The best multicore-parallelization refactoring you've never heard of*

Mike Rainey
Carnegie Mellon University
Pittsburgh, PA, USA
me@mike-rainey.site

Abstract

In this short paper, we explore a new way to refactor a simple but tricky-to-parallelize tree-traversal algorithm to harness multicore parallelism. Crucially, the refactoring draws from some classic techniques from programming-languages research, such as the continuation-passing-style transform and defunctionalization. The algorithm we consider faces a particularly acute granularity-control challenge, owing to the wide range of inputs it has to deal with. Our solution achieves efficiency from heartbeat scheduling, a recent approach to automatic granularity control. We present our solution in a series of individually simple refactoring steps, starting from a high-level, recursive specification of the algorithm. As such, our approach may prove useful as a teaching tool, and perhaps be used for one-off parallelizations, as the technique requires no special compiler support.

1 Challenge: traverse a pointer-based tree

We are to write a program that traverses a given binary tree and returns the sum of the numbers stored in the nodes, as exemplified by the following reference code, taking care to utilize parallelism when the input permits and to perform well in any case, even when parallelism is limited.

```
type node = {v : int, bs : node*[2]}
sum(node* n) → int { if (n == null) return 0
  return sum(n.bs[0]) + sum(n.bs[1]) + n.v }
```

We may use fork join to parallelize on nonempty input trees:

```
s = new int[2]
fork2join(λ () ⇒ {s[i] = sum(n.bs[i])}, ...) i ∈ {0,1}
return s[0] + s[1] + n.v
```

There are mature implementations of fork join (e.g. [7, 12]) based on work stealing [4–6], an algorithm well suited for irregular, input-dependent workloads like ours.

However, suppose our program can assume nothing of its input: it can range from balanced and large, where parallelism is abundant, to, e.g., a chain or a small tree, where traversal is mostly serial. As such, possible workloads may be fine grain and irregular, making granularity control acute [1, 3, 13]. Also, inputs, e.g., long chains, may cause callstack overflow [9]. We address both problems by combining two known techniques: (1) heartbeat scheduling [2, 10] for granularity control and (2) defunctionalization of continuation-passing style [11] to replace recursion by iteration, for efficient, serial traversal. The latter is inspired by Koppel's presentation [8], adding to his list a new refactoring application: multicore parallelization. We proceed via a series of refactorings, using conventional features of C++.

*Our title is a riff on "The Best Refactoring You've Never Heard Of" [8], from the popular blog post and Compose 2019 talk.

2 Refactoring for parallel traversal

First, we replace the direct-style treatment that allocates function-activation records on the C callstack with a continuation-passing-style (CPS) one that allocates on the heap. To support CPS, we use a lower-level interface with task scheduling: `new_task(`f`)` takes a thunk f and returns a pointer to a new, heap-allocated task that, when executed by the scheduler will run $f()$ to completion; `fork(`c`, `k`)` takes a child task c, its continuation task k, registers a dependency edge from c to k, and marks c ready to run; `join(`j`)` marks an incoming dependency edge on the task j as resolved, and, when all of its dependencies are resolved, schedules j.

Step 1: CPS convert the parallel algorithm. We introduce a continuation parameter `k` and a *join task* `tj`, which receives the results of the recursive calls and passes the results to the return continuation.

```
sum(node* n, k : int → void) → void {
  if (n == null) { k(0); return }; s = new int[2]
  tj = new_task(λ () ⇒ k(s[0] + s[1] + n.v))
  { t_i = new_task(λ () ⇒
       sum(n.bs[i], λ s_i ⇒ {s[i] = s_i; join(tj)})
    fork(t_i, tj) } i ∈ {0,1} }
```

Step 2: defunctionalization of CPS. We introduce one activation record to handle the final result, and another for completion of branch `i` $\in \{0, 1\}$ (full code in appendix).

```
type kont = | KTerm of int* // final result
  | KPBranch of {i : int, s : int*, tj : task*}
```

This refactoring delivers a highly parallel algorithm, but one with poor work efficiency, given that it performs little useful work per task.

3 Refactoring for serial traversal

Now, we obtain a work-efficient version by replacing recursion with iteration.

