Autotuning of a Cut-off for Task Parallel Programs

Shintaro Iwasaki
Graduate School of Information
Science and Technology
The University of Tokyo
Tokyo, Japan
iwasaki@eidos.ic.i.u-tokyo.ac.jp

Kenjiro Taura
Graduate School of Information
Science and Technology
The University of Tokyo
Tokyo, Japan
tau@eidos.ic.i.u-tokyo.ac.jp

***Abstract*—A task parallel programming model is regarded as one of the promising parallel programming models with dynamic load balancing. Since this model supports hierarchical parallelism, it is suitable for parallel divide-and-conquer algorithms. Most naive divide-and-conquer task parallel programs, however, suffer from a high tasking overhead because they tend to create too fine-grained tasks. There are two key idea to enhance the performance of such a program: serializing a task in a cut-off condition which is a tradeoff between decrease of concurrency and parallelization overheads, and applying effective transformations for the task in the condition. Both are sensitive to algorithm features, rendering optimization solely with a compiler ineffective in some cases.**

To address this problem, we proposed an autotuning framework for divide-and-conquer task parallel programs. It automatically searches for the optimal combination of three basic transformation methods [1] and switching conditions with less programmers' efforts. We implemented it as an optimization pass in LLVM. The evaluation shows the significant performance improvement (from 1.5x to 228x) over the original naive task parallel programs. Moreover, it demonstrates the absolute performance obtained by our autotuning framework was comparable to that of loop parallel programs.

***Index Terms*—compilers; performance optimization; task parallelism; cut-off; autotuning;**

I. Introduction

Parallel programming becomes more and more important to exploit modern processors which employ increasing number of cores. A task parallel programming model supporting creation of fine-grained tasks and dynamic load balancing is believed to be a desirable solution for parallel programming with high performance and productivity. This model is especially suitable for divide-and-conquer algorithms since it provides hierarchical parallelism. Task parallelism is adopted by numerous well-known parallel systems and libraries such as Cilk [2], Intel Threading Building Blocks [3], and OpenMP [4].

However, it is very challenging to achieve high performance by just writing a simple task-based divide-and-conquer program. Significant performance degradation for such a program is particularly caused by a large tasking overhead; naive divide-and-conquer algorithms tend to create too fine-grained tasks, increasing a runtime cost for managing them. A "cut-off" is a common optimization technique to reduce the runtime overhead; it enlarges the granularity of tasks by rewriting a program to call corresponding functions instead of creating tasks in a certain condition. A condition switching to call serialized functions is referred to as a "cut-off condition". In addition to determining a cut-off condition, programmers are often required to write optimized version of a leaf task to fully elevate performance, which hinders productivity.

We developed compiler-based automatic cut-off techniques for divide-and-conquer task parallel programs [1]. This system automatically applies a cut-off with an appropriate cut-off condition which is statically obtained, and optimizes a serialized task with one of a few optimization methods based on the cut-off condition analysis. It also supports the state-of-the-art dynamic cut-off algorithm proposed by Thoman et al. [5] as a fallback if our cut-off condition analysis fails. This system fully automates the cut-off optimization, but we found it is hard to obtain the optimal cut-off condition without any execution in some cases because a cut-off condition is an essential factor controlling task granularity which balances amount of concurrency and parallelization overheads.

In this paper, we propose an autotuning framework for divide-and-conquer task parallel programs; it optimizes a task by searching for the best cut-off conditions and optimization combination of a few cut-off techniques our compiler employs [1]. The proposed framework is expected to be run on a target machine and requires a script for compilation and execution in addition to a code which contains a task function. Our framework can highly optimize programs by combining with the best cut-off condition and our optimization techniques. We implemented the transformation methods as an optimization pass in LLVM [6] and wrote an autotuning interface outside LLVM in Python.

This paper makes the following contributions:

- We developed a framework which highly optimizes task parallel programs without any cut-off using an autotuning technique.
- Our proposed framework significantly enhanced performance of task parallel programs, showing speedups of a geometric mean of 15.4x over the original unoptimized task parallel programs. The performance was comparable to, or even faster in some cases than the loop-based programs written in OpenMP [4] with GCC and Polly [7] with LLVM.

The organization of the rest of this paper is as follows. Section II gives a brief explanation of the current cut-off transformations we previously developed [1], and discusses

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

Fig. 1: Original task parallel Fibonacci (`fib`)

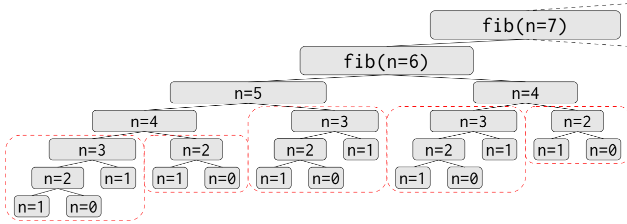


Fig. 2: Task tree of `fib` shown in Fig. 1. Tasks enclosed by a dotted red line are the cut-off candidates in the 2nd termination condition.

on three motivating cases to show the problems of our previous approach. Section III explains our proposed autotuning framework which tries to find the optimal combination of transformations and switching conditions. Section IV evaluates the performance obtained by our proposed autotuning system. Section V presents related work, and the following Section VI concludes this paper.

