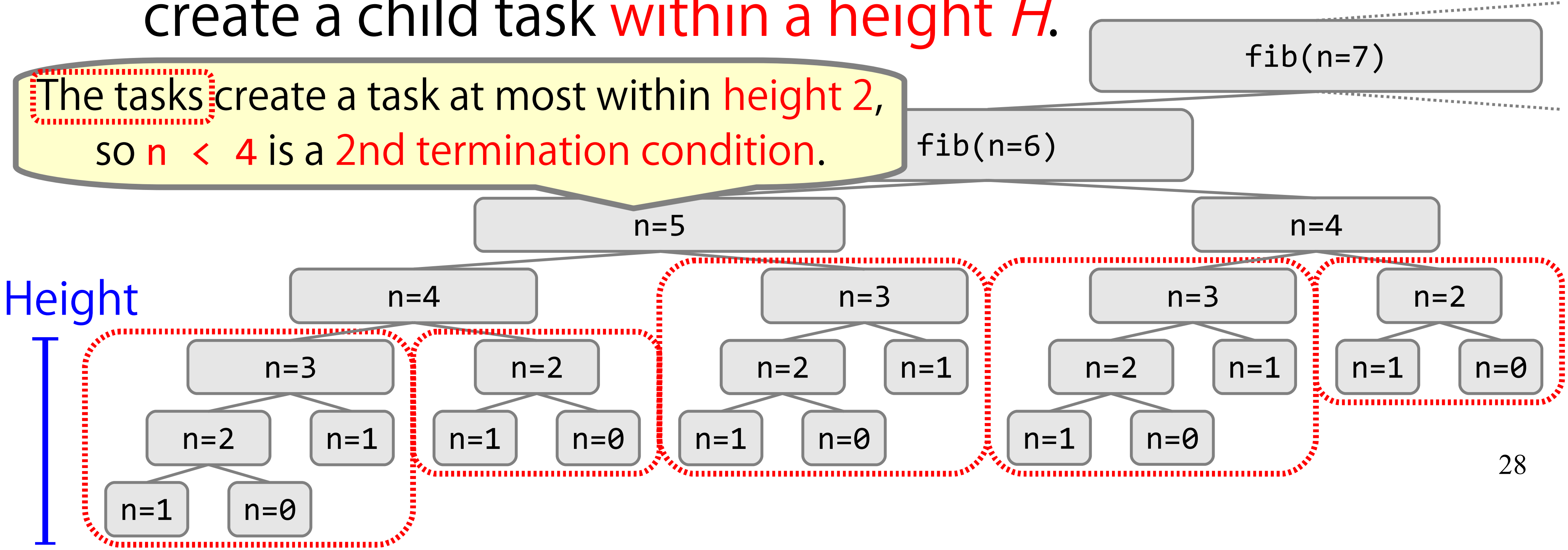
These tasks themselves never create a child, so `n < 2` is a termination condition.

1

*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*.

```
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;
  }
}
```



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

1

Termination Condition Analysis

- A *0*th termination condition is a condition in which tasks never create children.
  - A simple basic block analysis tells $n < 2$ is such a condition for ***fib*** example.
- An *H*th termination condition is recursively calculated by using an (*H*-*1*)th termination condition.
  - It requires a simple algebra solver.

```
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;
  }
}
```

2

Determining Cut-off Height *H*

- Basically, choose the larger *H*.

It is designed for very fine-grained tasks.

a. a height which makes the number of cycles after cut-off is less than 5000 cycles.

  - Task creation takes approximately 100 cycles.

Binary Task Creation (Height = 27) on MassiveThreads[*] with one core.

CPU	Frequency	Task Creation Time
Intel Xeon E7540	2.0GHz	36.0 [ns/task]
AMD Opteron 6380	2.5GHz	44.9 [ns/task]
Intel Xeon E5-2695 v2	2.4GHz	21.5 [ns/task]
Intel Xeon E5-2699 v3	2.3GHz	33.8 [ns/task]

  - 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 a minimum cut-off height.

THE UNIVERSITY OF TOKYO

[*] MassiveThreads https://github.com/massivethreads/massivethreads

3a

Static Task Elimination

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

```
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){
  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;*/
  }
}
```

Just remove spawn and sync
in the *H*th termination condition.

3b

General Inlining

```
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;*/
  }
}
```

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

Inlining vecadd_seq() almost doubles the code size.

3b

Code-bloat-free Inlining(1/2)

```
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);
    }
  }
}
```

1. Delay execution of spawned tasks to corresponding sync.

3b

Code-bloat-free Inlining(2/2)

```
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);
        } */
      }
    }
  }
}
```

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.

