Compiling Loop-Based Nested Parallelism for Irregular Workloads

Yian Su
Northwestern University
Evanston, IL, USA

Mike Rainey
Carnegie Mellon University
Pittsburgh, PA, USA

Nick Wanninger
Northwestern University
Evanston, IL, USA

Nadharm Dhiantravan
Northwestern University
Evanston, IL, USA

Jasper Liang
Northwestern University
Evanston, IL, USA

Umut A. Acar
Carnegie Mellon University
Pittsburgh, PA, USA

Peter Dinda
Northwestern University
Evanston, IL, USA

Simone Campanoni
Northwestern University
Evanston, IL, USA

Abstract

Modern programming languages offer special syntax and semantics for logical fork-join parallelism in the form of parallel loops, allowing them to be nested, e.g., a parallel loop within another parallel loop. This expressiveness comes at a price, however: on modern multicore systems, realizing logical parallelism results in overheads due to the creation and management of parallel tasks, which can wipe out the benefits of parallelism. Today, we expect application programmers to cope with it by manually tuning and optimizing their code. Such tuning requires programmers to reason about architectural factors hidden behind layers of software abstractions, such as task scheduling and load balancing. Managing these factors is particularly challenging when workloads are irregular because their performance is input-sensitive. This paper presents HBC, the first compiler that translates C/C++ programs with high-level, fork-join constructs (e.g., OpenMP) to binaries capable of automatically controlling the cost of parallelism and dealing with irregular, input-sensitive workloads. The basis of our approach is Heartbeat Scheduling, a recent proposal for automatic granularity control, which is backed by formal guarantees on performance. HBC binaries outperform OpenMP binaries for workloads for which even entirely manual solutions struggle to find the right balance between parallelism and its costs.

ACM Reference Format:
Yian Su, Mike Rainey, Nick Wanninger, Nadharm Dhiantravan, Jasper Liang, Umut A. Acar, Peter Dinda, and Simone Campanoni. 2024. Compiling Loop-Based Nested Parallelism for Irregular Workloads. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3620665.3640405

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA

ACM ISBN 979-8-4007-0385-0/24/04.
https://doi.org/10.1145/3620665.3640405

1 Introduction

Parallel programming continues to be hampered by the inability of compilers to translate fork-join nested parallelism in client programs into binaries that perform close to the limits of modern multicore hardware. One reason is that a binary that actually executes all possible parallel tasks described by high-level fork-join constructs (e.g., the `parallel for` of OpenMP) immediately encounters a massive overhead that results in slowdowns measured in orders of magnitude, thereby squashing all benefits of parallelism. Just spawning a parallel task/thread requires a few thousand cycles (even in a customized OS and even for the latest OS solutions), with context-switch costs being similar [21, 23]. Unfortunately, irregular workloads like sparse tensor computation and graph analytics tend to have many independent loop iterations that only take a few tens of clock cycles [6]. Spawning one parallel task per loop iteration in these workloads easily leads to slowing down execution rather than speeding it up, both due to task-spawning overheads and the barrage of context switches due to having many more tasks (e.g., loop iterations) than cores. In a run-time environment without preemption, the context-switch problem may be diminished, but the potential for excessive task-related costs remains.

On the other hand, coarsening loop iterations (known as chunking) to generate fewer tasks can easily degrade performance by starving cores of tasks. This is the old problem of *parallelism granularity control* [11, 13, 22][1]. For example, consider a loop where each iteration processes an element of an input vector, and the amount of computation performed depends on whether the element is non-zero. If the vector's non-zeros are not uniformly distributed, then static chunking leads to an unbalanced computation. Because the fork-join model requires a barrier at the end, the slowest task (the unlucky one with the most non-zeros, say) dictates the latency of the entire loop. The cores running the faster tasks (processing fewer non-zeros) are now idle for a substantial

[1] Our results in Section 6.7 corroborate this prior work.

period of time. The appropriate chunking to avoid this is input-dependent. Clearly, we need a dynamic solution.

The status quo places the burden on the application programmer's shoulders. Accordingly, programmers sidestep compilers by expressing a specific parallelism granularity in their code, with the hope of achieving the right balance between fork-join overhead and maximizing the program's performance on their multi-core CPU. But this decision is typically not transferable between platforms, as it depends on architecture-specific aspects related to its out-of-order capabilities (e.g., instruction-issue widths) and inter-core communication costs, as well as OS-specific aspects like thread-creation and thread-switching costs [48]. Such complications are most noticeable and difficult to manage in a certain, increasingly important class of applications that is characterized by its workload being irregular, its performance characteristics being significantly influenced by the input, and its input being drawn from a large range of possible shapes. Recent applications of this kind include those that perform significant amounts of sparse-matrix computation, e.g., in machine-learning algorithms [19], those that use a domain-specific language, e.g., for tensor algebra [29], or those that analyze large graphs [55]. For applications featuring such irregular parallelism, the granularity-control problem threatens to result in either a platform-specific, hard-to-maintain, and costly codebase, or binaries that do not leverage the full potential of the underlying architecture.

Recent research produced an alternative approach to granularity control, known as heartbeat scheduling [1]. It is the first approach that *provably* controls the overheads of parallelism *automatically*, *without embedding platform-specific aspects* into the program's code. At a high level, the properties proven for heartbeat scheduling are twofold: task-related costs are well amortized and the asymptotic amount of parallelism in the source program is preserved. Heartbeat scheduling works by postponing the decision of generating additional parallelism to run-time and makes that decision online, at a regularly occurring event, called the heartbeat. A heartbeat happens at a fixed rate while the program executes, and enables the program to continuously adapt to changing parallelism. The essential takeaway is that heartbeat scheduling lets programmers express *all* possible fork-join parallelism in their algorithm while allowing the runtime to decide which portion to materialize and when.

Heartbeat scheduling implementations currently require programmers to write their code, at least in part, in assembly. The reason is that the implementations require the code to be structured in an unconventional style, even though the program's code does not make granularity control decisions itself. A mapping to such a bespoke program structure from high-level fork-join programs is not supported by today's compilers. This gap leaves heartbeat scheduling in the hands of a few highly capable programmers with significant time to invest in restructuring their code. We seek to democratize heartbeat scheduling, allowing all programmers to reap its benefits. This requires a compiler that can automatically translate high-level fork-join constructs into a binary that is amenable to heartbeat scheduling at runtime.

This paper describes the first compiler capable of automatically generating binaries that are compatible with heartbeat scheduling. The heartbeat compiler (HBC) consumes an ordinary C/C++ program with high-level fork-join constructs, like those available in OpenMP or Cilk. HBC then automatically deconstructs the code into tasks that can be further split when driven by heartbeat events. The heartbeat linker (also introduced by this paper) modifies the generated assembly both to link a signaling mechanism to generate heartbeats and to link with the heartbeat runtime. Binaries generated by HBC from programs with irregular workloads have significantly better performance than binaries generated by a more conventional OpenMP compiler. Compared to sequential execution, HBC boosts the performance of irregular workloads from 14.2× (OpenMP) to 21.7× (Heartbeat) on a 64-core Intel-based machine (Fig. 4). Finally, the performance of binaries generated with HBC, which compiles them in just a few seconds, is comparable to what was previously obtained by manually writing heartbeat scheduling binaries over several months [42].

The contributions of this paper are:

- We introduce the first compiler capable of realizing heartbeat scheduling in a fully automated manner, generating a binary from given C/C++ programs with loops, which are written using conventional high-level fork-join constructs.
- We introduce the first linker capable of automatically embedding into a binary the rollforwarding mechanism, which prior work proposed for heartbeat scheduling.
- We introduce a new algorithm for automatically generating all possible parallel tasks from nested loops that can be parallelized by heartbeat scheduling. This improvement includes the generation of what we call the *leftover tasks* that were not generated before.
- We design, implement, and evaluate the first compilation pipeline to automatically generate binaries that are capable of performing heartbeat software polling (i.e., continuous checking if an event happened) with little overhead.
- We design, implement, and evaluate the first interrupt-based mechanism to deliver heartbeats on Linux with a custom kernel module to reduce the latency of delivering heartbeats.
- We compare for the first time two signaling mechanisms that are both able to deliver heartbeats. This comparison suggests a counter-intuitive result: software polling is as good as interrupt-based mechanisms.