II. Cut-off Transformation

A. Static Cut-off Overview

To provide better understanding of our autotuning framework, this section briefly overviews our static cut-off techniques [1] before discussing the potential limitations of the pure compiler approach. Our static cut-off mainly consists of two components: termination condition analysis and some transformation methods.

For example, a task parallel program calculating an nth Fibonacci number shown in Fig. 1 suffers from a very large overhead because the naive `fib` creates too fine-grained tasks until `n` gets less than two. A cut-off is a widely-used technique to reduce such a overhead by rewriting a task to call a serialized function instead of creating a task in a certain condition. Humans can easily identify a condition in which the amount of the task's work is sufficiently small for a cut-off, but it is not the case for runtime systems or compilers. To strike a balance between flexibility of dynamic load balancing and low tasking overheads, our static cut-off system tries to identify an *Hth termination condition*, a condition composed of the arguments in which the height of the task tree rooted from that task is within a given height H. Consider a task tree of `fib` in Fig. 2. Dotted lines in the figure enclose the cut-off candidates with the $2nd$ termination condition. Divide-and-conquer tasks in an Hth termination condition with an appropriate H usually perform a small amount of work, thus serializing them will not incur serious loss of concurrency.

If a termination condition can be successfully obtained by the compiler analysis, our compiler tries to apply one applicable transformation method among the following three: static task elimination, code-bloat-free inlining, and loopification. Examples of resulting code are shown in Fig. 3.

Static task elimination shown in Fig. 3a performs a naive cut-off transformation by simply replacing task creations with sequentialized function calls under the obtained termination condition. In Fig. 3a, a serialized function `fib_seq` is newly created by removing task creations and the original task `fib` is converted to call it in the 2nd termination condition. This transformation is so simple that it is applicable to any tasks, but a large function call overhead remains if the original task is extremely fine-grained.

Inline expansion is the most popular method to reduce a function call overhead. The normal inlining is, however, not suitable for divide-and-conquer functions because it increases the code size exponentially if they have multiple self-recursions. To avoid code bloat, the compiler tries to aggregate the recursive calls into one and then apply inline expansion. We call this method *code-bloat-free inlining*. This code-bloat-free inlining replaces multiple recursive call sites with a loop containing one recursive call site, then applies the simple inline expansion H times. It increases the code size linearly, not exponentially. Fig. 3b describes the serialized `fib_seq` to which code-bloat-free inlining is applied.

We developed more aggressive optimization which generates a flat or shallowly nested loop instead of a deeply (i.e., H times) nested loop created by code-bloat-free inlining. For example, a divide-and-conquer vector addition task presented in Fig. 3c is apparently do the same work of a single loop presented in Fig. 3d for humans. Our optimization called *loopification* tries to do that by the analysis with a symbolic algebra solver and an Hth termination condition. Though the applicable range of this loopification is limited to the tasks which can be potentially represented as a regular loop (e.g., stencil kernels or dense matrix multiplication) for now, it is useful to write such a program in a divide-and-conquer manner to take an advantage of achieving the effect of cache blocking at all levels [8] without any explicit tiling.

In addition to these three optimizations, our automatic cut-off system also employs a dynamic cut-off strategy [9] in order to cover a wider range of tasks. The current implementation adopts the dynamic cut-off with multiversioning proposed by Thoman et al. [5], which is the state-of-the-art dynamic cut-off algorithm to the best of our knowledge. Their proposal can be applied without any static analysis. In our system, the dynamic approach is applied when the static analysis fails to calculate a termination condition.

In summary, the algorithm selection flow of the automatic cut-off system we previously proposed is as follows. Our compiler first runs the static analysis to obtain a cut-off condition, and then tries to apply loopification, code-bloat-free inlining, and static task elimination in this order if the analysis succeeds; otherwise the compiler applies the dynamic cut-off [5].

```
void fib(int n, int* ret){
  if(n < 4){
    fib_seq(n, ret);
  }else{
    int a, b;
    spawn fib(n-1, &a);
    spawn fib(n-2, &b);
    sync;
    *ret = a + b;
}}
void fib_seq(int n, int* ret){
  if(n < 2){
    *ret = n;
  }else{
    int a, b;
    fib_seq(n-1, &a);
    fib_seq(n-2, &b);
    *ret = a + b;
}}
```

(a) Static task elimination

```
void fib_seq(int n, int* ret){
  if(n < 2){
    *ret = n;
  }else{
    int a, b;
    for(int i = 0; i < 2; i++){
      int n2, *ret2;
      switch(i){
      case 0: n2=n-1; ret2=&a; break;
      case 1: n2=n-2; ret2=&b; break;
      }
      if(n2 < 2)
        *ret2 = n2;
      else
        [...]
    }
    *ret = a + b;
}}
```

(b) Code-bloat-free inlining

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

(c) Task parallel vector addition

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

(d) Loopification

Fig. 3: Examples of resulting code [1]

B. Problems of Our Previous Approach

We describe problems of the static cut-off system, which are difficult to solve by a pure compiler approach. We will focus on three motivating cases to see the problems concretely.