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

3c

Loopification: Goal

- Try to convert recursion into a loop.

```
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.

```
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];
}
```

3c

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.

```
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 f(a, b, c, ...){
  if(...){
    //Leaf function.
    L(a, b, c, ...);
  }else{
    //Recursion block.
    /*spawn*/f(a0 , b0 , c0 , ...);
    /*spawn*/f(a1 , b1 , c1 , ...);
    ...
    /*sync*/
  }
}
```

Assumed input.

- ⬚ : 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.

```
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.

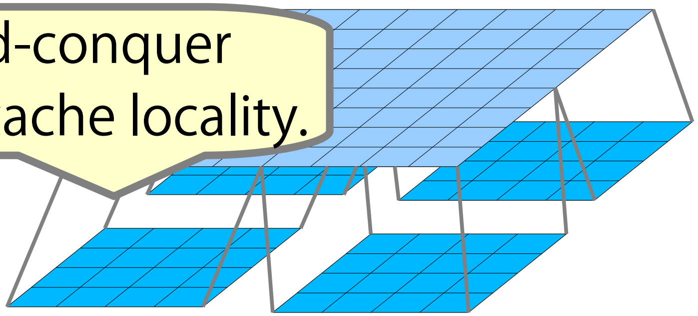
Why Loopification?

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.

```
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.