2 Background

To summarize key background concepts, we use the following running example: the sparse-matrix by dense-vector

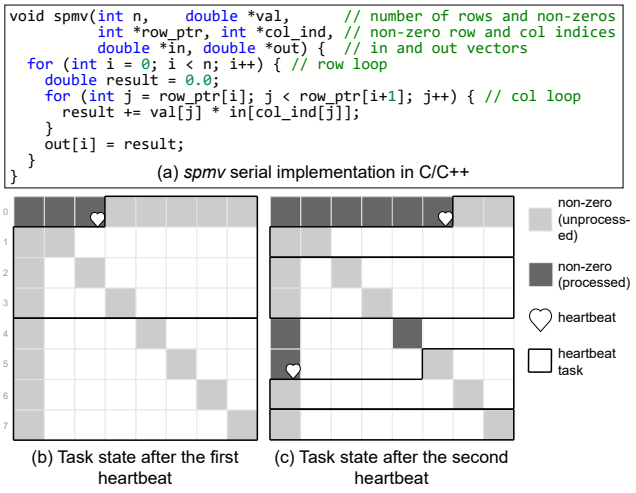

Figure 1. *spmv* and the heartbeat execution model.

product (*spmv*, shown in Fig. 1(a)). We use the implementation of *spmv* that expects its input matrix to be represented in compressed sparse-row format. The inner and outer loops are parallelized by annotating them as DOALL loops (and treating the reduction variable `result` appropriately). A DOALL loop is a loop where all iterations can run in parallel, in a fork-join execution model, without communication between them. The DOALL decoration can be done using, e.g., the OpenMP `parallel for` construct.

Iterations of a DOALL loop run in parallel by creating and executing tasks. First, tasks are created by partitioning the loop iterations (one task per iteration at the finest granularity). Then, tasks are dispatched at run-time to parallel threads (via thread-specific queues) that run on the parallel cores of the underlying architecture (one thread per core). Notice that the number of tasks can safely (and typically does) exceed the total number of threads or cores.

The parallelism granularity-control problem. If *spmv*'s DOALL loops were to maximize parallelism and spawn a task for each iteration, *spmv* would risk losing its parallel speedup to the overhead costs of creating and managing tasks. Task overheads can be amortized by assigning multiple subsequent loop iterations (called chunk) to a task. This operation is called *chunking* and the number of iterations assigned to a task is called the *chunk size*. OpenMP (like other languages) allows programmers to manually specify the chunk size of a loop. But tuning the chunk size risks pruning away too much parallelism, and even if the program is well-tuned, there is a risk of overfitting [47]. Achieving consistent granularity control is further challenged by irregularity in workloads. For example, our *spmv* is highly input dependent, and consequently, based on the sparsity pattern of the matrix, the concentration of parallelism may fluctuate between its two loops.

Heartbeat scheduling. Heartbeat scheduling achieves granularity control for nested fork-join constructs in an adaptive manner by ensuring that each task is amortized against a specified amount of useful work in the program. That amount is determined by the heartbeat rate, a system-wide parameter that controls the rate at which tasks are spawned. Let us see it in action by stepping through *spmv* when called on a well-known challenge input, the arrowhead matrix, whose diagonal, first row, and first column are filled with non-zero elements. Suppose, for explanatory purposes, the heartbeat rate takes as much time as is needed to process 3 non-zero elements. Now, after (serially) processing up to the 3rd element of the first row, our initial task has been running for enough time to be interrupted by a heartbeat. When it arrives, the interrupt kickstarts a *promotion*. Promotion is the process used by heartbeat scheduling to activate latent parallelism (in our case, the remaining loop iterations) held by a running task.

Under the hood, this two-step process of interrupt, followed by promotion, is implemented by the coordination of two mechanisms. The interrupt is driven by a hardware-based timer interrupt. Promotion happens downstream of an interrupt, where it reifies the loop context, divides up loop iterations, and parcels out the pieces to new tasks. This latter mechanism is called the *promotion-ready program point (PRPPT)* [42]. These two mechanisms are linked together by the application of a classic technique for synchronizing in the presence of interrupts known as *rollforwarding* [36] (see §4 for details).

To ensure parallel scalability, heartbeat scheduling assigns the highest priority to outermost parallelism, a policy we call the ***outer-loop-first*** policy. Let us see the policy in action in our running *spmv*, where we left off. As it enters its promotion handler, our running task sees latent parallelism in the outer loop, making that loop the target for promotion. The result is depicted in Fig. 1(b), where we see an even division of the remaining iteration space split into two subtasks. This process continues in a recursive fashion. The diagram in Fig. 1(c) shows a future program state after two more heartbeats, treated by each of the previous two tasks.

Current heartbeat scheduling implementation. Recent work proposed TPAL, the state-of-the-art, low-level model for heartbeat scheduling, but TPAL does not address automation: each TPAL benchmark started as a C++ source program and was manually modified by inserting PRPPTs, as needed. Further manual intervention was needed to mitigate performance issues related to PRPPTs, which required loop chunking by hand. Definitions of the PRPPT handler functions themselves require custom logic, which had to be written by hand. The back-end of TPAL used a semi-automatic rollforward compiler, which required significant human intervention. Finally, although crucial for practical performance, TPAL provided no comprehensive solution for

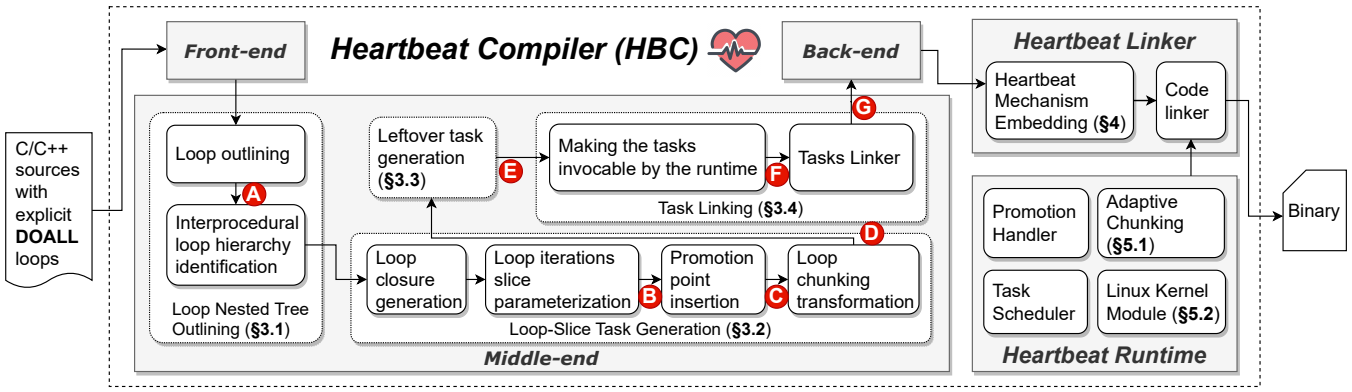

Figure 2. Compilation pipeline of HBC including a custom linker and runtime.

driving interrupts. Its approach was based on special support in a research kernel. A similarly effective solution for a commodity OS is lacking.

3 The Heartbeat Compiler (HBC)

The HBC pipeline (Fig. 2) takes C/C++ source files, where parallelism is specified by DOALL annotations, and lowers the program via a series of passes, resulting in a binary that implements heartbeat scheduling. The front-end of our HBC pipeline is an extension to the compiler `clang` that identifies all DOALL annotations in the program and emits them in the form of custom LLVM IR metadata terms. Our metadata-enhanced LLVM IR is consumed by a series of passes we implemented as extensions of LLVM's middle end. These lowering passes isolate DOALL loops by outlining them into separate functions and building a representation of their nesting structure (§3.1). This nesting structure is used by our middle-end passes to automate the generation of PRPPTs.