1) Inefficiency of Fallback Strategy: We apply the dynamic cut-off method if the compiler fails to analyze the termination conditions. Our static analysis fails due to lack of implementation in some cases, but there are tasks which essentially do not have simple termination conditions (e.g., tasks traversing pointer-based trees). For such a task, a manual cut-off commonly introduces a condition in which a depth from the root task is larger than a certain threshold D. This depth-based cut-off strategy is useful when a structure of the task tree is known to some extent in advance; otherwise, it may significantly reduce parallelism if the depth D is too small, or just imposes an overhead for evaluating the cut-off condition if D is too large. It is often the case, however, that the tree structure is hard to obtain at compile time.

The dynamic cut-off is an effective method to improve the performance of these tasks, but we found that it sometimes did not improve performance as expected due to an additional runtime overhead. It is ideal that the optimal depth parameter D is obtained at compile time, which allows the compiler to apply a straight-forward cut-off and enjoy further optimization for serialized tasks as well as the manual cut-off does.

2) Function Size Estimation: A cut-off threshold is an important parameter to balance between amount of parallelism and parallelization overheads. For tasks to which static task elimination or code-bloat-free inlining is applicable but loopification is not, the main purpose of cut-off is reduction of a tasking overhead. For example, if we want to maintain a tasking overhead lower than 2% of the total execution time, the task granularity should be at least 49 times larger than the task creation overhead. If this overhead is given as a constant, let say 100 cycles per task, the optimal height can be calculated by choosing the smallest height with which an estimated number of cycles of the function is more than 5000 cycles. Though this strategy seems effective on avoiding significant loss of parallelism, the estimation of the number of cycles is very difficult in reality; underestimation results in an ineffective cut-off while overestimation unnecessarily decreases parallelism. The `fib` task shown in Fig. 1 is a good example to see the difficulty in estimating an appropriate height parameter H, since the task tree of `fib` is not balanced as illustrated in Fig. 2.

3) Cache-Aware Loop Blocking Size: For tasks to which loopification is applicable, a cut-off is not only for alleviating a tasking overhead, but also for simplifying control flows to improve sequential performance. For such tasks, the cut-off height parameter H is more important; their divide-and-conquer strategies have an effect of recursive cache blocking [8] when the programs are converted into multi-dimensional loops. Since the best cache blocking size of the loopified task depends on application features and execution environments, the pure compiler approach can hardly address this problem.

III. Autotuning Framework

To tackle these problems originating from the difficulty in various estimations at compile time, we propose an autotuning framework for divide-and-conquer task parallel programs based on the automatic cut-off system we previously proposed [1]. This framework requires a target code written in LLVM-IR – an intermediate representation in LLVM – including a simple task parallel program, and a script file to compile and execute the program. As well as general autotuning frameworks do, it searches for the best parameters and transformation combination by executing programs on a target machine by changing compile options, and finally outputs a configuration file that can be used to compile the program with highly-optimized options. Fig. 4 presents the flow of our autotuning framework.

This autotuning approach itself is not so special for tuning parameters. Nevertheless, we note that there has been no recent work to focus on the potentials of the simple task parallel programs to the best of our knowledge. We believe a task written in a simple divide-and-conquer manner can achieve high performance on multi-threaded environments if compiler optimizations are properly applied.

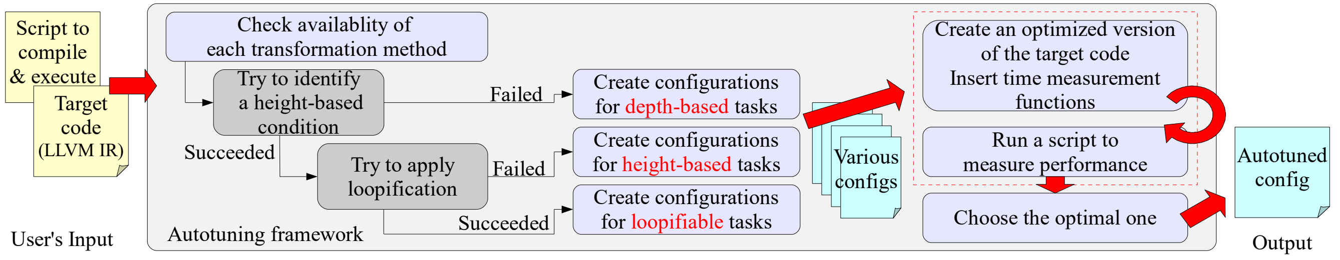
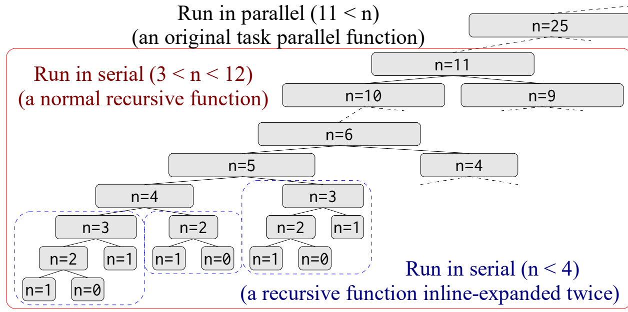