```
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;
  }
}
```

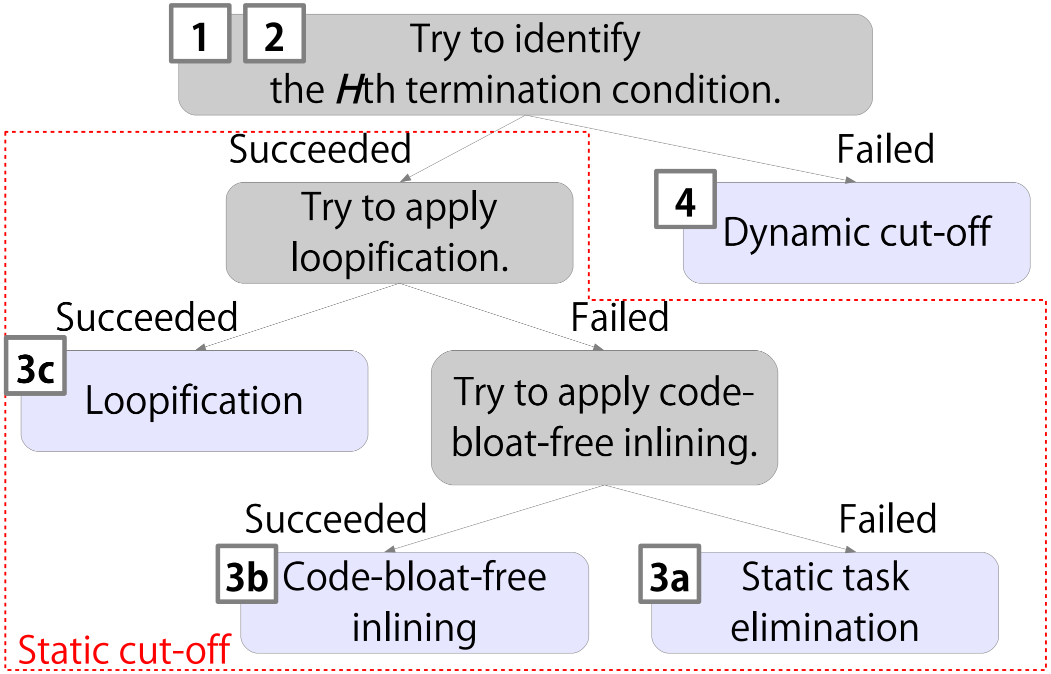
It's difficult to identify
the simple “*H*th termination condition”

- 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. [*]

[*] P. Thoman et al. Adaptive granularity control in task parallel programs using multiversioning. Euro-Par '13, 2013

Summary of Static Cut-off

- Our developed system...
  - 1 calculates an *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
  - 4 adopts dynamic cut-off if analysis (1) fails.



