Autotuning 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 parallelization of 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. To make the matter worse, reducing the overhead by replacing task creations with simple function calls in a certain condition is often not enough to obtain satisfactory performance; the serialized programs still contain multiple recursive call sites, degrading performance due to their complicated control flows. There are two key idea to enhance the performance of such programs: determining an optimal serialization condition which is a tradeoff between decrease of concurrency and a parallelization overhead, and applying an effective transformation for tasks in such a condition. Both are sensitive to algorithm features, rendering optimization solely with a compiler ineffective in most cases.**

To address this problem, we proposed an autotuning framework for divide-and-conquer task parallel programs, which automatically searches for the optimal combination using three basic transformation methods and selects switching conditions with less programmers' efforts. We implemented the proposed autotuning method 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.

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 the dynamic load balancing is one of the most well-known parallel programming model, which is 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 widely-used parallel systems and libraries such as Cilk [2], Intel Threading Building Blocks [17], and OpenMP [15].

However, it is a very challenging problem 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 often 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 transforming programs to simply call corresponding functions instead of creating tasks in a certain condition. A condition switching to call a serialized function is called a "cut-off condition". This condition is determined by programmers in general, which hinders productivity. To make the matter worse, the cut-off alone does not always fully elevate performance; there is still plenty of room for further aggressive compiler optimization for divide-and-conquer programs after removing task creations.

Our previous work [10] has developed a compiler-based automatic cut-off system for divide-and-conquer task parallel programs. This system includes two components. One is an optimization compiler which automatically applies a cut-off with an effective cut-off condition obtained statically, and then optimizes a serialized task with one of a few aggressive optimization methods based on the dependency information revealed by the semantics of task parallelism. The other is a task parallel runtime system supporting the state-of-the-art dynamic cut-off algorithm proposed by Thoman et al. [22] as a fallback if our compiler fails to apply the static cut-off. Our evaluation has shown that the proposed cut-off system drastically improved performance of naive divide-and-conquer task parallel programs, while our cut-off system strongly depends on both heuristics and cost estimation functions for determining optimal cut-off conditions and optimization methods, both of which in general do not always present right optimization directions or exact function costs. An optimization framework is demanded which can generate programs achieving the "best" performance without rewriting original task parallel programs.

In this paper, we propose an autotuning framework for divide-and-conquer task parallel programs, by searching for the best cut-off thresholds and optimization combinations our static cut-off system has already employed. 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 generate programs with higher performance by combining with the best cut-off condition and our proposed optimization techniques. We implement the transformation methods as an optimization pass in LLVM [11], and create the interface outside LLVM by Python.

```
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`)

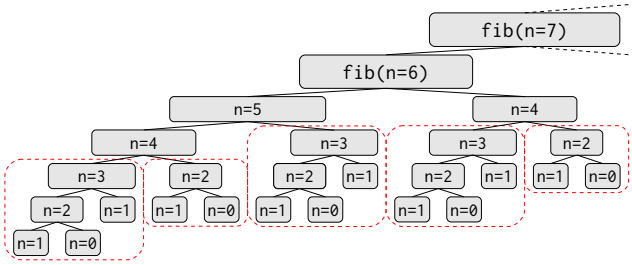


Fig. 2: Task tree of `fib`. Tasks enclosed by a dotted line are the cut-off candidates in the 2nd termination condition.

This paper makes the following contributions:

- We developed an autotuning framework for task parallel programs which essentially requires only unoptimized task parallel programs.
- 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 with GCC and Polly [8].
- The resulting program generated by our autotuning framework showed some insights into optimization for task parallel programs, which is especially useful for further improvement of our static cut-off technique.

The organization of the rest of this paper is as follows. Section II first describes the current static cut-off system we have already developed, and then focuses on three motivating cases to show the problems of the static cut-off. Next, Section III explains our proposed autotuning framework, which tries to find an optimal combination of transformation and cut-off condition. Section IV evaluates the performance obtained by our proposed autotuning system. Section V presents related work, and the following Section VI discusses our future work for conclusion of this paper.

II. Static Cut-off and Its Limitations

A. Static Cut-off Overview

To provide better understanding of our autotuning framework, this section overviews the static cut-off system we have developed [10], and then explains the potential limitations of the pure compiler approach using three motivating cases. Our static cut-off mainly consists of two components: termination condition analysis and optimization with three basic transformation methods.

Consider a program calculating an nth Fibonacci number in Fig. 1 as a running example. The original `fib` creates extremely fine-grained tasks until `n` gets equal to one, making a tasking overhead high. A cut-off is a widely-used method to reduce a tasking overhead by coarsening the task granularity. From a viewpoint of parallelism, enlarging the granularity however reduces flexibility of dynamic load balancing the original program potentially has; an exceeded cut-off results in serious loss of concurrency. In our static cut-off system, the compiler only tries to apply a cut-off for tasks which are proven to be near leaves of the task tree, as is commonly applied in a manual cut-off. The compiler, therefore, needs to identify a condition on function's arguments in which the height of the task tree from the leaves is within a given constant H. We call this condition an *Hth termination condition*. For instance, Fig. 2 shows a task tree of `fib`. Dotted lines in the figure enclose the cut-off candidates with the $2nd$ termination condition in which the heights from leaves of all candidates are not greater than 2. With an appropriately chosen height parameter H, serializing tasks under such conditions will not decrease the concurrency seriously since they will perform a small amount of work from a viewpoint of a divide-and-conquer strategy. To obtain the cut-off condition with a given height parameter, a special control flow analysis is implemented; it recursively calculates a kth termination condition using a $k-1$th termination condition where a 0th termination condition is a condition in which the task itself never creates child tasks.