Fig. 4: Overview of our autotuning framework



(a) A task tree the strategy aims at

```
[fib]{
  fib:  original, mode=task, call fib2 (n < 12)
  fib2: original, mode=serial, call fib3 (n < 4)
  fib3: simple_inline (unroll=2), mode=serial
}
```

(b) A configuration to realize the strategy

```
void fib(int n, int* ret){
  if(n < 12){
    fib2(n, ret);
  }else{
    int a, b;
    spawn fib(n-1, &a);
    spawn fib(n-2, &b);
    sync;
    *ret = a + b;
}}
void fib2(int n, int* ret){
  if(n < 4){
    fib3(n, ret)
  }else{
    int a, b;
    fib2(n-1, &a);
    fib2(n-2, &b);
    *ret = a + b;
}}
void fib3(int n, int* ret){
  if(n < 2){
    *ret = n;
  }else
    [inlined twice]
}
```

(c) The final program

Fig. 5: Example of a possible optimization pattern

A. Basic Strategy

One of the notable divide-and-conquer features is that a parent divide-and-conquer task and its children are not necessarily the same program if both the parent and its children do the equivalent work; it is only required that the parent divides a problem into subproblems and the children conquer the subproblems. This feature enables us to optimize task parallel programs by just connecting differently transformed tasks with a conditional branch. Fig. 5 describes an example of one possible optimization pattern for `fib`. Fig. 5b shows a configuration of the optimization strategy above and Fig. 5c presents transformed code with the configuration. The final program contains three differently transformed tasks connected by conditional branches (`n < 12` and `n < 4`). This transformation has a cut-off effect since it stops task creation when `n < 12`. The naive search space is, therefore, transformation methods for tasks and conditional branches to connect them.

However, it obviously exposes countless transformation patterns because we can use any number of tasks and any kinds of conditional branches. To limit the search space, the targets are programs which 1) work in parallel when tasks are large and 2) run sequentially when they get smaller, and 3) execute a task to which special optimization for the leaf is applied if necessary. The transformation shown in Fig. 5 is based on this strategy. 1) Tasks in the upper side of the task tree shown in Fig. 5a run in parallel to have other cores work sooner. 2) To reduce a tasking overhead, tasks satisfying an Hth termination condition, say 10 as H, are serialized. 3) Furthermore, to reduce a function call overhead, this strategy also applies a simple inline-expansion twice to the serialized leaf tasks in the 2nd termination condition and then eliminates recursive calls in that function because it is known to be executed only under the 2nd termination condition.

PetaBricks [10] is the most famous autotuning framework focusing on this divide-and-conquer feature, but it is designed to require users to write multiple versions of divide-and-conquer programs. Our compiler can create multiple tasks derived from an original task by applying several optimizations: inline-expansion, loopification, removing task creations, and/or inserting a conditional branch to another task transformed in different ways. Our framework thus searches for the best combination of transformations and connections of the divide-and-conquer tasks with an autotuning strategy.

B. Available Optimization Patterns

To describe the autotuning method based on the basic strategy we discussed in detail, we explain possible transformation patterns for task parallel programs with our current compiler. There are two fundamental elements for each task; 1) an optimization method for a task, and 2) a condition to branch another function based on either depth from the root or height above the leaves in addition to a function to which the task branches in that condition.

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

(a) No optimization

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

(b) Inline-expansion (once)

```
void vadd(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;
      }
      if(n2==1) *a2+=*b2;
      else{
        for(int i2=0;i2<2;i2++){
          float *a3, *b3; int n3;
          switch(i2){
          case 0: a3=a2; b3=b2; n3=n2/2; break;
          case 1: a3=a2+n2/2; b3=b2+n2/2;
                  n3=n2-n2/2;
          }
          vadd(a3,b3,n3);
}}}}}
```

(c) code-bloat-free inlining (once)

```
//assume 1<=n && n<=2
void vadd(float* a,float* b,int n){
  if(n==1) *a+=*b;
  else{
    if(n/2==1) *a+=*b;
    if(n-n/2==1) *(a+n/2)+=*(b+n/2);
}}
```

(d) Complete inline-expansion (in the 1st termination condition)

```
//assume 1<=n && n<=2
void vadd(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;
      }
      if(n2==1) *a2+=*b2;
}}}
```