1 2 Try to identify the *H*th termination condition.

Succeeded: Try to apply loopification.

- Succeeded: 3c Loopification
- Failed: Try to apply code-bloat-free inlining.
  - Succeeded: 3b Code-bloat-free inlining
  - Failed: 3a Static task elimination

Failed: 4 Dynamic cut-off

Static cut-off

Index

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

[*1] MassiveThreads https://github.com/massivethreads/massivethreads

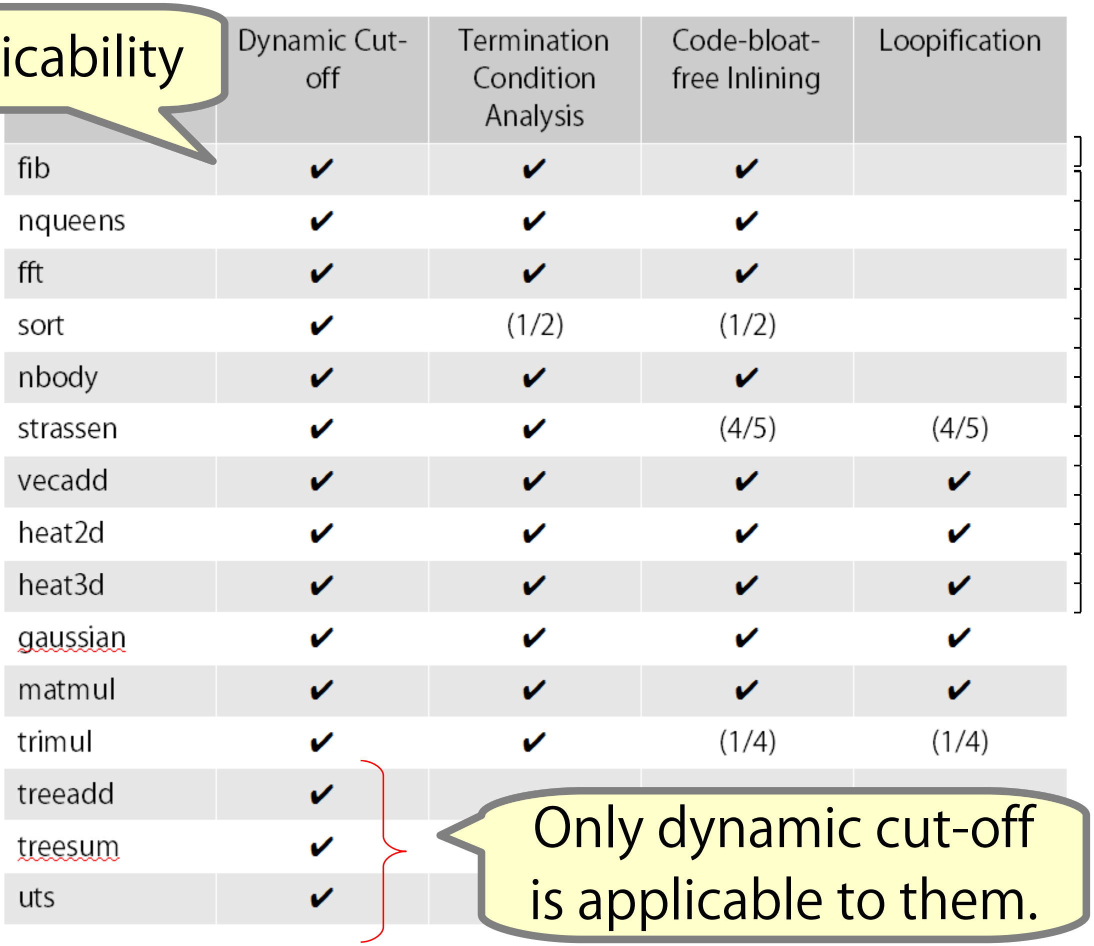
[*2] Mohr et al., Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs, LFP '90, 1990

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