Step 3: CPS convert & defunctionalization of CPS. We CPS convert our reference algorithm, first by introducing two new continuations, and then defunctionalize them, giving us two new activation records, such that the first represents an in-flight recursive call for the first branch of a node, and the second for the second branch, with s_0 storing result obtained for the first branch.

```
type kont = ... | KSBranch0 of {n : node*, k : kont*}
  | KSBranch1 of {s0 : int, n : node*, k : kont*}
```

Step 4: refactor for iterative, stack-based traversal. We eliminate recursion by applying to the **apply** function (introduced in Step 3) both tail-call elimination and inlining, and tail-call elimination to our defunctionalized `sum` function.

4 Merging parallel and serial refactorings

The conceptual glue for merging our serial and parallel algorithms is in heartbeat scheduling. With it, we make it so that our serial and parallel traversals alternate on a regular basis. Starting out, our program spends a certain amount of its time in serial traversal, specified by a heartbeat-rate parameter H, after which it switches momentarily to parallel traversal. It then switches back to serial, and the alternation repeats until the traversal completes.

By ensuring H serial traversal steps happen for each invocation of our parallel traversal, we amortize task-creation costs, and therefore, achieve granularity control for *all* inputs. On our test machine, we observed that, by experimenting with different settings of H, we can bound task-creation costs such that the total amount of work is increased by a desired amount, e.g., 10%, compared to the serial refactoring.

Step 5: give the serial traversal a heartbeat. To track the number of steps, we introduce a helper function `heartbeat` that returns `true` every H times it is called, and we insert calls to `heartbeat` in each of the two main loops of the serial traversal. When it returns `true`, we inspect the current continuation to see if it is holding onto any latent parallelism. If so, we *promote* that latent parallelism into an actual task, which may realize actual parallelism (if, e.g., the task is stolen).

Step 6: implement promotion. Promotion is initiated by calling `try_promote(`k`)`, which looks for latent parallelism in k and, if present, spawns from it a task and returns a modified continuation k'. There is latent parallelism in k if there is an instance of $\texttt{KSBranch}_0$ in k. The reason is that such an instance represents a recursive call to the first branch of some tree node (the only opportunity parallelism in a traversal).

However, there may be multiple instances of latent parallelism in a given k, and, for performance reasons, heartbeat scheduling requires that the *outermost* instance is the one that should be targeted for promotion. Heartbeat scheduling targets outermost parallelism because doing so turns out to be crucial for achieving worst-case bounds on the loss of parallelism [2]. Implementing this behavior efficiently requires some care, as a naïve implementation could repeatedly traverse the whole stack, leading to quadratic blowup. Fortunately, the blowup can be remedied by extending the continuation structure with a double-ended list, which marks promotion potentials [2, 10].

When it finds a $\texttt{KSBranch}_0\texttt{\{n, k=}k'\texttt{\}}$ activation record in k, our promotion handler modifies k so that, thereafter, it is as if our (defunctionalized) parallel algorithm was invoked at that point instead of the serial version. This behavior is achieved by (1) allocating storage for the results of the branches, `s = new int[2]` (2) replacing our $\texttt{KSBranch}_0$ activation record with `KPBranch{i=0, s=s, tj=tj}`, a task-parallel one (3) spawning a new task corresponding to the second branch, i.e., `n.bs[1]`, and giving that task the return continuation `KPBranch{i=1, s=s, tj=tj}`, and (4) creating a join task `tj` for this new fork point, which is seeded with the continuation of our promotion point, k'. The pseudocode below gives sketch of the main loops.

```
sum(node* n, k : kont*) → void {
  while (true)
    k = try_promote(k) if heartbeat() else k
    if (n == null) { sa = 0 // sum accumulator
      while (true)
        k = try_promote(k) if heartbeat() else k
        match *k with // all activation records in kont
        | KSBranch0{n=n1, k=k1} ⇒ { ... } | ... | ...
    else { k = KSBranch0{n=n, k=k}; n = n.bs[0] } }
```

5 Performance study

Table 1 summarizes our performance study, for which we used a C++ implementation. From the *perfect* tree, we see that our algorithm can achieve a speedup comparable to that of OpenCilk [12] with manually tuned granularity control, and a speedup almost twice faster than that of OpenCilk without granularity control. For *random*, our algorithm outperforms vanilla OpenCilk, but not the granularity-controlled version.

input	serial (s)	ours	cilk	cilk+granctrl
perfect	0.7	28.4x	15.4x	34.5x
random	0.8	31.8x	15.3x	33.7x
chains	2.5	11.5x	n/a	n/a
chain	1.2	0.4x	n/a	n/a

Table 1. Performance results from an Intel Xeon system, using all 64 cores, showing speedup over the iterative, serial algorithm, with four inputs: (1) *perfect* is a perfect binary tree of height 27 (2) *random* is a tree built from a series of path-copying insertions targeting random leaves (3) *chains* is a small initial tree of height 20 extended with 30 paths of length 1 million (4) *chain* is a long chain.