(e) Complete code-bloat-free inlining (in the 1st termination condition)

```
//assume 1<=n && n<=2
void vadd(float* a,float* b,int n){
  for(int i=0;i<n;i++)
    *(a+i)+=*(b+i);
}
```

(f) Loopification (in the 1st termination condition)

Fig. 6: Six transformation methods

1) Optimization Methods: Our compiler supports three basic transformations: normal inlining, code-bloat-free inlining, and loopification. Derived from these three, six versions of transformations are employed. Three transformations remain tasks recursive, so the optimized tasks can be used as either internal nodes or leaves of the task trees. The other three translate them into serial functions which can be used as only leaves of the trees because resulting tasks have no recursive calls. Fig. 6 illustrates the six divide-and-conquer vector addition tasks generated with these methods above. As shown in the figure, tasks shown in (a), (b), and (c) can be used as both internal nodes or leaves, whereas (d), (e), and (f) are no longer self-recursive.

We note that our autotuning framework is capable of adopting optimization methods other than inlining methods or loopification due to the divide-and-conquer feature.

2) Conditional Branch to Other Task: A conditional branch to another task is a key to connecting multiversioned tasks. Considering a branch from a parallelized function to a serialized function for example, this concept includes the idea of a cut-off. Thus determining a branch condition has a large impact on balancing concurrency and parallelization overheads. Two parameters can constitute a branch condition: a depth from the root and a height above the leaves. As we discussed, a height-based condition is more suitable for a cut-off, but the termination condition analysis is necessary to identify the height-based condition. On the other hand, the depth-based cut-off is applicable to any tasks because the depth can be easily obtained by incrementing the depth variable from the root. A condition can contain either of them, or in theory, combine them if both available (e.g., a condition in which height is less than 8 or depth is more than 10).

C. Search Space

Since transformed tasks can be connected to any other tasks, the possible patterns are countless. We therefore follow the basic strategy argued in Section III-A to search a limited space which only contains reasonable transformation patterns. Let us assume there is only one self-recursive task to optimize. Our proposed system classifies an input task into three patterns corresponding to the motivating cases described in Section II-B. The common direction among these patterns is simple: tasks near leaves are transformed into serialized ones for further optimizations, while tasks near the root should be original in order not to decrease concurrency. We explain the three patterns.

1) Depth-based Cut-off: Our system applies a depth-based cut-off to a task in which the termination condition analysis fails. Our previous cut-off approach adopts the dynamic cut-off for such a task. We found, however, that it could not always enhance performance as expected. Our autotuning framework chooses an appropriate depth by implicitly utilizing information of a structure of the task tree by executing a program, whereas the pure compiler approach cannot acquire such information. The depth parameter D is determined as a depth which is the largest depth to achieve 99% of the best performance obtained during the autotuning; the 99% criterion is imposed to avoid unnecessary reduction of parallelism.

We cannot eliminate self-recursive function calls since the analysis fails to obtain the exact termination condition, so tasks after the cut-off must be functions which can be used as internal nodes ((a) to (c) in Fig. 6); simple inlining and

code-bloat-free inlining are therefore only applicable to the serialized function.

2) Height-based Cut-off (No Loopification): A height-based cut-off is applied to a task if the static analysis succeeds to obtain an Hth termination condition, but loopification fails. We use a height-based cut-off condition for such a task since it is expected to bind the size of each serialized task to some degree regardless of input size. The smallest height to achieve 99% performance compared to the best one is used as the cut-off parameter H as well as the depth-based cut-off does.

Optimizations based on inlining can be applied to the serialized function in a certain termination condition. We consider the following six patterns. The first pattern attempts to optimize tasks by simply unrolling a function once or multiple times. The second applies code-bloat-free inlining instead, which can inline functions more times when the resulting code size is restricted. In order to remove recursive function calls in a leaf function, the third and the fourth adopt *complete* inline expansion and *complete* code-bloat-free inlining, illustrated in Fig. 6d and Fig. 6e. These four patterns above introduce a simple recursive function between the original task and the leaf function to connect them if inlining the leaf function cannot be repeated H times due to the code size limitation. The fifth and sixth patterns are similar to the third and fourth, but replace the simple recursive function with a function optimized by simple inlining or code-bloat-free inlining to reduce a function call overhead more.

3) Cut-off with Loopification: Loopification, if applicable, improves performance drastically compared to the original task parallel programs. In this case, a cut-off condition affects performance of the loopified function significantly because the cut-off virtually changes the size of cache blocking. Our system therefore employs a special autotuning method for loopifiable tasks. As a height parameter H, this system adopts the smallest height with which the loopified program can achieve more than 99% performance compared to the best performance. After that, it changes the loopification condition with the cut-off threshold fixed in order to achieve the best cache blocking.