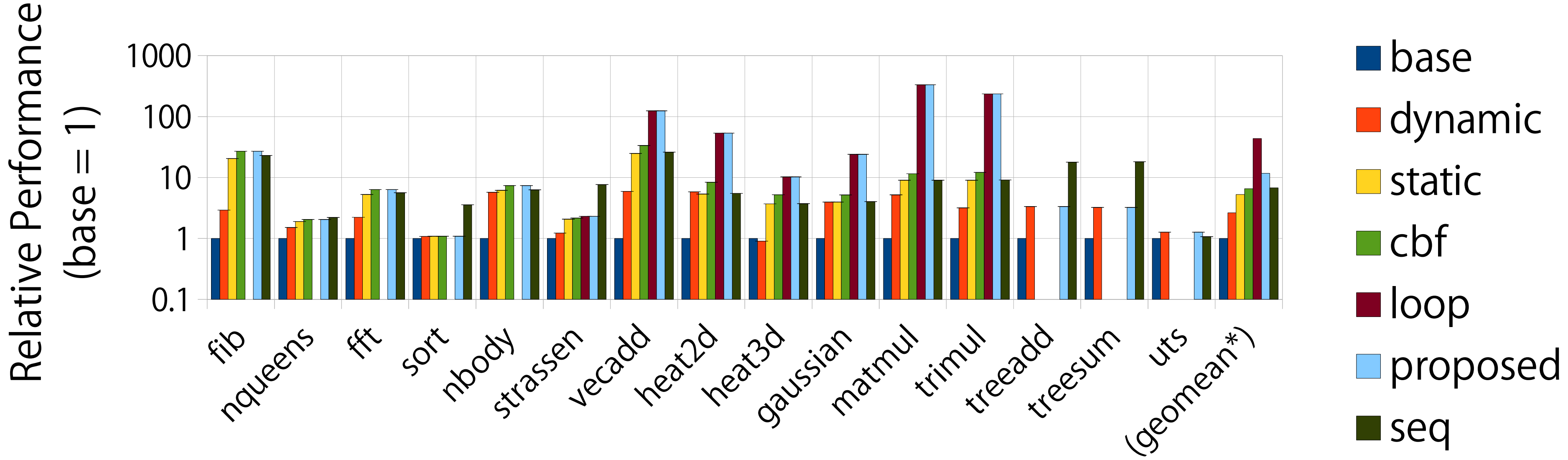
Applicability



	Dynamic Cut-off	Termination Condition Analysis	Code-bloat-free Inlining	Loopification
fib	✔	✔	✔	
nqueens	✔	✔	✔	
fft	✔	✔	✔	
sort	✔	(1/2)	(1/2)	
nbody	✔	✔	✔	
strassen	✔	✔	(4/5)	(4/5)
vecadd	✔	✔	✔	✔
heat2d	✔	✔	✔	✔
heat3d	✔	✔	✔	✔
gaussian	✔	✔	✔	✔
matmul	✔	✔	✔	✔
trimul	✔	✔	(1/4)	(1/4)
treeadd	✔			
treesum	✔			
uts	✔			

Only dynamic cut-off is applicable to them.

How to Read?

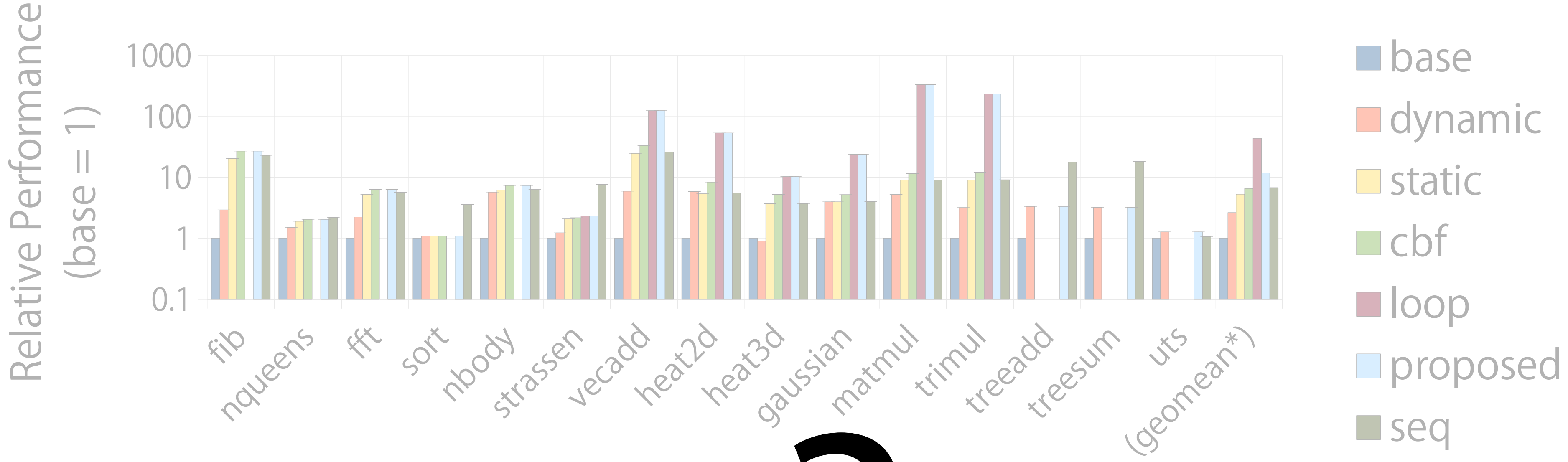


- Y-Axis: Relative performance over 🟦 base (divide-until-trivial)