The reason relates to the data structure we used in our C++ implementation to store the activation records, an STL deque, which uses heap-allocated chunks internally, whereas OpenCilk uses the callstack, which is more efficient. However, our algorithm supports long chains, whereas OpenCilk crashes with stack overflow (indicated by cells with n/a). From *chains*, we see that our algorithm can obtain speedup even when parallelism is somewhat scarce. On *chain*, our algorithm is about 2.5x slower serial.

References

[1] U. A. Acar, V. Aksenov, A. Charguéraud, and M. Rainey. Provably and practically efficient granularity control. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, pages 214–228, New York, NY, USA, 2019. ACM.

[2] U. A. Acar, A. Charguéraud, A. Guatto, M. Rainey, and F. Sieczkowski. Heartbeat scheduling: Provable efficiency for nested parallelism. In *39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '18. ACM, 2018.

[3] U. A. Acar, A. Charguéraud, and M. Rainey. A work-efficient algorithm for parallel unordered depth-first search. In *ACM/IEEE Conference on High Performance Computing (SC)*, pages 67:1–67:12, New York, NY, USA, 2015. ACM.

[4] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46:720–748, Sept. 1999.

[5] D. Chase and Y. Lev. Dynamic circular work-stealing deque. In *SPAA '05*, pages 21–28, 2005.

[6] R. H. Halstead, Jr. Implementation of Multilisp: Lisp on a Multiprocessor. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, LFP '84, pages 9–17. ACM, 1984.

[7] Intel. Intel threading building blocks, 2011. https://www.threadingbuildingblocks.org/.

[8] J. Koppel. The best refactoring you've never heard of, 2019. https://www.pathsensitive.com/2019/07/the-best-refactoring-youve-never-heard.html.

[9] I.-T. A. Lee, S. Boyd-Wickizer, Z. Huang, and C. E. Leiserson. Using memory mapping to support cactus stacks in work-stealing runtime systems. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 411–420, New York, NY, USA, 2010. ACM.

[10] M. Rainey, K. Hale, R. R. Newton, N. Hardavellas, S. Campanoni, P. Dinda, and U. A. Acar. Task parallel assembly language for uncompromising parallelism. In *Proceedings of the 42nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '21, New York, NY, USA, June 2021. ACM.

[11] J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference - Volume 2*, ACM '72, page 717–740, New York, NY, USA, 1972. Association for Computing Machinery.

[12] T. B. Schardl, W. S. Moses, and C. E. Leiserson. Tapir: Embedding fork-join parallelism into llvm's intermediate representation. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 249–265, 2017.

[13] A. Tzannes, G. C. Caragea, R. Barua, and U. Vishkin. Lazy binary-splitting: a run-time adaptive work-stealing scheduler. In *Symposium on Principles & Practice of Parallel Programming*, pages 179–190, 2010.

```
sum(node* n, k : kont*) → void {
  if (n == null) { apply(k, 0); return }
  s = new int[2]
  tj = new_task(λ () ⇒
                 apply(k, s[0] + s[1] + n.v))
  { t_i = new_task(λ () ⇒
            sum(n.bs[i], KPBranch{i=i, s=s, tj=tj))
    fork(t_i, tj) } i ∈ {0, 1}  }

apply(kont* k, s_a : int) → void {
  match *k with
  | KPBranch{i, s, tj} ⇒ {s[i] = s_a; join(tj)}
  | KTerm ans ⇒ {*ans = s_a} }
```

Figure 1. Step 2: defunctionalize the continuation

```
sum(node* n, k : int → void) → void {
  if (n == null) { k(0); return }
  sum(n.bs[0], λ s_0 ⇒
    sum(n.bs[1], λ s_1 ⇒
      k(s_0 + s_1 + n.v))) }
```

Figure 2. Step 3(a): CPS convert

A Reference code

In this section, we present the pseudocode omitted from the main body of the paper, starting with the parallel algorithm, then the serial one, and finally the heartbeat algorithm.

A.1 The parallel traversal

Step 2: defunctionalization of CPS. In Figure 1, we show the code resulting from taking the CPS converted version of our parallel algorithm and defunctionalization.