D. Preprocessing for Autotuning

Our autotuning framework first checks availability of each transformation method for the task because not all the transformations are applicable to all tasks. Our framework skips patterns including inapplicable transformations. It is also required to add additional instructions for the performance measurement; the system automatically creates an interface function for calls from outside (i.e., not from a task) and inserts performance measurement function calls into the function, which lowers the programmers' loads to put them manually.

E. Autotuning Results

After executing programs with various optimization patterns, our autotuning system writes down the optimal optimization pattern to the configuration file. This file can be used for the input of our compiler to optimize a program with the optimal parameters. It is written in a human-readable format, so it may provide hints to manual optimization.

IV. Evaluation

The transformations discussed in the previous section were implemented as an optimization pass in LLVM 3.6.0 [6]. The interface of our autotuning framework was written in Python. Task parallel programs were executed on MassiveThreads [11], a lightweight task library adopting a child-first scheduling strategy [12].

For the evaluation, we prepared eleven benchmarks containing a task function without any manual cut-off. The benchmarks are as follows:

1) **fib** calculates a 45th Fibonacci number.
2) **nqueens** counts solutions of the N-Queens problem with $N = 14$. Both of **fib** and **nqueens** adopt the same task creation patterns in benchmarks of BOTS [13].
3) **nbody** calculates forces between all N to N pairs, with $N = 30K$. Each particle has mass, 3D position, velocity, and a temporal variable to accumulate the force.
4) **vecadd** adds two float arrays and stores the result into another array. Each array has 10^9 elements.
5) **heat2d** is a stencil computation, solving two-dimensional thermal diffusion equation on a $30K \times 30K$ mesh.
6) **heat3d** is a 3D version of **heat2d** on a $(1K)^3$ mesh.
7) **gaussian** applies a 5×5 Gaussian filter to an array of $30K \times 30K$ single precision floating point numbers.
8) **matmul** multiplies two matrices and stores the result into another matrix. All three matrices have 2000×2000 single precision floating point numbers.
9) **treeadd** receives a pointer-based binary tree structure and traverses it to update values at the leaves. The input is a balanced tree including $2^{30} - 1$ elements.
10) **treesum** receives a pointer-based binary tree and sums up all the values in the leaves by tree traversal. The input is the same of **treeadd**.
11) **uts** runs Unbalanced Tree Search [14]. The input parameter set is "T1XL" in the official samples, generating a geometric tree with 1,635,119,272 elements.

TABLE I: Data size for an evaluation shown in Fig. 8

nbody	$(40K)^2$	vecadd	$2G$	heat2d	$(40K)^2$
heat3d	$(1.2K)^3$	gaussian	$(40K)^2$	matmul	$(5K)^2$

TABLE II: Applicability of cut-off methods

(code-bloat-free inlining is abbreviated to "cbf" in this table)

	dynamic	static	cbf	loopification
fib	✓	✓	✓	
nqueens	✓	✓	✓	
nbody	✓	✓	✓	
vecadd	✓	✓	✓	✓
heat2d	✓	✓	✓	✓
heat3d	✓	✓	✓	✓
gaussian	✓	✓	✓	✓
matmul	✓	✓	✓	✓
treeadd	✓			
treesum	✓			
uts	✓			

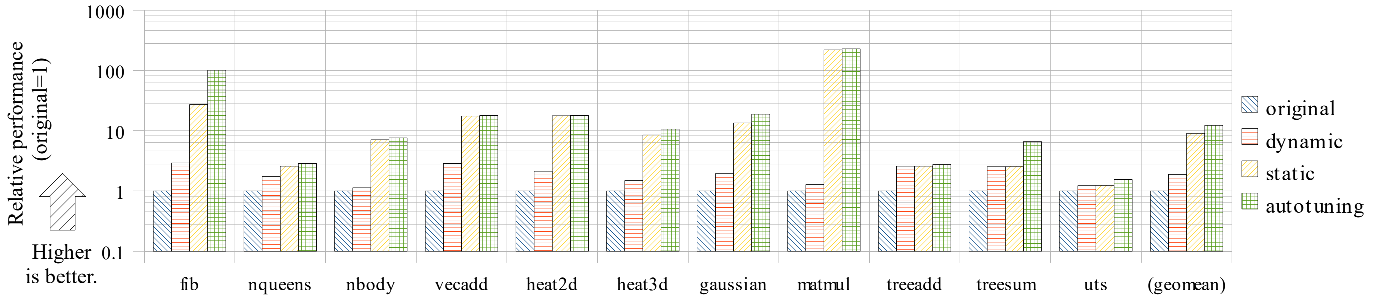


Fig. 7: Multi-threaded performance. It shows relative performance of the original (original), the dynamic multiversioning [5] (dynamic), our static cut-off [1] (static), and the autotuning we propose in this paper (autotuning)

For the performance evaluation shown in Fig. 8, we instead used benchmarks with larger input data presented in Table I to make a single execution time longer than 0.2 seconds. Table II shows the applicability of optimizations to the eleven benchmarks.