  🟧 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?

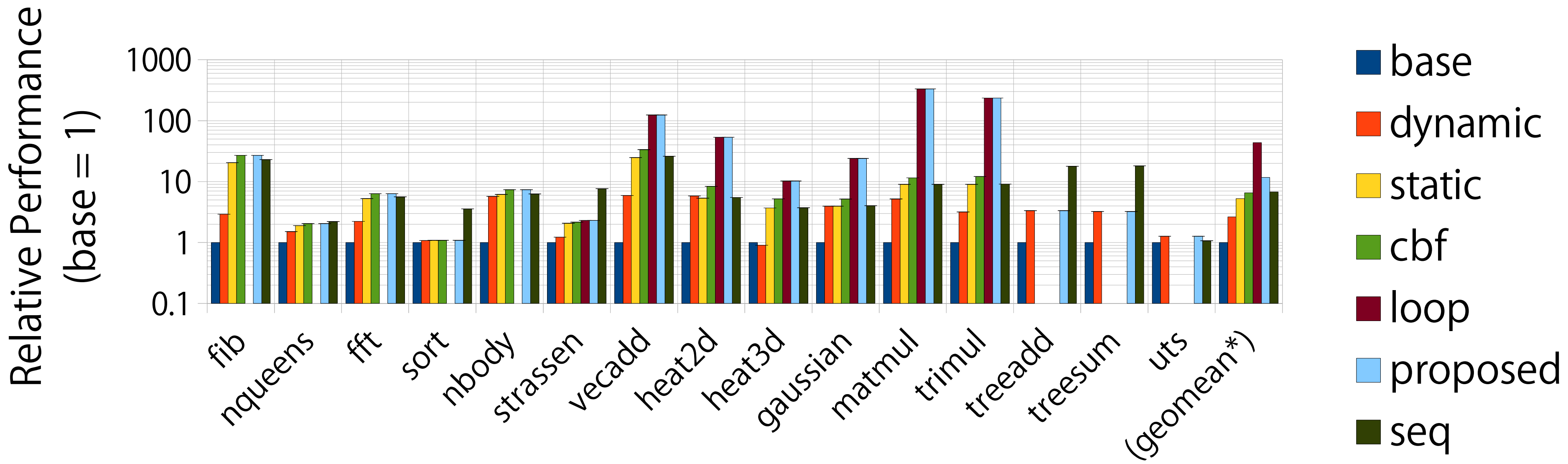


?

- 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.

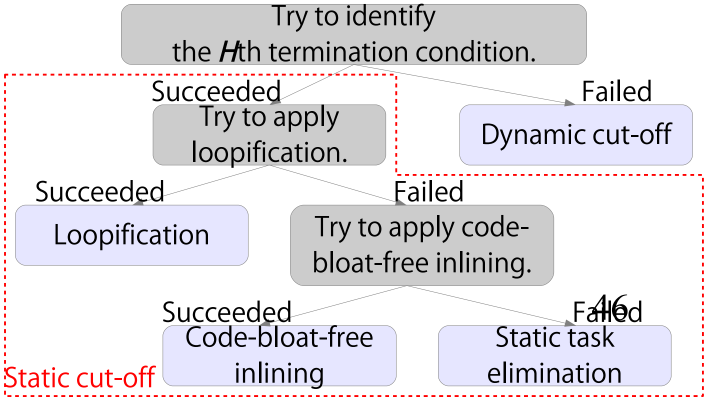
  🟨 static: static task elimination if applicable

  🟩 cbf: code-bloat-free inlining if applicable

  🟥 loop: loopification if applicable

  🟦 proposed: our proposal (the right chart →)

  🟩 seq: not task-parallelized



Try to identify the *H*th termination condition.

Succeeded → Try to apply loopification.

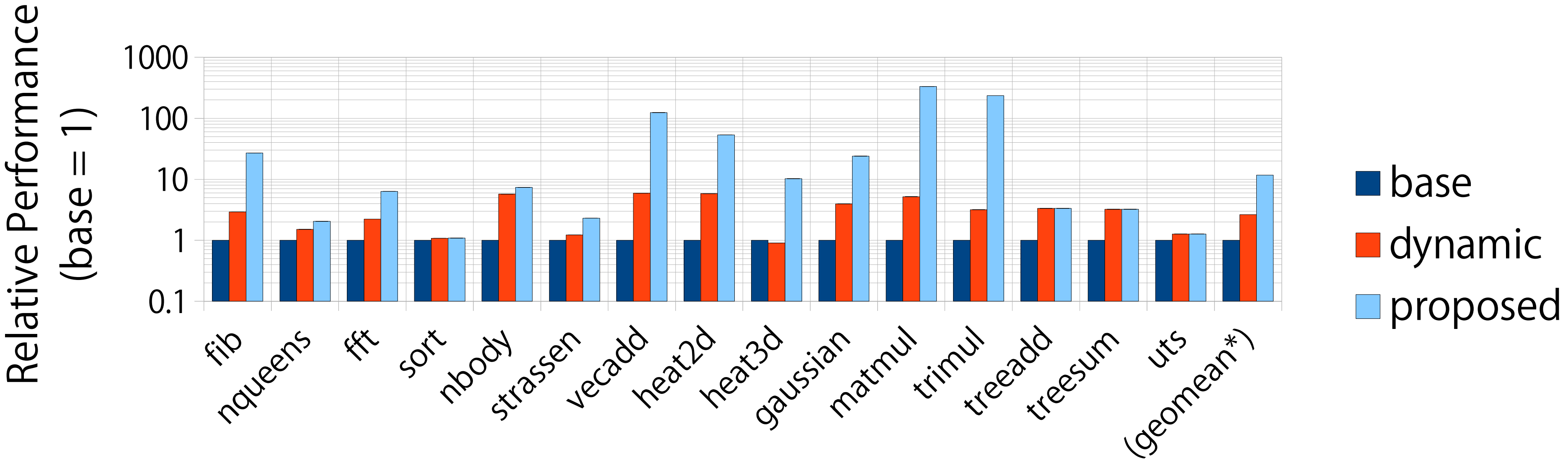
Failed → Dynamic cut-off