To do so, the middle-end computes the closure of each outlined loop such that the generated function can be invoked to execute a specific set of subsequent loop iterations (e.g., from 5^{th} to 9^{th} iteration) while being able to promote parallelism at runtime (§3.2). The result is a set of functions (one per DOALL loop) that can be invoked as parallel tasks, called *loop-slice tasks*. After generating the loop-slice tasks, the HBC middle-end generates the *leftover tasks* (§3.3). According to the outer-loop-first policy of heartbeat scheduling, the decision of splitting the parallel task T_o related to an outer loop L_o while running its z^{th} iteration can be made during the execution of the j^{th} iteration of one of its inner loops L_i. To maximize parallelism, our work splits T_o into three parallel tasks whose code is generated at compile time. The first one is the execution of the loop-slice task of L_o to execute the first half of the iterations left of L_o. The second one invokes the same loop-slice task but executes the second half of the iterations left of L_o. The last task will execute the remaining computation of the z^{th} iteration of L_o, which includes the remaining computation of the current invocation of L_i (from its $j+1^{th}$ iteration to the end) as well as the code from the end of L_i to the end of the z^{th} iteration of L_o. We call this last task a *leftover task*. As shown in Fig. 1(b), the first two loop-slice tasks generated by HBC cover the remaining iterations of the `row` loop from `1` to `7`. And a third leftover task covers the rest of the iterations of `col` loop from `3` to `7`, and the remaining computation after invoking the `col` loop of row `0`. That is, `out[i] = result`.

After generating both loop-slice tasks and leftover tasks, the HBC middle-end links them into the original code (§3.4). This is done by first modifying the above tasks to make them controllable and invocable by the HBC runtime (so the runtime can perform parallelism promotions), and then it replaces the original IR code of the target loops with invocations of the loop-slice tasks where the slice specified is the entire iteration space of the related loop. Hence, if no promotion happens at runtime, the execution stays sequential.

After the passes in the middle-end, we lower the program to binary code, using an off-the-shelf back-end available in the LLVM codebase (e.g., the intel *x86_64* back-end). The output program then reaches our heartbeat linker, the final stage of our pipeline. The heartbeat linker enables the heartbeat to be seen by the running program by injecting hooks from the runtime into the program's IR.

3.1 Loop Nested Tree Outlining

The middle-end starts by outlining all DOALL loops. For each loop of this set, a conventional data-flow analysis identifies its live-in and live-out variables. Then, a function is created to copy the loop in it as done by prior work [5, 7, 14, 33, 37]. This function has live-ins of the loop as parameters and its live-out variables are passed as references. Now the loop can be executed by invoking the function with proper parameters. The resulting functions are then modified to replace any nested DOALL loops with a call to their outlined versions. Live-ins and pointers of the live-outs of a nested DOALL loop are passed via parameters of the injected call. Live-outs are allocated on the caller's stack.

HBC needs to represent the original nesting relation of DOALL loops because it is needed by the heartbeat runtime to implement promotions. Hence, HBC extends the loop-nesting-relation analysis in LLVM to make it interprocedural. The result of this analysis is a directed graph where nodes are loops and edges represent the nesting relation from a parent loop to one of its children (similarly to [8]). HBC prunes this graph to remove all nodes that do not represent DOALL loops. The result is a tree because the original DOALL loops formed a tree (they came from the same original function).

Our solution follows the general approach of OpenMP compilers, where DOALL loops are outlined first into separate functions before reaching the middle-end. An alternative solution to this problem is to compute the loop-nesting relation of the original code and then implement an ad-hoc outliner transformation that also generates the mapping from the original loops to the new ones that now belong to different functions. This solution would be equivalent to the one we implemented. We chose our solution to allow us to re-use the general-purpose outliner transformation already available in conventional compilers, which does not provide the loop mapping mentioned above.

The inter-procedural loop nesting tree is used to compute the IDs of each DOALL loop. The ID of a DOALL loop is a pair `(level, index)`, where `level` represents the nesting level of that loop, starting from 0 for the root loop, and `index` represents the position of the loop within its respective nesting level, incrementing from 0 onwards. The root loop has an index value of `0`. In *spmv*, the `row` loop has the identification pair `(0, 0)` and the `col` loop has the identification `(1, 0)`.

3.2 Loop-slice task generation

Following Fig. 1(b), when the first heartbeat interrupts *spmv*'s `col` loop, a promotion handler is called to spawn parallel tasks. To seed the task, we need the loop's environment. The challenge is that the loop we need to promote (the `row` loop) is not the loop that was in execution at the time a heartbeat was received (the `col` loop). To solve it, we need to pass the environment of the `row` loop to the inner `col` loop and its promotion handler. To do so, HBC generates code to couple each loop with a data structure that captures its closure, its iteration space, and its induction variable. We call this structure the *Loop-Slice Task (LST) context*. LST contexts of a given DOALL loop of a loop nesting tree are allocated before the outermost loop of that tree (the root loop) is invoked. These LST contexts are passed down to all nested loops as a set. When a nested loop invokes the promotion handler, all loops' LST contexts are accessible to the promotion handler to seed any task that runs any parent loop.

Loop closure generation. HBC first modifies the signature of the outlined DOALL loops of each loop-nesting tree such that all parameters are replaced by a pointer to a set of LST contexts, which includes the LST context of the invoked loop itself. The outlined loop function is now considered a loop-slice task. Next, HBC generates code to load live-ins, live-outs, and iteration space to run from the corresponding LST context and replaces all values previously read from the function parameters with loaded values.

Promotion point insertion. HBC inserts a call to the promotion handler at the latch of a DOALL loop, for all DOALL loops, to enable promotion within a loop-slice task. The latch of a loop L is a basic block within L that is the predecessor of the header of L [4]. DOALL loops only have one latch.

The promotion handler returns whether a promotion happened (`1`) or not (`0`). When a promotion happens, the promotion handler returns only when the execution of all remaining loop iterations has been completed. Therefore, HBC adds a conditional branch to read the value returned by the promotion handler to exit the loop when a promotion happens.

Loop chunking transformation. The promotion handler inserted by HBC can degrade performance because the call breaks up the control flow of the target loop, blocks compiler optimizations, and can impose dynamic costs (depending on which mechanism the heartbeat linker uses to drive heartbeats). To mitigate such costs, the loop chunking transformation modifies the target loop to invoke the promotion handler every S number of iterations (called chunk size). This is obtained by creating within the target loop a sub-loop L_s whose body contains only the original code of the target loop.

The loop chunking transformation needs to guarantee that the promotion handler is invoked every S iteration. This requires extra code for when the number of iterations executed by the original loop is not a multiple of S. In more detail, a chunk can be partially finished within a given invocation of the target loop (e.g., S is bigger than the total number of loop iterations). Therefore, a task needs to track how many iterations remain to be executed till the full completion of a chunk between (potentially several) loop invocations. To do so, each task maintains a private counter R (initialized to S), and the number of iterations of L_s is set to be the minimum between R and the number of iterations left to finish the current invocation of the target loop. The chunking transformation adds a check after L_s, which will execute after L_s finishes its iterations. It compares the number of iterations C executed by L_s to R, and it invokes the promotion handler only if they match (and reinitializes R to S). Otherwise, it updates R to be $R - C$ (called chunk size transferring).

The loop chunking transformation is applied to every innermost DOALL loop of a loop nesting tree. Finally, the chunk size S is determined by a dynamic technique (§4).

Algorithm 1 Generating all leftover tasks between loop pairs.

Input: t: Loop nesting tree
Input: s: Set of all LST contexts of all loops
1: **for all** l : $t.getLeaves()$ **do**
2: $p \leftarrow l.getParent()$
3: **while** p **do**
4: GenerateLeftoverTask(l, p, s)
5: $p \leftarrow p.getParent()$

3.3 Leftover task generation

This compilation step generates all possible leftover tasks of a loop nesting tree of DOALL loops. The generation of such tasks allows HBC to generate additional parallelism compared to prior work, as the leftover task of a promotion can now run in parallel with the other two tasks generated with it. This opportunity was not explored by prior work because tasks were written manually and the number of leftover tasks can grow quadratically with the number of loops in a nesting tree. Hence, it is too much to ask from a programmer to write them all. However, HBC generates them automatically while keeping the code size under control by sharing code between leftover tasks.

Iterating over possible leftover tasks. The possible leftover tasks depend on the shape of the loop nesting tree. They are generated using Algorithms 1 and 2. Algorithm 1 takes as input a loop nesting tree of DOALL loops and all LST contexts. The algorithm identifies the need for generating a leftover task for a pair of loops (L_i, L_j). The loop L_i represents the loop that gets a heartbeat that leads to a promotion; L_j represents the loop that gets split. To identify the above set of pairs, Algorithm 1 iterates over the leaves of the loop-nesting tree (line 1). For each leaf l, it iterates over its ancestors starting from its parent (lines 2-5). For each ancestor p, the pair (l, p) is identified and the associated leftover task is generated by invoking Algorithm 2 (line 4).