The benchmarks written in C language were first translated into LLVM IR using Clang, a frontend C/C++ compiler of LLVM. Then, we input them into our autotuning framework. Finally we compiled them into machine language programs with the LLVM compiler with optimization flags `-O3 -ffp-contract=fast` and machine-specifying options.

Experiments were conducted on dual sockets of Intel Xeon E5-2699 v3 (Haswell) processors (36 cores in total). We ran every benchmark with `numactl --interleave=all` to balance physical memory across sockets. All results in the charts show the averages of five measurements.

A. Performance compared to task parallel programs

Fig. 7 presents the performance improvement using 36 threads. We measured performance of the original programs (**original**), the dynamic cut-off [5] (**dynamic**), the static cut-off [1] (**static**) and our proposal (**autotuning**). For the dynamic cut-off, since we were unable implement the simplification in the unrolling step due to lack of details, we inlined it and applied the LLVM's maximum `-O3` optimization as our best effort. Our autotuning framework repeated executing each pattern three times for search and took the geometric mean of 894 times as long as the optimal execution time finally obtained. The baseline is a task parallel program without any cut-off (**original**).

As shown in Fig. 7, our optimization achieved a significant speedup in comparison to **original**, **dynamic** and **static**; our autotuning framework did from 1.5x to 228x (geometric mean of 12.2x), while the dynamic cut-off elevated performance from 1.1x to 2.9x (geometric mean of 1.9x) and our static cut-off did from 1.2x to 220x (geometric mean of 9.0x).

Compared to the static cut-off using the same compiler transformations, this result itself was not very surprising considering the mechanism of the autotuning. The result indicated the static cut-off left room for performance improvement especially for **fib** and **gaussian**, and some tasks which our static analysis failed: **treesum**, **uts**. The configuration files our autotuning framework created provide the following insights.

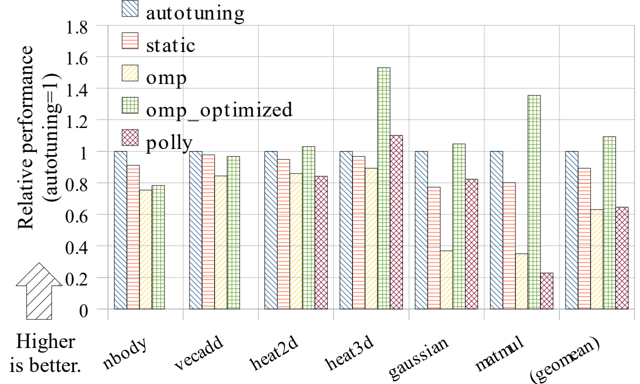


Fig. 8: Performance comparison of autotuned task parallel programs to loop parallel programs. The baseline is task parallel programs optimized by our autotuning framework. Note `(geomean)` of `polly` excludes the cases in which optimization failed.

fib was a case in which the cut-off height selected in the static cut-off was much smaller. For **gaussian**, a larger loop was preferable as its leaf calculation. Performance of **treesum** and **uts** was not fully elevated by the dynamic cut-off due to its overheads.

B. Performance compared to loop parallel programs

We manually loop-parallelized **nbody**, **vecadd**, **heat2d**, **heat3d**, **gaussian**, and **matmul** all of which are expressible in loops and compared them with the autotuned task parallel programs. They performed the same calculations of the corresponding task parallel programs, while parallelization strategies and execution orders were different. The loop programs were parallelized by OpenMP [4] and Polly [7]. One used OpenMP with `omp parallel for` and was compiled by GCC 4.8.4, with `-O3 -ffp-contract=fast`. The other was automatically parallelized by Polly [7], a locality-optimizer for LLVM based on a loop polyhedral model. It was compiled by Clang 3.8.0 with `-O3 -ffp-contract=fast` and its Polly with `-polly -polly-parallel` and `-polly-vectorizer = stripmine` if it made the programs faster.

Fig. 8 presents the comparison of the optimized task parallel programs to the loop parallel programs. In the figure, **autotuning** is performance of task parallel programs optimized by our autotuning framework, and **static** is performance obtained

by the static cut-off. **omp** and **polly** are those of simple loop parallel programs optimized by OpenMP with GCC and by Polly with Clang respectively. We also made **omp_optimized**, a hand-tuned OpenMP version optimized by changing blocking sizes, scheduling strategies, scheduling parameters (chunk size, etc.), and collapse clauses for nested loops. The results of **polly**'s **nbody** and **vecadd** are omitted in the chart because they were not parallelized. The autotuning version (**autotuning**) is a baseline. Fig. 8 shows the performance of **autotuning** was overall faster than **static**, **omp** and **polly**; the geometric mean of **static**'s relative performance was 0.89x and those of **omp** and **polly** were 0.63x and 0.65x. Furthermore, **autotuning** was nearly comparable to **omp_optimized**, which boost performance to show that of 1.1x. It suggests that task parallel programs carefully optimized by a compiler may reach the performance of hand-optimized loop parallel programs by utilizing advantages of divide-and-conquer features (e.g., recursive cache-blocking).