If the compiler successfully obtains a cut-off condition, it tries to apply one applicable transformation methods among static task elimination, code-explosion-free inlining, and loop-ification. Each transformation is visually presented in Fig. 3.

Fig. 3a shows static task elimination which is the simplest cut-off transformation among them: it just replaces task creations with simple function calls under the termination condition. In this example, the original task function `fib` is transformed to call a serialized function `fib_seq` in the 3rd termination condition (`n` < 4). It is often the case, however, that just reducing a tasking overhead is insufficient to achieve a good speedup because this transformation leaves the serialized function to call recursive functions several times; the serialized function suffers form a large function call overhead if the original task is extremely fine-grained.

Inline expansion is the most popular method to alleviate a function call overhead, but the ordinary inline expansion often grows the code size exponentially when the function has more than one self-recursive call sites. For example, applying a simple inlining once to the serialized`fib_seq` shown in Fig. 3a doubles the number of the function calls because it has two self-recursive call sites. Divide-and-conquer functions tend to have more than onc self-recursions, making decision of inline expansion more difficult than normal cases in which functions are not recursive, or there appear only a single recursive call site even if they are recursive functions. We have developed a method to aggregate the recursive call sites into a single call site to completely apply inline expansion

```
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)
        *A2 = n2;
      else
        [...]
    }
    *ret = a + b;
  }
}
```

(b) Code-explosion-free inlining

```
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;
  }
}
```

(c) Task parallel vector addition

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

(d) Loopification

Fig. 3: Overview of the methods in the static cut-off [10]

to a divide-and-conquer function without incurring code-size explosion. We call this method *code-explosion-free inlining*. This code-explosion-free inlining first tries to gather recursive call sites to a single call site by inserting nested loops to point where a synchronization of target tasks is originally located, then applies normal inline expansion repeatedly; it increases the code size linearly, not exponentially. Fig. 3b presents the serialized `fib_seq` transformed with code-explosion-free inlining. It is notable that since a serialized function is only called when the Hth termination condition is satisfied, the self-recursive function calls derived from task creations can be completely eliminated from the target function after the inline expansion is applied H times.

In some cases, it is possible to transform recursive functions into functions with more natural, flat or shallowly nested loops, while code-explosion-free inlining tends to create deeply nested loops corresponding to the given height H. A divide-and-conquer vector addition task shown in Fig. 3c is a typical task which can be easily transformed into a single loop presented in Fig. 3d for humans; our compiler just tries to do that by analysis with a symbolic algebra solver. We call this optimization *loopification*. Since this loopification succeeds typically when the original program is potentially represented as a regular loop (e.g., stencil kernels or dense matrix multiplication), one might argue that such loops could have easily been manually written in a parallel loop nest in the first place. We note that, the divide-and-conquer version has an advantage of achieving the effect of cache blocking at all levels [7] with a uniform and simple code; loops, on the other hand, require cumbersome coding of explicit manual tiling to enjoy cache blocking.

These three transformations are only applicable when the static analysis of termination conditions succeeds. Our analysis fails for various reasons, though: some tasks have too complicated control flows for our current implementation of the static analysis, and others potentially hardly have explicit,

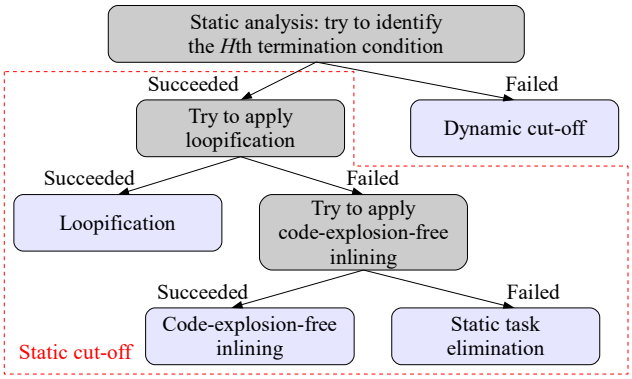


Fig. 4: Flow of optimization selection

simple termination condition (e.g., most tree traversal tasks such as Unbalanced Tree Search [14]). In order to deal with wider variety of tasks, our automatic cut-off system also adopts a dynamic cut-off strategy [4] as a fallback when the static cut-off cannot be applied. The current implementation adopts the dynamic cut-off with multiversioning proposed by the Thoman et al. [22], which is the state-of-the-art dynamic cut-off algorithm to the best of our knowledge. Their proposed algorithm can be applied without any static analysis. Thanks to the dynamic approach, our automatic cut-off system covers all tasks by employing both the static and the dynamic cut-off approach.