Code generation of a leftover task. For each (L_i, L_j) pair found in Algorithm 1, HBC creates a leftover task that will execute in that case. Thus, HBC implements Algorithm 2 taking as input L_i and L_j that represent the case where L_i gets a heartbeat and L_j gets split. It also takes as input the LST contexts of the loops included in the current loop nesting tree. The output is the leftover task t for the (L_i, L_j) case.

The algorithm starts by creating a new empty leftover task t (line 2) followed by increasing the induction variable by 1 of the LST context used by L_i when it gets a heartbeat (lines 3-4). The algorithm then adds the code to invoke the loop-slice task of L_i starting from its next iteration until the end (line 5). At this point, Algorithm 2 has generated the code for t to complete the current invocation of L_i.

What is left for the algorithm is to append the code that composes the work between the end of L_i to the end of the current iteration of L_j, referred as *tail work*. To do so, it

Algorithm 2 Generating a leftover task between two loops.

Input: L_i: Loop that gets a heartbeat
Input: L_j: Loop that gets split
Input: s: Set of all LST contexts of all loops
1: **function** GenerateLeftoverTask$(L_i,\ L_j,\ s)$
2: $t \leftarrow new\ LeftoverTask()$
3: $t.addCode(c \leftarrow s.getLSTContext(L_i))$
4: $t.addCode(c.increaseIV(1))$
5: $t.addCode(call$ LoopSliceTask$(L_i, c))$
6: $prev \leftarrow L_i$
7: $p \leftarrow L_i.getParent()$
8: **while** $p \neq L_j$ **do**
9: $t.addCode(c \leftarrow s.getLSTContext(p))$
10: $t.addCode($TailWork$(p, prev, c))$
11: $t.addCode(c.increaseIV(1))$
12: $t.addCode(call$ LoopSliceTask$(p, c))$
13: $prev \leftarrow p$
14: $p \leftarrow p.getParent()$
15: $t.addCode(c \leftarrow s.getLSTContext(L_j))$
16: $t.addCode($TailWork$(L_i, prev, c))$

iterates over the ancestors of L_i starting from its parent to the ancestor just before reaching L_j (lines 8-14). For each ancestor p, Algorithm 2 adds code to get its LST context, which contains the current induction variable of p (line 9). It then places the tail work of p, which composes of the code after invoking the previous ancestor till the end of the body of loop p using p's LST context (line 10). After that, Algorithm 2 increases p's induction variable by 1 and invokes p's loop-slice task passing its updated LST context (lines 11-12). Finally, the algorithm adds code for the tail work of L_j using its LST context after invoking its previous ancestor (lines 15-16).

3.4 Task linking

At this point of the compilation pipeline, the HBC middle-end has generated all possible tasks that could run in parallel. To be performant the heartbeat runtime also needs to quickly access the right triple of tasks that will instantiate when a heartbeat happens. This triple depends both on the innermost loop that has received the heartbeat and the loop that gets split. Because of this, the HBC middle-end ends with the next two steps. First, all tasks are organized to enable an efficient identification of the triple of tasks. Then, the tasks are linked into the program by allocating and initializing the necessary LST contexts of the DOALL loops of a loop nesting tree. This allocation is performed just before jumping to the header of the root of the corresponding loop nesting tree.

Making the tasks invocable by the runtime. To help the heartbeat runtime to quickly find the right task to invoke for a promotion, this step takes advantage of the structure of the loop IDs described in §3.1. The middle-end allocates a two-dimensional array for every loop nesting tree with DOALL loops. This array is called the *loop-slice task array*

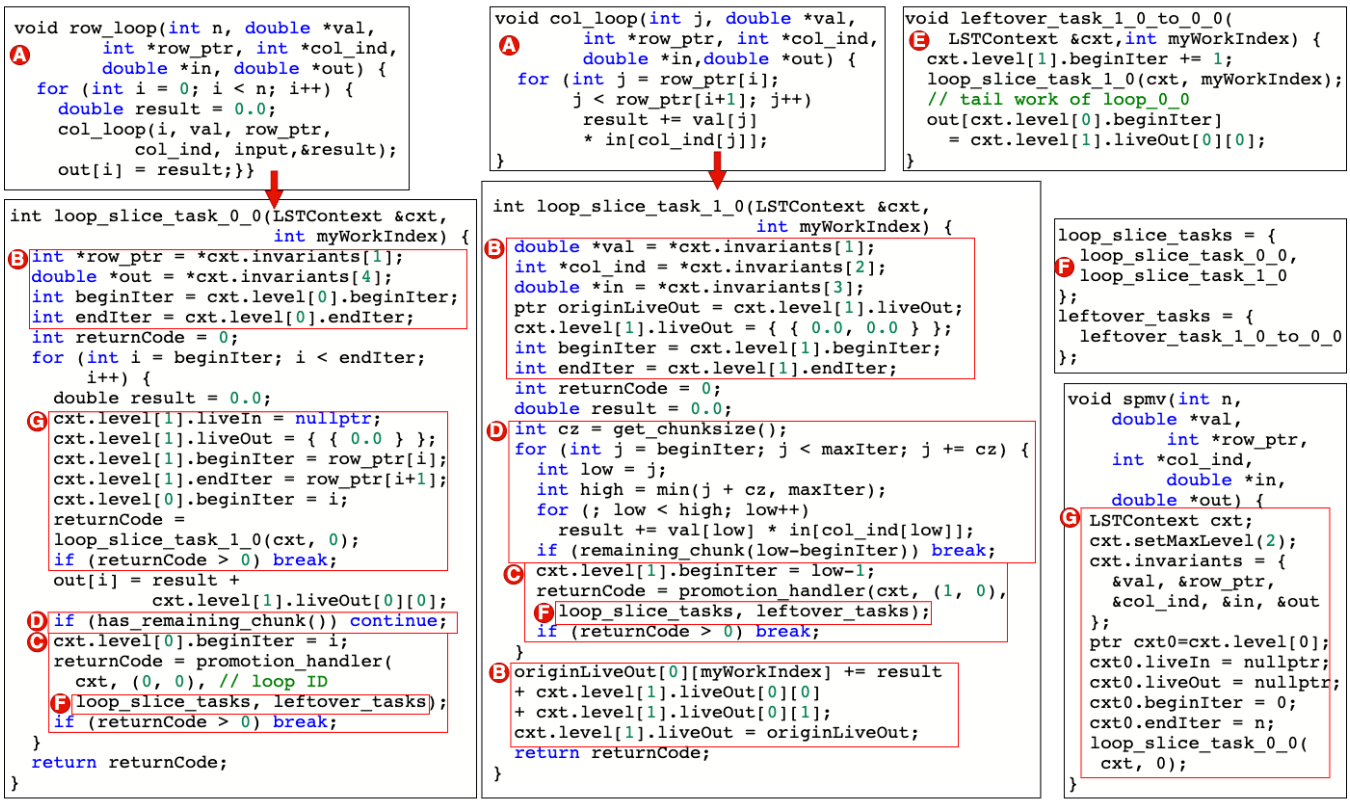

Figure 3. Sequence of transformations of *spmv* that are performed by the HBC's middle-end. We used C++ code for readability and illustrative purposes. HBC performs these transformations in LLVM IR. The white letter in a red cyrcle attached to the code surrounded by a rectangular red box indicates where in the pipeline of Fig. 2 that code is added.

of a loop nesting tree. Each value of this array is a pointer to the function of a loop-slice task. The array is passed as a parameter to the promotion handler inside all loop-slice tasks that compose this nesting tree. The `level` and `index` that compose the ID of a loop are used as the row and column of the loop-slice task array. The value obtained at that row and column is the pointer of the loop-slice task with that specific loop ID. The loop-slice task array is allocated as globals and are statically initialized.