A.2 The serial traversal

Step 3(a): CPS convert. Starting back from our our reference algorithm, we now begin the process of turning that code into a iterative, stack-based one. The algorithm shown in Figure 2 is the result of converting our original recursive algorithm to continuation-passing style.

Step 3(b): defunctionalization of CPS. In this step, we defunctionalize the three continuations we introduced in the previous step, giving us the code shown in Figure 3. The first such continuation is the final continuation, `KTerm`, the continuation that receives the final result of our `sum` function. The second, namely $\texttt{KSBranch}_0$, is the continuation that receives the result of the first recursive call, and the third, namely $\texttt{KSBranch}_1$, is the continuation that receives the final result of the `sum` call, that is, $\texttt{s}_0 + \texttt{s}_1 + \texttt{n.v}$.

Step 4(a): tail-call eliminate `apply`**.** In this step, we get rid of the recursion in the `apply` function from our previous step by turning it into a loop. We do so by applying tail-call elimination, giving us the code shown in Figure 4.

```
sum(node* n, k : kont*) → void {
  if (n == null) { apply(k, 0); return }
  sum(n.bs[0], KSBranch_0{n=n, k=k})

apply(kont* k, s_a : int) → void {
  match *k with
  | KSBranch_0{n, k=k1} ⇒ {
    sum(n.bs[1], KSBranch_1{s_0=s_a, n=n, k=k1}) }
  | KSBranch_1{s_0, n, k=k1} ⇒ {
    apply(k1, s_0 + s_a + n.v) }
  | KTerm ans ⇒ {*ans = s_a} }
```

Figure 3. Step 3(b): defunctionalization of CPS

```
apply(kont* k, s_a : int) → void {
  while (true)
    match *k with
    | KSBranch_0{n, k=k1} ⇒ {
      sum(n.bs[1], KSBranch_1{s_0=s_a, n=n, k=k1})
      return }
    | KSBranch_1{s_0, n, k=k1} ⇒ {
      s_a = s_0 + s_a + n.v; k = k1 }
    | KTerm ans ⇒ {*ans = s_a; return } }
```

Figure 4. Step 4(a): tail-call eliminate `apply`

```
sum(node* n, k : kont*) → void {
  if (n == null)
    while (true)
      s_a = 0
      match *k with
      | KSBranch_0{n, k=k1} ⇒ {
        sum(n.bs[1], KSBranch_1{s_0=s_a, n=n, k=k1})
        return }
      | KSBranch_1{s_0, n, k=k1} ⇒ {
        s_a = s_0 + s_a + n.v; k = k1 }
     | KTerm ans ⇒ {*ans = s_a; return }
    return
  sum(n.bs[0], KSBranch_0{n=n, k=k})
```

Figure 5. Step 4(b): inline `apply`

Step 4(b): inline `apply`**.** Now, we inline the `apply` function from the previous step into the body of our `sum` function, giving us the code in Figure 5.

Step 4(c): tail-call eliminate `sum`**.** Here, we get rid of the recursion in the `sum` function from our previous step. To this

```
sum(node* n, k : kont*) → void {
  while (true)
    if (n == null)
      while (true)
        s_a = 0
        match *k with
        | KSBranch_0{n=n1, k=k1} ⇒ {
          n = n1.bs[1]
          k = KSBranch_1{s_0=s_a, n=n1, k=k1}
          break }
        | KSBranch_1{s_0, n, k=k1} ⇒ {
          s_a = s_0 + s_a + n.v; k = k1 }
        | KTerm ans ⇒ {*ans = s_a; return }
    else
      k = KSBranch_0{n=n, k=k}
      n = n.bs[0] }
```

Figure 6. Step 4(c): tail-call eliminate sum

end, we apply tail-call elimination, giving us the code shown in Figure 6.

A.3 The heartbeat traversal

Step 5: give the traversal a heartbeat. Now, we are going to merge the final versions of the parallel and serial algorithms we obtained in the previous steps. The result of our merging is shown in Figure 7. In particular, it is the merging of our defunctionalized parallel algorithm from Figure 1 and the serial algorithm in Figure 6. For our merging, we introduce two calls to `try_promote`, which have the effect of controlling the switching between serial and parallel traversals. The switching is enabled by simply merging the `match` expressions of the serial and parallel versions.