In summary, Fig. 4 presents the flow of the algorithm selection; the compiler first runs the static analysis to obtain a cut-off condition, and then tries to apply loopification, code-explosion-free inlining, and static task elimination in this order if the analysis succeeds; otherwise the compiler applies the dynamic cut-off.

B. Problems of Our Previous Apporach

In this section, we describe the problems of the static cut-off system we previously developed [10], which are difficult

to solve with a pure compiler approach. We will focus on the three motivating cases to see the problems concretely.

1) Inefficiency of Fallback Strategy: As shown in Fig. 4, we apply the dynamic cut-off method to tasks if the compiler failed to analyze their termination conditions. Our static analysis fails due to lack of implementation in some cases, but there are tasks which essentially do not have simple conditions near leaves of the task trees; tree traversing tasks are typical ones. For such tasks, the manual cut-off is often done with the condition in which a depth from a root task is larger than a certain threshold D. This depth-based cut-off strategy is effective when a typical structure of a task tree is known in advance; otherwise, the cut-off may significantly reduce parallelism if the depth D is too small; or just imposing an overhead for evaluating the cut-off condition if D is too large. It is often the case, however, that the tree structure depends on input of programs, which cannot be obtained at a compiling phase.

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

2) Function Size Estimation: A cut-off threshold is an important parameter to balance parallelism and sequential performance. For programs to which static task elimination or code-explosion-free inlining can be applied while loopification is not applicable, the main purpose of a cut-off is to reduce a tasking overhead. For example, if we want to keep the tasking overhead lower than 2% of the execution time, we would choose granularity at least 49 times larger than the task creation overhead. Considering that the state-of-the-art runtime systems create a task in roughly a hundred cycles, our compiler estimates the number of cycles of a function under various depths and chooses the minimum depth making the granularity larger than 5000 cycles. This strategy is effective on avoiding significant loss of parallelism, while the estimation of the number of cycles is very difficult in reality. Our static cut-off system currently estimates a cost of a function ($Cost_{function}$) by simply summing up the cost of all instructions in the target function. After the estimation of $Cost_{function}$, it determines a cut-off parameter H as the largest integer satisfying the following inequality:

$$5000 > Cost_{function} * (Child_{function})^H \qquad (1)$$

where $Child_{function}$ is a number of self-recursive calls in the function (or 2 if failed to identify the number). We use a constant parameter 4 instead of the height obtained by the estimation not to select a too small cut-off number.

Our evaluation has shown that this estimation was enough accurate to determine cut-off thresholds in most programs [10], but we have found that there was room to choose better cut-off thresholds for a few programs for performance improvement. The `fib` task shown in Fig. 1 is a good example to see the difficulty in estimating an appropriate height parameter H. Even if a number of cycles of `fib` can be estimated correctly, the right expression in (1) calculates smaller cycles than the real because the task tree of `fib` is not balanced as shown in Fig. 2. As a result, the current estimation algorithm underestimates the cut-off parameter H.

3) Cache-Aware Loop Blocking Size: For programs to which loopification is applicable, the cut-off is not only for reducing a tasking overhead, but also for simplifying control flows to improving sequential performance. For such tasks, the cut-off height parameter H is more important; a divide-and-conquer strategy for such programs also has an effect of cache blocking at all levels [7] when the programs are converted into more than one dimensional loop. If loopification is applicable, we now instead adopt the largest H whose resulting function is estimated to access less than 256KB of data, a typical L2 cache size to utilize the advantage of cache blocking and efficiency of control flows of loops.

In addition to the inaccuracy of assuming the amount of accessing data size in a loop, the best cache-blocking strategy is not obvious; we currently use L2 cache size, but L1 (or perhaps L3) cache fitting may be more effective on improving performance. Since the best cache blocking size is supposed to be different on the application features or the execution environments, the pure compiler approach can hardly solve this problem.

III. Autotuning Framework

To tackle the problems originating from the difficulty in various estimations at a compiling phase, we propose an autotuning framework for divide-and-conquer task parallel programs based on the automatic cut-off system we have proposed [10]. 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 is done by usual autotuning frameworks, it searches for the best parameters and transformation combination by executing the program on a target machine with compile options changed, and finally outputs an configuration file, which can be used to compile the program with highly-optimized options. Fig. 5 presents the flow of our autotuning framework.

This autotuning approach itself is a general one for tuning parameters. Nevertheless, we note that there has been no recent work to focus on the potentials of the simple naive task parallel programs to the best of our knowledge, which we believe to achieve high performance on multi-threaded environments if some compiler optimizations are applied properly.