Leftover tasks also need to be efficiently retrieved by the heartbeat runtime. To this end, the HBC middle-end allocates a hash table called the *leftover task table* where it returns the pointer to the function of a leftover task from a pair of loop IDs. The first ID of the input pair is the loop that has received the heartbeat. The second ID is of the loop that gets split. A perfect hashing function is generated at compile time to perform this mapping. Finally, the leftover task table is allocated as global, statically initialized, and the signatures of all tasks are modified to take a leftover task table.

Tasks linker. The last step of the HBC middle-end is to allocate the LST contexts for all loop nesting trees with DOALL loops. For every loop-nesting tree, it replaces the code of the loop at the root of the tree with a call to the equivalent loop-slice task after preparing the initial environment and specifying the whole iteration space for that loop inside its LST context. Any call to a nested loop inside a loop-slice task is replaced by the call to its corresponding loop-slice task, with the environment and the iteration space set by the parent loop. Fig. 3 shows the full transformation of *spmv*.

4 Heartbeat Linker

HBC implements two mechanisms for handling heartbeats: software polling and hardware interrupts. Software polling proactively reads the timestamp register to decide if a heartbeat has arrived. Hardware interrupts invoke heartbeat processing on receipt of a timer interrupt. HBC uses software polling by default as it delivers better average performance on an unmodified Linux platform (§6.5), but allows the user to select either heartbeat mechanism.

Software polling injection. The linker injects the polling function at PRPPTs (§2) and guards the promotion handler call with a conditional branch. The polling function reads the timestamp register (i.e, TSC for x86) to tell whether a

heartbeat has arrived. If it has, the promotion handler gets called, generating parallelism. Otherwise, it does not.

Hardware interrupt enablement via rollforwarding compilation. In software polling, we always pay the cost of the polls. In contrast, using hardware timer interrupts avoids this, but an interrupt can arrive at any assembly instruction, not just at PRPPTs, and thus rollforwarding [36] is needed.

Conceptually, we implement rollforwarding by having the hardware interrupt trigger an instruction pointer switch from the "source" version of the object code (which contains no polls) to the "destination" version (which contains the polls). A mapping table (the "rollforward table") from source to destination instruction addresses is included in the binary and used by the hardware interrupt mechanism (§5.2) to find the appropriate "destination" address to switch to. An inverse table (the "rollback table") is also needed.

Our rollforward compiler (RFC) automates the process of producing the source and destination code (at the assembly level), and tables. RFC is a source-to-source translator that operates over the assembly intermediate file (the "`.s` file"). To generate the source, every input line is prepended with a new label we generate based on its line number (e.g. `__RF_SRC_42`). Lines that involve polling have their instruction elided. To generate the destination, every input line is repeated, but now prepended with a label that corresponds to the source line (e.g. `__RF_DST_42`). In the destination, the polling instructions are left in place. Finally, we emit the tables, mapping between the newly introduced labels (i.e., we add `__RF_SRC_42` ↔ `__RF_DST_42`). GNU `ld` resolves all the labels to addresses. Special care is taken to handle numerous edge cases and other issues.

Despite the apparent complexity of this transform, the novelty of RFC is that it operates entirely using regular expressions, instead of requiring compiler backend changes or assembler modifications. RFC comprises 250 lines of dense Perl. It takes < 1 second to translate each of our benchmarks.

5 Heartbeat Runtime

This section describes the two runtimes that we designed for the two heartbeat solutions we implemented for HBC: software polling and interrupt-based solutions.

5.1 Software Polling using Adaptive Chunking (AC)

AC dynamically updates the chunk size of a leaf loop to reduce the number of wasted polls per heartbeat. The initial chunk size is set to 1. The runtime runs a sliding window algorithm [31] where the loop chunk size is updated only at the end of the window when a given number of heartbeats (called the window size) have been received. Each worker thread keeps track of how many times the polling function is invoked since the last heartbeat. On each heartbeat, the runtime logs the number of polls made during that heartbeat interval. After the number of heartbeats that compose the window, a thread records the minimum number of polls in the log since the beginning of the window. Then, it computes the ratio of the minimal poll count to a ***target polling count***. This ratio is then multiplied to the current chunk size to form the new chunk size (minimum 1) for the worker thread.

5.2 Hardware Interrupt-based Solutions

HBC supports hardware interrupt-based heartbeats using either portable user-level code or a Linux kernel module.

Interrupt Ping Thread for Purely User-level Operation. This mechanism uses the POSIX `SIGALRM` signal to drive the heartbeat via the *ping thread* model from earlier work [42].

Kernel Module for Accelerated Operation. In earlier work [42], an alternative mechanism was proposed and evaluated that used the x86 APIC hardware timer and interprocessor interrupt (IPI) mechanisms directly. While this dramatically improves heartbeat accuracy, precision, and scalability, it requires the non-trivial inclusion of the application directly into a specialized kernel. As a middle ground, we developed a Linux kernel module that provides many of the same benefits, while requiring no application changes.

Our kernel module configures one core to use the kernel's `hrtimer` interface, a thin veneer over the APIC timer, and allows the runtime to configure heartbeat rates. A timer interrupt invokes the module, which in turn broadcasts an IPI to all current heartbeat-enabled cores. Each of these determines if the current interrupted user thread is a heartbeat application thread. If it is, the handler searches the rollforward table for the interrupted "source" RIP and finds its corresponding "destination". It then edits the return address of its own interrupt frame so that on IRET, control returns to the destination address, switching to the rollforward code.

Unlike a ping thread, this structure operates mostly in kernel, directly using the hardware. It also avoids the high cost of general-purpose POSIX signal injection. An event requires only 3800 cycles (user→kernel→user) on our system.

6 Evaluation

This section evaluates the first fully automatic solution for heartbeat scheduling, HBC. After describing the experimental settings, this section shows the higher performance obtained by HBC compared to the clang-based OpenMP compiler for irregular workloads. Then, this section compares the performance of the binaries automatically generated by HBC against those manually generated in the TPAL work [42]. Our results suggest that the automation done by HBC to generate binaries preserves the performance obtained by the intense manual work done by TPAL. By leveraging our automatic solution, this paper shows the first empirical comparison between two signaling mechanisms for heartbeat scheduling: software polling and hardware interrupts. Our results counter-intuitively show that the former is as good as

the latter, even when the latter is implemented with custom OS support. This result has the potential to help broaden heartbeat scheduling's adoption as it can now be used on the off-the-shelf Linux OS without sacrificing performance. This section also evaluates the need for adapting the chunk size at run-time and evaluates how well our runtime solution performs. Finally, this section ends by comparing HBC and OpenMP for regular workloads.

6.1 Experimental Settings

Next, we describe our test-bed, where we performed all empirical evaluations described in this paper. We will open source HBC and all benchmarks/results described in this section.

HBC, TPAL, and OpenMP compilation. HBC builds upon NOELLE [34], a compilation framework that augments LLVM with additional dependence-oriented abstractions like the Program Dependence Graph [18]. We ported NOELLE to LLVM 14 because the public version only supports LLVM 9. HBC, OpenMP, and TPAL parallelize the same set of loops for all benchmarks. All loops parallelized are DOALL.

LLVM 14 is the underlying compilation infrastructure used for all of our results. The TPAL and OpenMP binaries are generated using clang as included in LLVM 14. This enables a fair comparison between the three systems as they share a significant portion of their compilers, leaving parallelism granularity control to be the main difference between them. All benchmarks are compiled using `-O3 -march=native` with vectorization passes disabled (as done in evaluating TPAL [42]).

Benchmarks. We evaluated HBC on two sets of benchmarks. Table 1 lists the benchmarks we used to evaluate HBC, along with the input used, their regularity, and whether they were also evaluated by TPAL.

The first set of benchmarks is composed of all the iterative (eight) benchmarks that the prior work TPAL [42] was evaluated on. As HBC aims to replace all significant manual efforts behind the manual generation of the assemblies used in TPAL [42], it is important to evaluate HBC against the original benchmarks used by this prior work. To this end, we targeted the iterative benchmarks from this prior work since HBC targets loops and not recursive functions. In more detail, we imported the same implementation of benchmarks previously evaluated by TPAL [42]. We have also used the same matrix generator implementation [41] for generating the different inputs of the benchmark *spmv*.