V. RelatedWork

A. Autotuning framework

Autotuning is a well-known approach to optimize parameters which cannot be determined analytically or by using simple heuristics. ATLAS [15] and FFTW [16] are the most famous autotuning software to highly optimize specific programs. PetaBricks proposed by Ansel et al. [10] is a general autotuning system containing an original language and its compiler, which mainly focuses on tuning algorithmic choices by utilizing divide-and-conquer features. The basic concept is similar to ours, but PetaBricks requires programmers to write multiple implementations of the algorithms in their original language, while our goal is to achieve high performance with simple task parallel programs basically written in C/C++.

B. Dynamic cut-off

A dynamic cut-off was firstly proposed by Duran et al. [9] as an adaptive cut-off; the runtime calls a function instead of creating tasks if the runtime information such as depth of the task and the number of ready tasks tells there exist sufficiently many tasks. The state-of-the-art dynamic cut-off algorithm proposed by Thoman et al. [5] is a multiversioning method which creates multiple versions including inlined versions and fully serialized one, and switches between them based on the runtime information. Though these runtime-based approaches are in general applicable to a wider range of tasks, they reveal little opportunity for applying aggressive optimizations including autotuning. Combining autotuning and the dynamic approach is our future work.

VI. Conclusion

This paper describes an autotuning framework to optimize divide-and-conquer task parallel programs, which combines multiple transformation techniques based on a divide-and-conquer feature. The evaluation shows significant speedup by our proposed framework whose performance is better than that of the dynamic cut-off and comparable to that of manually-optimized loop parallel programs.

Acknowledgement

This work was in part supported by Grant-in-Aid for Scientific Research (A) 16H01715.

References

[1] S. Iwasaki and K. Taura, "A static cut-off for task parallel programs," in *Proceedings of the 25th International Conference on Parallel Architectures and Compilation Techniques (PACT '16)*, Haifa, Israel, Sept. 2016, (to appear).

[2] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," in *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '95)*, Santa Barbara, California, USA, Jul. 1995, pp. 207–216.

[3] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly Media, 2007.

[4] OpenMP Architecture Review Board, "OpenMP Application Program Interface Version 3.0," Tech. Rep. May, 2008.

[5] P. Thoman, H. Jordan, and T. Fahringer, "Adaptive granularity control in task parallel programs using multiversioning," in *Proceedings of the 19th International Conference on Parallel Processing (Euro-Par '13)*, Aachen, Germany, Aug. 2013, pp. 164–177.

[6] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '04)*, San Jose, California, USA, Mar. 2004, pp. 75–.

[7] T. Grosser, H. Zheng, R. Aloor, A. Simbürger, A. Größlinger, and L.-N. Pouchet, "Polly - Polyhedral optimization in LLVM," in *Proceedings of the 1st International Workshop on Polyhedral Compilation Techniques (IMPACT '11)*, Chamonix, France, Apr. 2011.

[8] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS '99)*, New York City, New York, USA, Oct. 1999, pp. 285–.

[9] A. Duran, J. Corbalán, and E. Ayguadé, "An adaptive cut-off for task parallelism," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (SC '08)*, Austin, Texas, USA, Nov. 2008, pp. 36:1–36:11.

[10] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe, "PetaBricks: A language and compiler for algorithmic choice," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*, Dublin, Ireland, Jun. 2009, pp. 38–49.

[11] J. Nakashima and K. Taura, "MassiveThreads: A thread library for high productivity languages," in *Symposium on Concurrent Objects and Beyond. From Theory to High-Performance Computing*, Kobe, Japan, 2014, pp. 222–238.

[12] E. Mohr, D. A. Kranz, and R. H. Halstead, Jr., "Lazy task creation: A technique for increasing the granularity of parallel programs," in *Proceedings of the 1990 ACM Conference on LISP and Functional Programming (LFP '90)*, Nice, France, Jun. 1990, pp. 185–197.

[13] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade, "Barcelona OpenMP Tasks Suite: A set of benchmarks targeting the exploitation of task parallelism in OpenMP," in *Proceedings of the 2009 International Conference on Parallel Processing (ICPP '09)*, Vienna, Austria, Sept. 2009, pp. 124–131.

[14] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, and C.-W. Tseng, "UTS: An unbalanced tree search benchmark," in *Proceedings of the 19th International Conference on Languages and Compilers for Parallel Computing (LCPC '06)*, New Orleans, Los Angeles, USA, Nov. 2007, pp. 235–250.

[15] R. C. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software," in *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing (SC '98)*, San Jose, California, USA, Nov. 1998, pp. 1–27.

[16] M. Frigo and S. G. Johnson, "FFTW: An adaptive software architecture for the FFT," in *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP '98)*, vol. 3, Seattle, Washington, USA, May 1998, pp. 1381–1384 vol.3.