A. Available Optimization Patterns

We first explain possible transformation patterns for task parallel programs with our current compilers. One of the notable divide-and-conquer features is that a parent divide-and-conquer function and its child functions are not necessarily the same program if both the parent and its children correctly work; the parent divides a problem into subproblems,

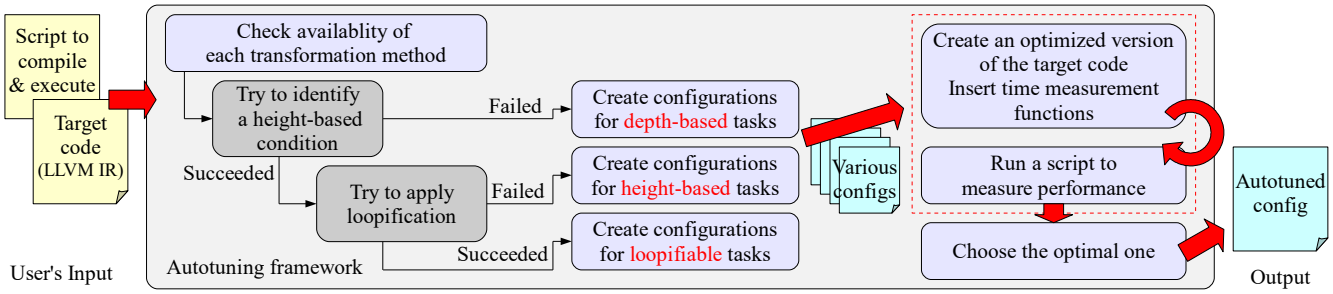


Fig. 5: Overview of our autotuning framework

and the children conquer the subproblems. PetaBricks [1] is the most famous autotuning framework focusing on this feature, but it is designed to require users to write multiple versions of the divide-and-conquer programs. Our compiler can create multiple functions derived from an original task function by applying optimizations with some parameters, removing task creations, and/or inserting a conditional branch to other function transformed in different ways. As well as other usual autotuning frameworks, our framework searches for the best combination and connection of various divide-and-conquer functions.

There are two fundamental elements to optimize each multiversioned task:

1. An optimization method to the task. If an optimization method takes parameters, they also become optimization targets (e.g., unrolling times of inline expansion).
2. A condition to branch other function based on either depth from the root task or height from the leaves in addition to a function to which the task branches.

Fig. 6 describes an example of one possible optimization pattern for `fib`. It is based on the following strategy. Tasks in the upper side of the task tree shown in Fig. 6a must be run in parallel to have other cores work sooner. To reduce a tasking overhead, tasks satisfying an Hth termination condition, say 10 as H, are serialized. Furthermore, to reduce a function call overhead, this strategy also applies a simple inline-expansion twice to the serialized tasks under the 2nd termination condition, and then eliminates recursive calls in that function because this function is only executed in the 2nd termination condition. Fig. 6b shows a configuration of the optimization strategy above and Fig. 6c presents a transformed code with the configuration.

1) Optimization Methods: Our compiler employs three basic transformations: normal inlining, code-explosion-free inlining, and loopification. Derived from these three, six versions of transformations are implemented. Three transformations remain a task recursive, so the optimized task can be used as either edges or leaves of the task trees. The other three transform it into a function which can be used as only leaves in the trees because the task is transformed to have no recursive calls. Fig. 7 shows the six divide-and-conquer vector addition tasks generated with the respective methods. As shown in the figure, tasks shown in (a), (b), and (c) can be used as both edges or leaves, whereas (d), (e), and (f) are no longer self-recursive functions.

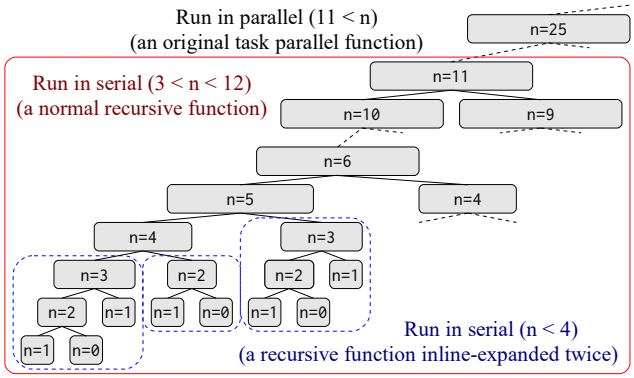
We note that our autotuning framework is capable of adopting other optimization methods than inlining methods or loopfication due to the divide-and-conquer algorithm feature as we previously discussed.

2) Conditional Branch to Other Task: A conditional branch to other task is a key to combining multiversioned tasks. Considering a branch from a task parallel function to a serialized function for example, the concept of conditional branch includes the idea of cut-off. Determining a branch condition has therefore a large impact on balancing concurrency and sequential performance. Two parameters compose the branch condition: depth from the root and height from the leaves. As we discussed, a height-based condition is suitable for a cut-off, but static analysis is necessary to identify the height-based condition. On the other hand, the depth-based one can be applied to any tasks because the depth can be easily obtained by incrementing depth from the root task, or using a task parallel runtime. A compiler can use either of them, or in theory, combining them to construct the condition if both available (e.g., a condition in which height is less than 8 or depth is more than 10).