We have also evaluated HBC on a second set of benchmarks to further demonstrate HBC's effectiveness in targeting irregular workloads. To this end, we evaluated ten extra benchmarks whose computations are irregular. The rest of this subsection describes them. The first one is the benchmark *cg* from the NAS benchmark suite; we used its implementation in NPB3.0 [39]. We chose only *cg* from this suite because it is the only benchmark in the NAS suite whose input can lead to an irregular workload. In more detail, we used a non-synthetic and irregular input because most synthetic inputs of *cg* (including the one included in the suite) lead to a regular workload. We used the input cage15 [50], which is a real-world matrix from the Univerity of Florida sparse matrix collection [12]. The second benchmark we targeted is *mandelbulb* [52], which is an extension of *mandelbrot* to handle 3D inputs.

Benchmark	Source	Input	Regularity
OpenMP pragmas are generated by programmers			
mandelbrot	TPAL [40, 42]	512 × 1024 × 40k	irregular
spmv-arrowhead		150 million rows 450 million nonzeros	irregular
spmv-powerlaw		16.7 million rows 402 million nonzeros	irregular
spmv-random		6 million rows 600 million nonzeros	regular
floyd-warshall		4k × 4k	regular
kmeans		10 million elements	regular
plus-reduce-array		100 billion elements	regular
srad		10k × 10k	regular
mandelbulb	3D Mandelbrot [52]	100 × 200 × 300 × 400	irregular
cg	NAS [39]	cage15 [50]	irregular
OpenMP pragmas are automatically generated			
ttv	TACO [28, 29]	nell-2 [46]	irregular
ttm			irregular
bfs	GraphIt [54, 55]	Twitter [30]	irregular
cc			irregular
pr			irregular
cf		LiveJournal [12]	irregular
pr-delta			irregular
sssp			irregular

Table 1. The benchmarks used in this paper, with input used, their regularity, and if they were also evaluated by TPAL.

We have also evaluated eight more benchmarks from two domains: sparse tensor computation and graph analytics applications. We targeted these two domains because applications in these domains tend to be highly irregular due to their computational sparsity. For the sparse tensor computation, we imported *TTV* and *TTM* from TACO [29], a domain-specific language for sparse tensor algebra. TACO asks programmers to supply an index expression of a tensor algebra kernel and specify each tensor's storage format (*dense* or *sparse*) in the index expression provided. This high-level expression is compiled to C/C++ code where the main kernel computation is a loop nest, within which all loops are DOALL. TACO disables nested parallelism by annotating OpenMP pragmas only on the outermost loop. We manually changed the C/C++ code generated by TACO to have multiple versions of the same benchmark by enabling (or disabling) parallelization of the nested loops. This enabled us to evaluate how both an OpenMP compiler and HBC handle nested parallelism. We used the same inputs that were used to evaluate TACO [29]; these inputs are obtained from the

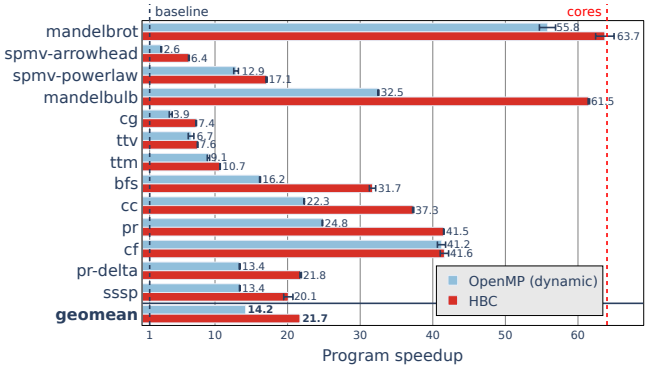

Figure 4. 64-core evaluation comparing OpenMP dynamic scheduling and HBC over irregular workloads.

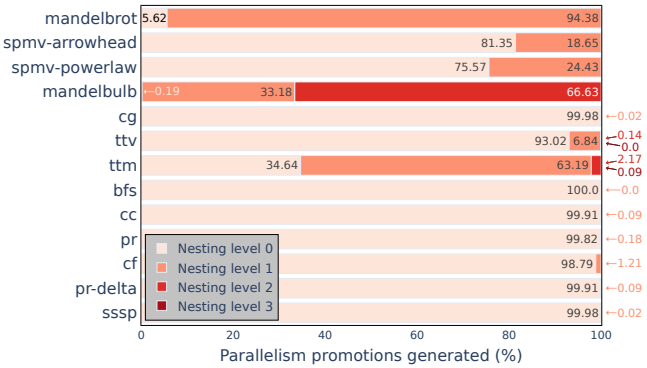

Figure 5. Parallelism is generated at different loop nesting levels.

FROSTT Tensor Collection [46]. We use the storage format of *dense* for the first dimension of the input tensor and *sparse* for the rest of the dimensions.

For graph analytics applications, we imported *bfs*, *cc*, *pr*, *cf*, *pr-delta* and *sssp* from GraphIt [55], a domain-specific language for writing graph analytics. GraphIt enables programmers to write high-level computation with the freedom to apply various optimizations such as parallelization, cache partitioning, and data layout optimizations. GraphIt compiler transforms high-level graph computations to C/C++ code with OpenMP pragmas. The main kernel from the above graph analytics benchmarks is often a DOALL loop that goes through every node of a graph and applies an update function on outgoing neighbors of that node. We enabled parallelization for all benchmarks and used *DensePull* as the direction of applying the update function. We used social network graphs Twitter [30] and LiveJournal [12], which exhibit high irregularity for the input data. These graphs are the same inputs that the authors of GraphIt have used to evaluate their work.

Platform. All of our results are computed using an AWS instance that runs Linux kernel 5.19.0 equipped with an Intel Xeon Platinum 8375C processor featuring 64 cores over two sockets running at 3.0 GHz, with 2 MiB L1i, 3 MiB L1d, 80 MiB L2 and 108 MiB L3. Both hyperthreading and turbo-boost are disabled throughout the evaluation. We set the heartbeat rate to 100 μs following the tuning process proposed in TPAL [42]. We report the median result over 100 runs.

6.2 HBC Outperforms OpenMP for Irregular Workloads

HBC binaries outperform OpenMP for all irregular workloads. Fig. 4 shows the speedups of these two sets of binaries on 64 cores. The speedup (on average) increases from 14.2× to 21.7× when switching from OpenMP to HBC. This shows HBC's effectiveness in generating optimized parallel binaries that benefit from heartbeat scheduling. HBC outperforms OpenMP by adapting the parallelism at run-time, following the heartbeat approach. Fig. 5 shows that HBC enables the program to generate parallelism at different loop nesting levels upon a heartbeat. This suggests that using a static granularity decision is sub-optimal as the best granularity depends on the input data. This also demonstrates that the granularity decisions can be offloaded to the compiler and runtime, reducing the burden on programmers.

6.3 HBC Automates TPAL's Prior Work

HBC automates the code generations and optimizations done manually in the TPAL work [42]. Hence, it is important to understand whether the automation performed by HBC delivers the same binary quality that was manually obtained by TPAL.

Fig. 6 shows the speedups of the HBC and TPAL binaries in our testbed. **HBC automatically delivers comparable or superior performance compared to the state-of-the-art and manually-generated code implemented using TPAL**. The TPAL-based heartbeat manual transformation uses rollforwarding to switch between the serial version of the code and rollforwarded code to promote parallelism. Its runtime reserves an extra interrupt ping thread to signal the arrival of heartbeats to all active worker threads. Similar to our solution, TPAL inserts the promotion handler at the end of each loop body. TPAL binaries perform chunking on all leaf loops using a static chunk size determined with a tuning process and defined at compile time per benchmark. These per-benchmark manual tunings performed in TPAL's binaries are fully automated in HBC (on top of the code parallelization, generation, and optimization). Next, we discuss in detail why HBC outperforms TPAL on some benchmarks.