Step 6: implement promotion. All that remains of our implementation is the code that handles promotions, which we show in Figure 8. In this function, we search in our input continuation `k` for an instance of the activation record $\texttt{KSBranch}_0$, which represents the continuation waiting for the result of the first branch. If this search fails, then it returns a `null` pointer value in `kt`, and we exit early by returning the original continuation `k`. Otherwise, a sucessful search means that we have captured an instant in our serial traversal when it is in the middle of traversing the first branch of a tree node. For such a case, we can parallelize the in-flight traversal of that first branch with the yet-to-start traversal of the corresponding right branch.

The rest of this function spawns a new task for the right branch following the pattern in the code shown in Figure 1. It uses the `replace` function to rewrite the input continuation in place such that the $\texttt{KSBranch}_0$ activation record is replaced by a `KPBranch` activation record. The effect of this step is to change the behavior of the in-flight traversal of the affected

```
sum(node* n, k : kont*) → void {
  while (true)
    k = try_promote(k) if heartbeat() else k
    if (n == null)
      s_a = 0
      while (true)
        k = try_promote(k) if heartbeat() else k
        match *k with
        | KSBranch_0{n=n_1, k=k1} ⇒ {
          n = n_1.bs[1]
          k = KSBranch_1{s_0=sa, n=n_1, k=k1}
          break }
        | KSBranch_1{s_0, n=n_1, k=k1} ⇒ {
          s += s_0 + n_1.v; k = k1 }
        | KPBranch {i, s, tj} ⇒ {
          s[i] = sa; join(tj); return }
        | KTerm ans ⇒ { *ans = sa }
    else { k = KSBranch_0 {n=n, k=k}; n = n.bs[0] } }
```

Figure 7. Step 5: give the traversal a heartbeat.

```
try_promote(k : kont*) → kont* {
  kt = find_outermost(k, λ k ⇒ {
                         match *k with
                         | KSBranch_0 _ ⇒ true
                         | _ ⇒ false })
  if (kt == null) { return k }
  match *kt with
  | KSBranch_0{n, k=kj} ⇒ {
    s = new int[2]
    tj = new_task(λ () ⇒ {
           k_0 = KSBranch_1{s_0=s[0] + s[1], n=n, k=kj}
           sum(null, k_0) })
    tc = new_task(λ () ⇒ {
           sum(n.bs[1], KPBranch {i=1, s=s, tj=tj})})
    fork(tj, tc)
    k_1 = KPBranch {i=0, s=s, tj=tj}
    return replace(k, kt, k_1) } }

// returns a pointer value k1 to the outermost activation record in k
// s.t. f(k1), or null if there is no such k1 in k
find_outermost(k : kont*, f : kont* → bool) → kont*

// returns a pointer value k1 s.t. any frame kt in
// k is replaced by k_1
replace(k : kont*, kt : kont*, k_1 : kont*)
  → kont*
```

Figure 8. Step 6: implement promotion

$$
\begin{aligned}
E\,[\texttt{fork2join}\,(f_0, f_1)]_k \stackrel{\text{def}}{=} \{ & \\
& \texttt{tj} = \texttt{new\_task}\,(k) \\
& \texttt{t}_0 = \texttt{new\_task}(E\,[f_0\,()]_{\lambda\,()\,.\,\texttt{join(tj)}}) \\
& \texttt{t}_1 = \texttt{new\_task}(E\,[f_1\,()]_{\lambda\,()\,.\,\texttt{join(tj)}}) \\
& \texttt{fork}\,(\texttt{t}_0, \texttt{tj})\,;\,\texttt{fork}\,(\texttt{t}_1, \texttt{tj})\ \}
\end{aligned}
$$

Figure 9. CPS conversion rule for `fork2join`

branch so that it synchronizes with its sibling task once its part of the traversal finishes.

Optimizing the layout of continuation records. One final optimization needed for work efficiency relates to the representation of the continuation. As it is currently, continuation records are laid out in different heap objects, and are linked by explicit `kont*` pointers. This representation can be improved by simply laying out these records in a linear fashion, which we do by packing the records in a STL deque container. This way, our final implementation is more compact, as it allows us to do without explicit tail-pointer values.

B CPS conversion of `fork2join`

Originally, we introduced two ways of achieving fork-join parallelism: the direct style as in our `fork2join` primitive and the CPS-friendly library interface. Here, we show the direct connection between the two versions in terms of a CPS transformation, as shown in Figure 9.