B. Searching Space

Since transformed tasks can be connected to any other task, the possible patterns are countless. We therefore search a limited space including reasonable transformation patterns. Let us assume there is only one task to optimize. Our proposed system classifies a 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 to apply further optimizations, while the tasks near the root should be original not to decrease the concurrency. We respectively explain the three patterns.

1) Depth-based Cut-off: Our system applies a depth-based cut-off to a task in which the static analysis fails. In our previous static cut-off approach, the dynamic cut-off is applied to such a task. We found, however, that it could not effectively improve performance. Our autotuning framework can choose an appropriate depth; it can implicitly utilize information of the structure of the task tree by executing a program indeed, whereas the pure compiler approach cannot acquire



(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. 6: Example of a possible optimization pattern

such information. The depth parameter D is determined as a depth which is the largest depth to achieve 99% of the optimal performance; the 99% condition is imposed to avoid unnecessary reduction of parallelism.

We cannot eliminate self-recursive function calls because the exact termination condition cannot be obtained for such tasks. Tasks after a cut-off must be, hence, functions which can be used as edges ((a) to (c) in Fig. 7). Thus the serialized function can be optimized either by simple inlining, or code-explosion-free inlining.

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 the height-based cut-off condition for such a task since the height-based condition guarantees the size of each serialized

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

(a) No optimization

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

(b) Inline-expansion (once)

```
void vecadd(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(i){
          case 0: a3=a2; b3=b2; n3=n2/2; break;
          case 1: a3=a2+n2/2; b3=b2+n2/2;
                  n3=n2-n2/2;
          }
          vecadd(a3,b3,n3);
}}}}}
```

(c) Code-explosion-free inlining (once)

```
//assume 1<=n && n<=2
void vecadd(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 1st termination condition)

```
//assume 1<=n && n<=2
void vecadd(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-explosion-free inlining (in 1st termination condition)

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

(f) Loopification (in 1st termination condition)

Fig. 7: Six transformation methods

task to some extent regardless of an input of programs. It is a height used as the cut-off parameter H that is the smallest height to achieve 99% performance compared to the best one, as well as the depth-based cut-off does.

Optimizations based on inlining can be applied to the serialized function under a certain termination condition. We consider six patterns as follows. The first pattern attempts to optimize tasks in a common way; simply inline-expands a function once or multiple times. The second pattern applies code-explosion-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 pattern adopt *complete* inline expansion and *complete* code-explosion-free inlining, shown in Fig. 7d and Fig. 7e. They introduce simple recursive functions to connect the original task to the leaf function, 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-explosion-free inlining to reduce a function call overhead.

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 significantly affects performance of a loopified function because the cut-off in practice changes a size of the cache blocking. Considering the differences from the previous case, our system employs special autotuning method for loopifiable tasks. It uses the smallest height with which the loopified program can achieve more than 99% performance compared to the best performance as a height parameter H. After that, we change the loopification condition with the cut-off threshold fixed in order to enjoy the best cache-blocking effect.

C. Preprocessing for Autotuning

Our autotuning framework first checks the availability of each transformation method for a task because not all the transformations are applicable to every task. If some transformations are found to be inapplicable to the target task by trying to apply them in practice, our framework skips patterns including them. It is also required to add additional instructions for the performance measurement; the system automatically creates a function as an interface for external calls and inserts performance measurement functions into the function, which lowers the programmers' loads.

D. Autotuning Results

After executing programs with various optimization patterns, our autotuning system writes down the best optimization pattern to the configuration file. This file can be used for an input of our compiler to optimize a program with the best parameters. It is written in a human-readable format, which 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 [11]. The interface of our autotuning framework was written in Python. Task parallel programs are run on MassiveThreads [13], a lightweight task library adopting a child-first scheduling strategy [12].

For the evaluation, we prepared eleven benchmarks containing a task function. The benchmarks are as follows:

1) **fib** calculates a 45th Fibonacci number with the naive double recursion.
2) **nqueens** counts solutions of the N-Queens problem, with $N = 14$. Both of our **fib** and **nqueens** adopt the same task creation patterns in benchmarks of BOTS [5].
3) **nbody** directly calculates forces between all N to N pairs. The input array consists of $30K$ particles with its mass, position, velocity, and a temporal variable to accumulate the force. Positions, velocities, and forces are 3D vectors.
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 3-dimensional version of heat2d, on a $1K \times 1K \times 1K$ mesh.
7) **gaussian** is an image processing kernel that 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 (floating point numbers) at the leaves. The input is a balanced tree with the height of 30 with $2^{30} - 1$ elements.
10) **treesum** receives a pointer-based binary tree and sums up the values (floating point numbers) in the leaves by tree traversal. The input is a balanced tree with the height of 30 (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.

To examine the performance compared to those of loop parallel programs shown in Figure 10, we instead used benchmarks with larger data presented in Table I to make a single execution time longer than 0.2 seconds.

TABLE I: Data size for an evaluation shown in Figure 10

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

All of the benchmarks are originally written in C language, so we first transformed them into LLVM intermediate representations with Clang, a frontend C/C++ compiler of LLVM, and then input the LLVM-IR into the 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.