HBC generates more parallelism. HBC obtains higher speedups than TPAL for *kmeans* (+13.7%), *mandelbrot* (+24.4%) and *srad* (+44.8%) because HBC runs in parallel all three tasks generated per promotion (two loop-slice tasks and a leftover task), while TPAL only runs two tasks in parallel, placing the third one in its critical path. According to the authors, this was done because running the leftover task in parallel

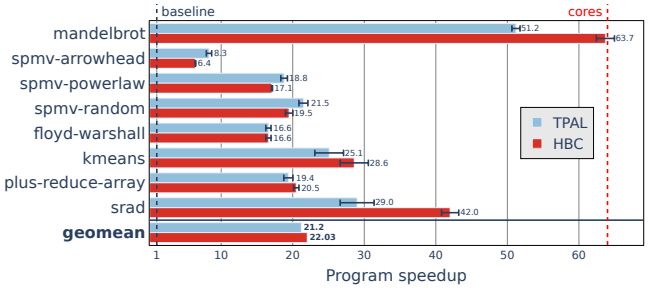

Figure 6. HBC automatically delivers comparable performance compared to the manually-generated TPAL binaries. These are the loop-based benchmarks used in [42] running on 64 cores.

would have further increased the already high time it took them to manually generate the code. The limitation was due to having the leftover task still referencing the execution environment (e.g., stack) of the parent task; in other words, the leftover task in TPAL did not have a complete closure. Notice that in the worst case, the number of possible static leftover tasks grows quadratically with the number of nested loops; writing all of them manually with their closure is impractical. Because HBC is fully automatic, it is able to automate the closure generation of all possible leftover tasks, which unlocks an extra level of parallelism. This additional level of parallelism becomes important when the time it takes to sequentially execute all leftover tasks is non-negligible, which is the case for *kmeans*, *mandelbrot* and *srad*.

HBC is penalized for chunk size transferring. In all *spmv*-based benchmarks, HBC performs worse than TPAL. The worst is *spmv-arrowhead*, which shows a 22% slowdown. This slowdown is caused by continuously tracking and updating the remaining chunk size before making a poll. This overhead is in the critical path for this input, starting from the second row. TPAL relies on interrupts and thus does not generate this extra overhead. The chunk size transferring cost decreases for HBC once there are more non-zeros to be processed per row, as shown in *spmv-powerlaw* (-9%) and *spmv-random* (-9%).

6.4 HBC Overhead Analysis

Next, we analyze the extra work performed by the HBC binaries compared to the baseline. To this end, we compile all the loops without enabling parallelism promotion and thus avoid the cost of task scheduling. Since now the program runs sequentially, all extra work that the HBC binaries perform comes from loop outlining, closure generation, loop chunking, promotion insertion, chunk size transferring and polling overhead. Fig. 7 shows the overhead results and their breakdowns.

Only *spmv-arrowhead* and *spmv-powerlaw* have significant overhead (others have less than 10% overhead). The extra work added by HBC for all benchmarks includes the

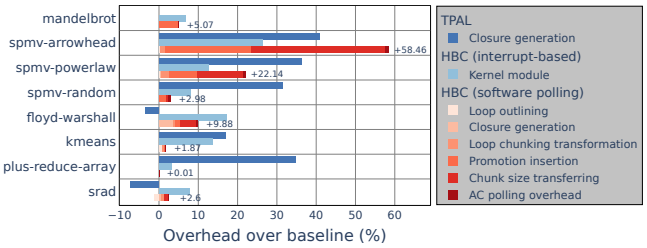

Figure 7. Overhead of HBC (w/ and w/o software polling) and TPAL.

parent loop passing via memory the iteration space for the nested loop to run. This overhead is negligible when the invoked loop runs many iterations, which is true among all these benchmarks. HBC inserts a promotion handler call at the end of the body of a loop, followed by a check on the call's return value. This does not generate much overhead because HBC applies chunking to all leaf loops. Hence, promotion insertion overhead becomes insignificant compared to the amount of work performed inside a chunk.

When the loop getting invoked repeatedly runs only a small number of iterations, the overhead generated by HBC cannot be amortized and becomes significant. This is because for *spmv-arrowhead* (+58.46%) and *spmv-powerlaw* (+22.14%), the binary needs to do chunk size transferring each time a leaf loop is invoked, while only processing a few non-zero elements. For the same reason, the overhead of promotion insertion at the outer loop accumulates quickly when a small leaf loop is invoked, and gets added into the critical path.

6.5 Software Polling is as Good as Hardware Interrupts

HBC supports both software polling and hardware interrupts for handling heartbeats. The former is the default one and it is the mechanism used for all results shown in this paper outside this sub-section. We first analyze the overhead of software polling and then compare it with the interrupt-based mechanism implemented using rollforwarding.

Software polling overhead. Software polling has the potential to broaden the adoption of heartbeat scheduling as it does not require any hardware or OS support. The main problem with software polling is its overhead. We show, however, that adding a few optimizations (e.g., loop chunking) significantly reduces the polling overhead to the point that software polling becomes an interesting design choice. To show this, we start with an unoptimized implementation of a simple algorithm for software polling, showing its high overhead. Then, we slowly improve its quality, leading towards a more and more optimized algorithm, until we reach the final algorithm (with low overhead) described in §5.1, which is the default implementation of HBC.

Our simplest implementation disables the loop chunking transformation in HBC. Hence, a poll is performed at every loop iteration. The "No chunking" bar of Fig. 8 shows

the overhead (in clock cycles) obtained by this simple implementation compared to the same baseline of Fig. 6. This is computed by running the generated code without doing promotions; hence, the execution stays sequential even if we perform the polls. The overhead is significant and completely erases the benefits of parallelism, causing a 7.5x slowdown.

Our second implementation adds loop chunking using static chunk sizes used in TPAL [42]. Hence, a poll is performed per chunk. The bar "Static chunking" of Fig. 8 shows that the overhead of this algorithm is significantly lower.

Finally, we measured the overhead of polling generated by the algorithm described in §5.1 where the chunk size is decided by our runtime. This is the bar called "Adaptive Chunking" of Fig. 8. This low overhead led to the speedups shown in Fig. 6.

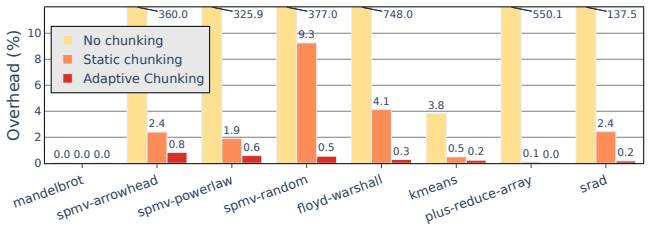

Figure 8. Software polling overhead with different chunking mechanisms. Both static and adaptive chunking significantly reduce overhead, with adaptive performing best.

Software polling and interrupt-based solutions are comparable. Prior work on heartbeat scheduling relied on hardware interrupts. HBC implements this solution as well as software polling, so for the first time, these techniques can be compared within the context of heartbeat scheduling.

TPAL work relies on a dedicated thread to send heartbeat interrupts to worker threads. HBC is the first compiler that automated the code generation needed by this technique (§4). To measure the efficiency of this solution, we used a user-space thread to send heartbeat interrupts as done by prior work [42]. Results obtained with this technique are shown by the bar "Interrupts (ping thread)" of Fig. 9. This figure shows that software polling boosts the speedup obtained by this prior work technique from 17.7× to 22.0×. This is because of the high signaling overheads created by the interrupt ping thread, making it unable to deliver heartbeats at a rate that matches the desired one. This results in missing up to 45% of the heartbeats (and therefore generating less parallelism).

Because of this high overhead, we have implemented a custom OS support to reduce the latency of delivering heartbeats (§5.2). This is the bar "Interrupts (kernel module)" of Fig. 9. While this new implementation is better than the one adopted by prior work, its performance is still compatible to that of software polling. This suggests that heartbeat scheduling can be embedded in programs without special OS support, increasing adaptability.

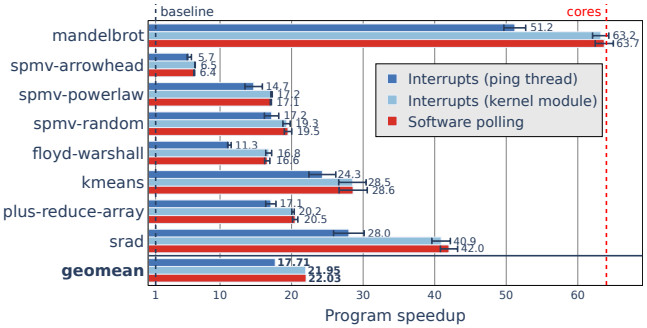

Figure 9. Software polling is as good as interrupt-based mechanisms.