Try to apply loopification: Succeeded → Loopification; Failed → Try to apply code-bloat-free inlining.

Try to apply code-bloat-free inlining: Succeeded → Code-bloat-free inlining; Failed → Static task elimination

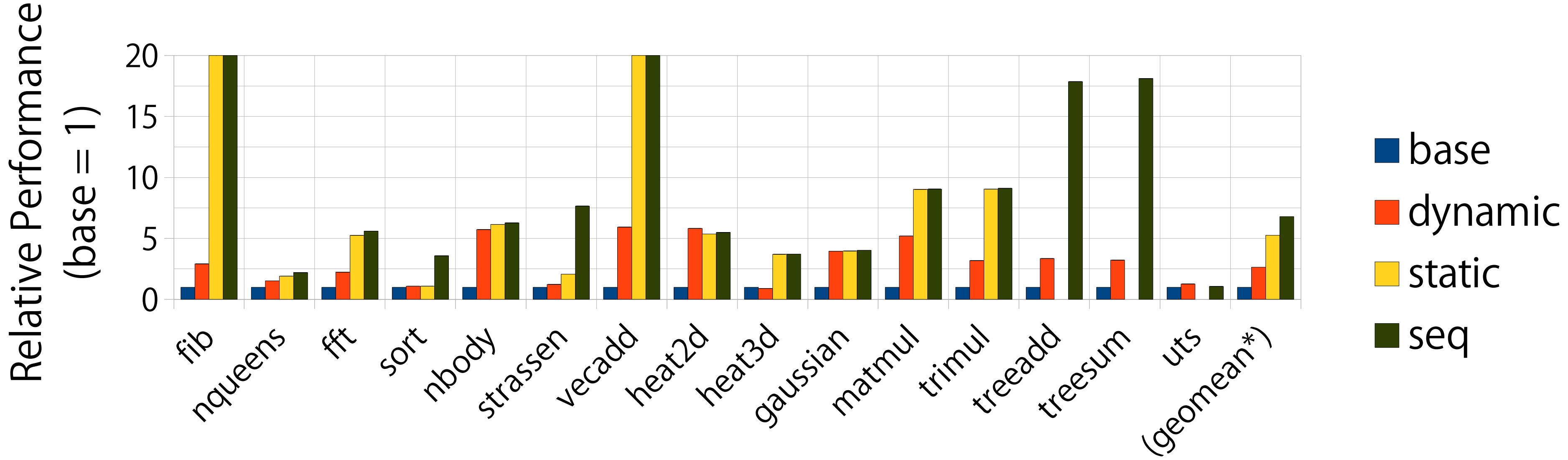
Static cut-off

Single-threaded Performance (1/3)



- Cut-off (🟥 & 🟦) improved performance overall.
- Compared to 🟥 dynamic cut-off,
  🟦 our proposed cut-off achieved higher performance.

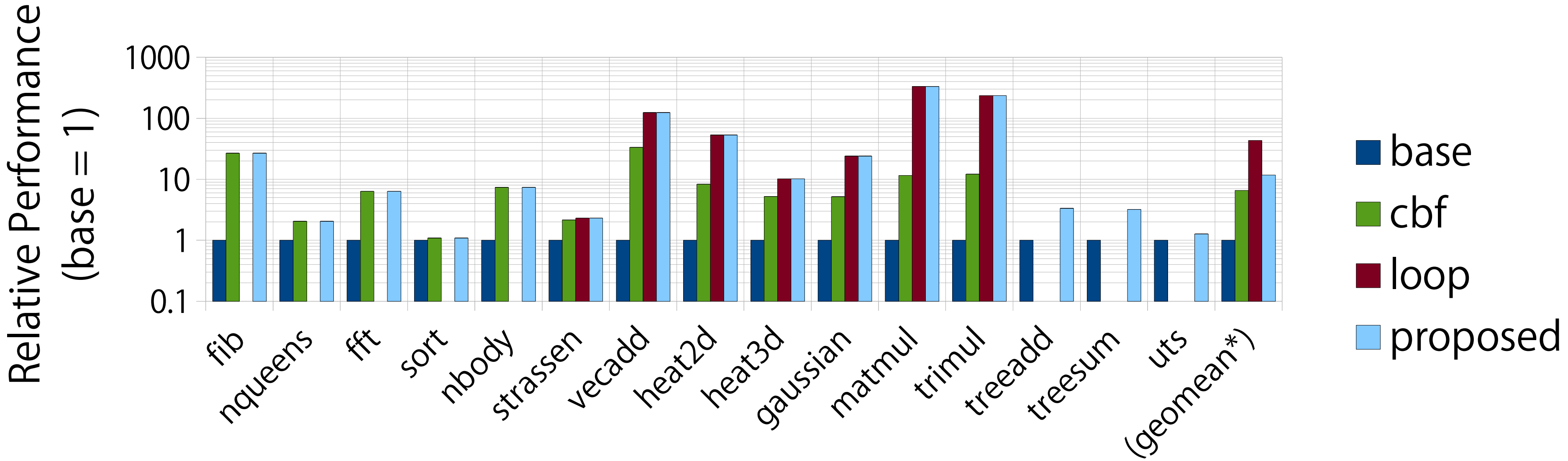
Single-threaded Performance (2/3)



- 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.