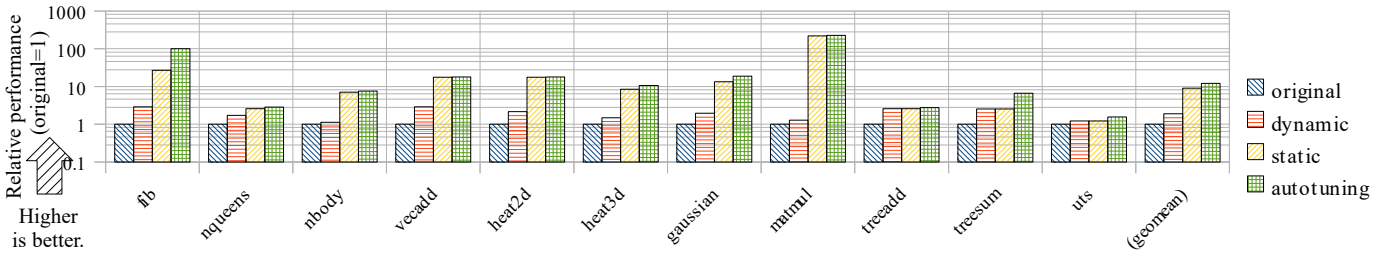


Fig. 8: Multi-threaded performance. It shows relative performance of the original (original), the dynamic cut-off [22] (dynamic), our static cut-off [10] (static), and the autotuning we propose in this paper (autotuning)

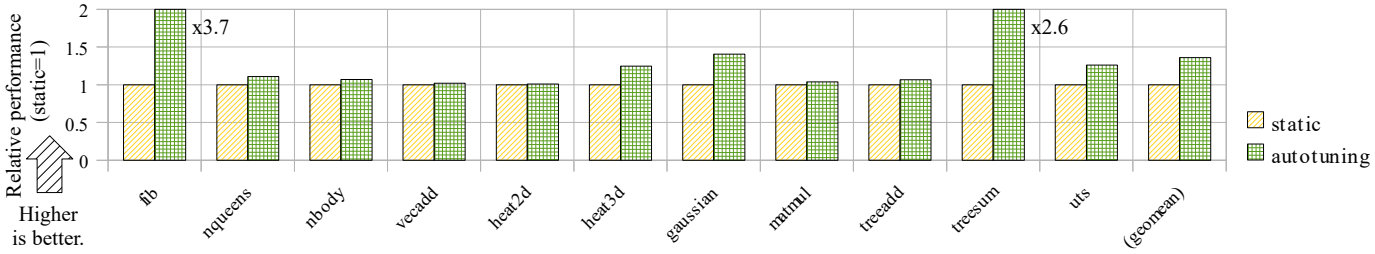


Fig. 9: This shows the same result of Fig. 8, while the baseline is performance of the static cut-off.

Note that we applied no manual cut-off to them.

Table II shows which optimizations were applicable to these eleven benchmarks.

TABLE II: Applicability of cut-off methods

(code-explosion-free inlining is abbreviated to "cef" in this table)

	dynamic	static	cef (serial)	cef (task)	loopification
fib	✓	✓	✓	✓	
nqueens	✓	✓	✓	✓	
nbody	✓	✓	✓		
vecadd	✓	✓	✓	✓	✓
heat2d	✓	✓	✓	✓	✓
heat3d	✓	✓	✓	✓	✓
gaussian	✓	✓	✓	✓	✓
matmul	✓	✓	✓		✓
treeadd	✓				
treesum	✓				
uts	✓				

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 following charts show the averages of five measurements.

A. Performance compared to task parallel programs

Fig. 8 shows the improvement of performance using 36 threads. The baseline is a task parallel program with no cut-off (**original**). We measured performance of the original programs and the dynamic cut-off to show the efficacy of our optimizations. Our autotuning framework took 320 seconds per task on average to generate the optimal optimization pattern. As shown in Fig. 8, our optimization achieved a significant speedup in comparison to the original one (**original**) and tasks optimized by the dynamic cut-off; our static cut-off achieved from 1.2x to 220x (geometric mean of 9.0x) and our autotuning framework did from 1.5x to 228x (geometric mean of 12.2x), while the dynamic cut-off elevated performance only from 1.1x to 2.9x (geometric mean of 1.9x).

To see the detail of performance difference between the static cut-off and the autotuning, Fig. 9 focuses on **static** and **autotuning**. It shows that our autotuning framework enhanced the performance higher than the static cut-off: achieved a speedup of geometric mean of 1.3x. Considering the mechanism of our autotuning, this result itself was not very surprising. The result indicated room for optimization exist especially for **fib** and **gaussian**, and tasks which our static analysis fails: **treesum**, **uts**. The configuration files our autotuning framework output provide some insights as follows:

1. **fib** is a case in which the cut-off height selected in the static cut-off is much smaller.
2. For **gaussian**, a larger loop is preferable as its leaf calculation; L2 cache blocking is not the optimal strategy for **gaussian**.
3. **treesum** and **uts** do not fully perform with a dynamic cut-off: cut-off methods should be improved to elevate performance of such tasks.

The results overall demonstrated the efficacy of the autotuning framework we propose, in addition to revealing some insights for further improvement of our static cut-off system.

B. Performance compared to loop parallel programs

For programs naturally expressible in loops, we manually loop-parallelized them and compared them with optimized task parallel programs to examine the absolute performance achieved by the autotuning framework. We loop-parallelized **nbody**, **vecadd**, **heat2d**, **heat3d**, **gaussian**, and **matmul**. They performed the same calculations with the corresponding task parallel programs, with the only

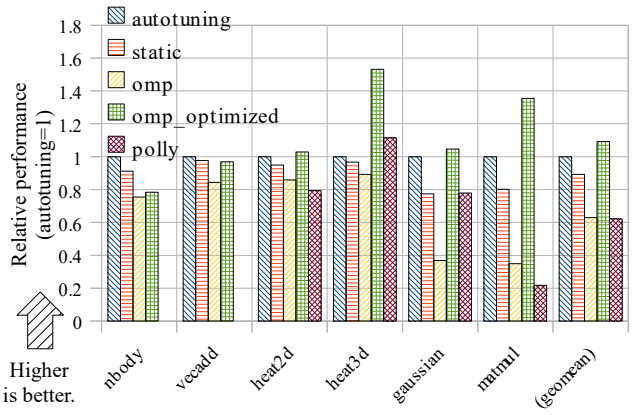


Fig. 10: Performance comparison of autotuned task parallel programs to loop parallel programs. The baseline is task parallel programs optimized by our autotuning framework.

differences in load partitioning and execution order. The loop programs were parallelized in two ways. 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 [8], a locality-optimizer for LLVM with an automatic parallelization feature. The Polly version was compiled by the latest Clang with `-O3 -ffp-contract=fast`, and Polly with `-polly -polly-parallel` and `-polly-vectorizer` = `stripmine` if it made the program faster.

Figure 10 shows the comparison of optimized task parallel programs to loop parallel programs, using the task parallel version as a baseline. In the figure, **autotuning** is performance of task parallel programs optimized by our autotuning framework, and **static** is one of the static cut-off. **omp** and **polly** are those of simple loop parallel programs optimized by OpenMP with GCC, or by Polly with Clang respectively. Both of them used default scheduling algorithm. We also made **omp_optimized**, a manually-tuned OpenMP programs by changing block sizes, scheduling strategies, scheduling parameters (chunk size, etc.), and collapse clauses for nested loops. Since we found that **nbody** and **vecadd** were not parallelized by **polly**, these results are missing in the chart. Figure 10 shows the performance of **autotuning** was faster than **static**, **omp** and **polly**; the geometric mean of **static**'s relative performance was 0.89x, those of **omp** and **polly** were 0.63x and 0.62x respectively. 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 in some cases by utilizing advantanges 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 [23] and FFTW [6] are the most famous autotuning software to highly optimize specific programs. Ansel et al. [1] proposed PetaBricks, which is more general autotuning system containing original language and its compiler. PetaBricks mainly focused on tuning algorithmic choices, by utilizing a divide-and-conquer feature. The basic concept is very similar to that of our proposed system. However, our goal is to achieve high performance with very simple task parallel programs basically written in C/C++, while PetaBricks requires programmers to write multiple implementations of the algorithms in their original language.

B. Automatic cut-off

There have been several studies on automating cut-off [4], [22]. Duran et al. [4] proposed a heuristic cut-off based on the runtime information such as depth of the task and the number of ready tasks. Thoman et al. [22] proposed generating multiple versions including inlined versions and fully serialized version, and switching between them based on a similar criteria. These proposals, or runtime-based approaches in general, have advantages of being widely applicable and not requiring substantial compiler development efforts. However, the dynamic cut-off reveals little opportunity for applying aggressive optimizations; evaluations shows our static cut-off, not to say our autotuning framework, achieved higher performance than that of the dynamic cut-off obtained.

C. Optimization for recursive programs

To optimize recursive programs, a number of studies have sought effective expansion techniques. Tail call elimination [20] is a traditional technique to transform recursive procedures into jumps (a control flow without stacks). It can handle only tail-recursive calls. Particularly, it cannot handle most divide-and-conquer algorithms, which perform two or more recursive calls one of which is necessarily not tail-recursive. Tang [21] proposed *complete inlining*, which can be thought of a generalization of tail call elimination. It recognizes, more broadly, recursive calls that can be transformed into a jump. The recursive call does not have to be tail-recursive, yet the technique still cannot handle procedures that perform recursive calls twice or more.

There have been several methods primarily targeting divide-and-conquer algorithms [19], [9]. Rugina et al. proposed function unrolling and rerolling, which together transform a recursive procedure to another recursive procedure. The success of the transformation relies on "conditional fusions," which essentially require the depths of the two recursions be identical. This condition seems very restrictive; most programs do not satisfy it. Herrman and Lengauer [9] have shown transformation of divide-and-conquer Haskell programs to parallel loop nest in C. It assumes Haskell programs written in a specific skeleton (template) and vectorizes them. The vectorizable skeleton essentially assumes that the recursion stops at the same depth everywhere, which seems restrictive for practical purposes.