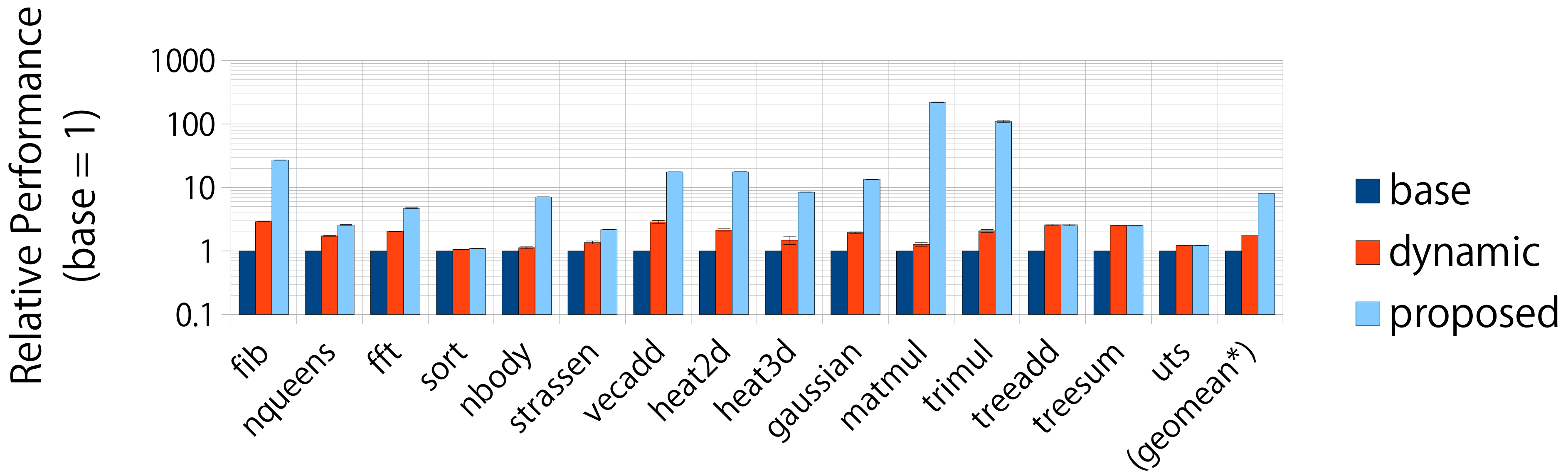
Single-threaded Performance (3/3)



- 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.

THE UNIVERSITY OF TOKYO

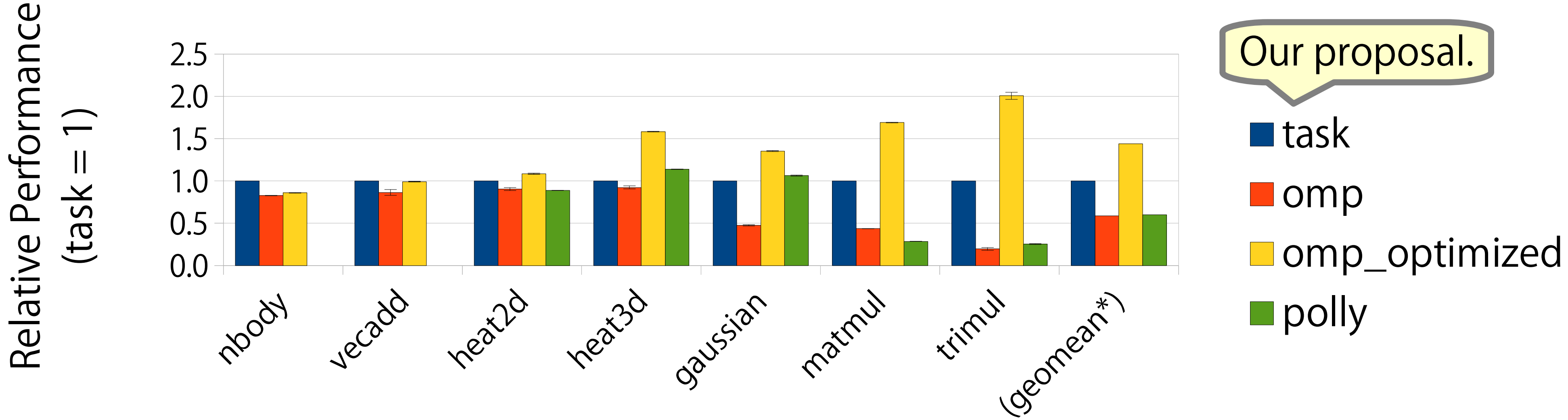
Multi-threaded Performance



  - 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



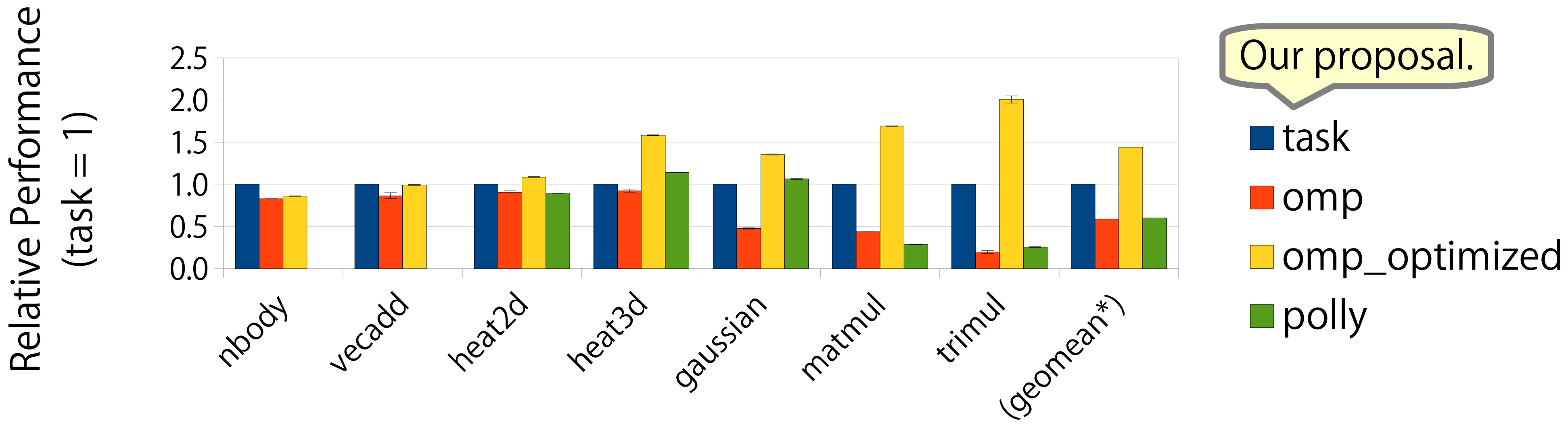
Our proposal.

- 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.

Even faster in some cases.

- 🟨 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

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*.