Vectorization for recursive functions has been eagerly studied recently. Auto-vectorization on recursive functions written

in Haskell has been proposed by Petersen et al. [16], which focuses on a simple recursive function, but a divide-and-conquer function with multiple recursive call sites. Ren et al. [18] proposed a general vectorization technique for divide-and-conquer programs. In their framework, a compiler assigns multiple tasks created by a single task to different SIMD lanes. This strategy is applied to the root of the task tree, effectively executing subtrees near the root of the task tree in different SIMD lanes. Their vectorization technique is a novel transformation method; our autotuning framework is capable of adopting any optimization techniques for divide-and-conquer programs.

VI. Conclusion

In this paper, we describe 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 illustrates significant speedup by our proposed framework, which is faster than the static cut-off, and comparable to that of manually-optimized loop parallel programs.

For future work, we first want to improve an autotuning framework to allow programs including more than one tasks. Since allowing multiple tasks enlarges a search space, more sophisticated searching method should be required for shorter autotuning time. Addressing an input sensitivity problem is also our future work. In the field of autotuning, input sensitivity (e.g., [3]) has been a challenging topic. The input-sensitivity problem in our context is for example that of determining an optimal depth-based cut-off threshold for a task program whose structure of task tree is varied. Our proposal works well for a similar pattern of input, whereas the autotuned program may be not efficient for another type of input. Concurrently, we will improve our static cut-off system by utilizing insights obtained by this autotuning framework.

References

[1] 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, pages 38–49, New York, NY, USA, 2009. ACM.

[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 Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '95, pages 207–216, New York, NY, USA, 1995.

[3] Y. Ding, J. Ansel, K. Veeramachaneni, X. Shen, U.-M. O'Reilly, and S. Amarasinghe. Autotuning algorithmic choice for input sensitivity. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 379–390, New York, NY, USA, 2015. ACM.

[4] 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, pages 36:1–36:11, Piscataway, NJ, USA, 2008.

[5] 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, pages 124–131, Washington, DC, USA, 2009. IEEE Computer Society.

[6] M. Frigo and S. G. Johnson. FFTW: an adaptive software architecture for the FFT. In *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, volume 3, pages 1381–1384 vol.3, May 1998.

[7] 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, pages 285–, Washington, DC, USA, 1999. IEEE Computer Society.

[8] T. Grosser, H. Zheng, R. Aloor, A. Simbürger, A. Größlinger, and L.-N. Pouchet. Polly - polyhedral optimization in LLVM. In *Proceedings of the First International Workshop on Polyhedral Compilation Techniques*, IMPACT '11, 2011.

[9] C. A. Herrmann and C. Lengauer. Transformation of divide & conquer to nested parallel loops. In *Proceedings of the 9th International Symposium on Programming Languages: Implementations, Logics, and Programs: Including a Special Trach on Declarative Programming Languages in Education*, PLILP '97, pages 95–109, London, UK, UK, 1997. Springer-Verlag.

[10] S. Iwasaki and K. Taura. Effective decision of cut-off threshold for task parallel programs. In *Summer United Workshops on Parallel, Distributed and Cooperative Processing 2015*, SWoPP '15, 2015.

[11] 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, pages 75–, Washington, DC, USA, 2004.

[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, pages 185–197, New York, NY, USA, 1990. ACM.

[13] 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*, 2012.

[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, pages 235–250, 2007.

[15] OpenMP Architecture Review Board. OpenMP Application Program Interface Version 3.0. Technical Report May, 2008.

[16] L. Petersen, D. Orchard, and N. Glew. Automatic SIMD vectorization for Haskell. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 25–36, New York, NY, USA, 2013. ACM.

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

[18] B. Ren, Y. Jo, S. Krishnamoorthy, K. Agrawal, and M. Kulkarni. Efficient execution of recursive programs on commodity vector hardware. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 509–520, New York, NY, USA, 2015. ACM.

[19] R. Rugina and M. C. Rinard. Recursion unrolling for divide and conquer programs. In *Proceedings of the 13th International Workshop on Languages and Compilers for Parallel Computing-Revised Papers*, LCPC '00, pages 34–48, London, UK, UK, 2001.

[20] G. L. Steele, Jr. Debunking the "expensive procedure call"; myth or, procedure call implementations considered harmful or, lambda: The ultimate goto. In *Proceedings of the 1977 Annual Conference*, ACM '77, pages 153–162, New York, NY, USA, 1977. ACM.

[21] P. Tang. Complete inlining of recursive calls: Beyond tail-recursion elimination. In *Proceedings of the 44th Annual Southeast Regional Conference*, ACM-SE 44, pages 579–584, New York, NY, USA, 2006.

[22] 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, pages 164–177, Berlin, Heidelberg, 2013.

[23] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, SC '98, pages 1–27, Washington, DC, USA, 1998. IEEE Computer Society.