Why is software polling comparable? To understand why software polling is comparable to the interrupt-based solution adopted in prior work, we compared the overhead of these two techniques. Fig. 7 shows such a comparison where promotion is disabled for both techniques; hence, the promotion cost and task scheduling cost are not included in the overhead.

Software polling pays (on average) less overhead than the best interrupt-based technique (the one with custom OS support). This is because the overhead for the latter per heartbeat is almost two orders of magnitude compared to a single poll. For each heartbeat, it can take 3800 cycles (§5.2). Instead, by our measurement, a poll takes 50 cycles. Ideally, a single poll per heartbeat is enough for the software polling technique. Even when 10 polls are performed per heartbeat, the total cost is still an order of magnitude less than the cost of an interrupt.

6.6 Chunking Needs to be Adapted at Runtime

The best chunk size is input-dependent and therefore it needs to be adapted at run-time. Next we are going to use *mandelbrot* as an example of code that highlights this need.

The need for adapting. Fig. 10 shows the execution time of *mandelbrot* using heartbeat scheduling on 64 cores with two different inputs, one with high latency (input 1) and the other with low latency (input 2). As we increase the static chunk size used from 2^0 to 2^9, input 2 performs better while input 1 performs worse. Therefore, the best chunk size

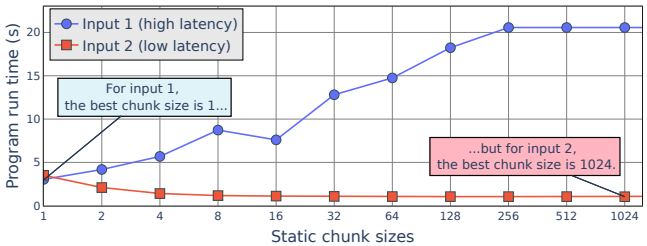

Figure 10. Optimal chunk size for *mandelbrot* is input-dependent.

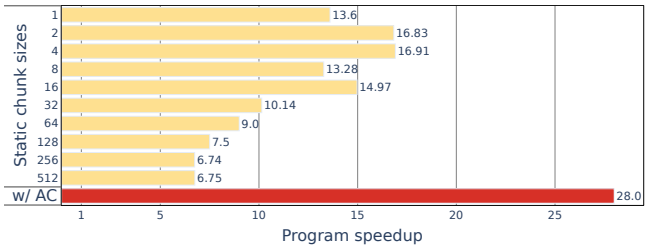

Figure 11. Speedup of invoking mandelbrot 10 times with different inputs, using static chunk size versus adapting chunk size at run-time.

setting for *mandelbrot* is input-dependent, and there is no single chunk size setting that is optimal across all inputs.

A common scenario in many applications is that an important loop gets invoked repeatedly and possibly with different inputs. To further show the limitations of setting the chunk size statically, we studied *mandelbrot* in this scenario by invoking its main loop 10 times, using input 1 and 2 five times each. We measured the time it takes to sequentially run these invocations (compiling the code with clang) and we consider this time to be the baseline for this experiment. The speedup obtained by HBC forcing a static chunk size or by adapting it at run-time are shown in Fig. 11. Adapting the chunk size at run-time boosts the speedup from 17× (static chunk size set to 2) to 28×, an increase of 64%.

The benchmark *mandelbrot* is just an example that shows the need for adapting the chunk size at run-time. To show this, let us now observe *spmv* with different matrix inputs and use the number of non-zeros per row to show the impact of having different latencies per loop iteration. AC is included in our HBC solution and is needed when there are different latencies between loop iterations. This is because the code needs to perform the right amount of polls to minimize their overhead while avoiding missing heartbeats. The higher the latencies, the smaller the chunk size needs to be. Fig. 12 shows how the chunk size changes in reaction to loop iteration latency.

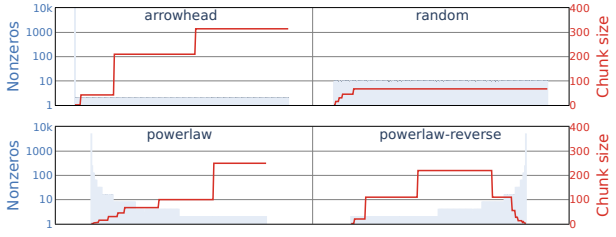

Figure 12. Visualization of Adaptive Chunking.

Adapting chunk size requires the right target. To adapt the chunk size of a loop, our runtime implements a sliding window algorithm (§5.1), controlled by two parameters: target polling count and window size. All HBC results shown in this paper use 8 for both the target polling count and the window size for all loops in all benchmarks, which we will justify next.

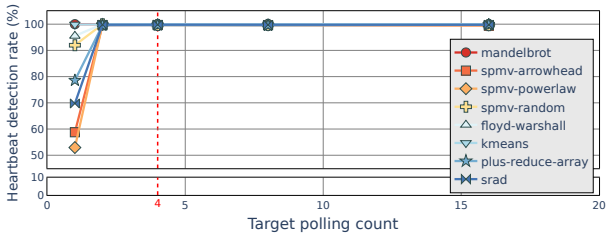

Figure 13. Heartbeat detection rate via AC. A target polling count value of 4 is efficient in capturing almost all heartbeats.

Target polling count. Our primary goal is to avoid missing heartbeats while minimizing polling overhead, thus we compare how many heartbeats are detected to the total number of heartbeats generated. We perform this experiment at different target polling ratios. Our results in Fig. 13 show that a too-low target polling count results in missing a significant number of heartbeats (almost 50% for *spmv-powerlaw*), which leads to less parallelism. On the other hand, having a too-high target polling count results in extreme polling overhead. Setting this value to 4 captures over 99% of the heartbeats while paying a small overhead. Similar results are obtained for other benchmarks. Hence, we used the target count 4 for all loops in all benchmarks for every result shown in this paper.

Window size. Theoretically, this parameter could impact the reaction speed of changing the chunk size at run-time to changes of the latency of loop iterations. In practice, our results show negligible impact for this parameter on all benchmarks used. Hence, we used window size 8 (anything ≥ 2 would have worked fine) for all results shown in this paper.

6.7 Manual Granularity Control for OpenMP Compilers

All OpenMP-related results described in prior sub-sections are obtained by parallelizing only the outermost loops of a benchmark (and by using the default chunk size for each parallel loop, which is one). This is recommended as a good practice to control the scheduling overhead. However, if an OpenMP compiler can perform granularity control automatically (like HBC), then a better solution for programmers is to expose the parallelism of all DOALL loops of a benchmark without worrying about the scheduling overhead. This is what we did when we used HBC in the prior sub-sections.

OpenMP compilers rely on the programmer to make granularity control decisions such as determining which loops to parallelize and specifying the chunk size of a loop being parallelized. To show that the decisions that OpenMP programmers of our target benchmarks are reasonable, this subsection performs the following experiments. We changed

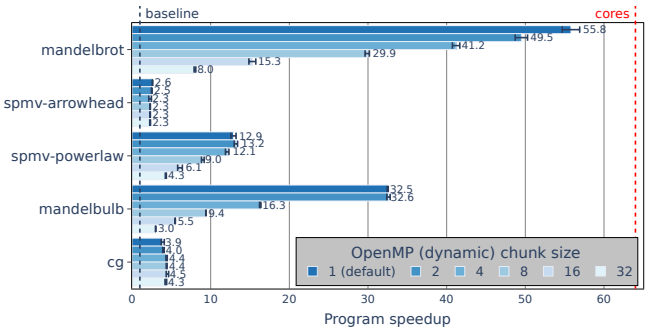

Figure 14. 64-core evaluation of OpenMP dynamic scheduling using varying chunk sizes. Only the outermost loop is parallelized.

(manually) the selection of which loops to parallelize as well as their chunk size while using the OpenMP compiler to see if these changes can improve performance. We performed these experiments on all manually implemented benchmarks, where programmers have the full control over where and how OpenMP pragmas are generated. Our results show that both tuning the chunk size and parallelizing all DOALL loops (instead of parallelizing only the outermost loops) degrade the performance.

Tuning the chunk size. OpenMP programmers can manually perform granularity control of the parallelism of their code by changing the chunk size of a parallel loop. While tuning the chunk size for performance is often input-dependent and labor-intensive (and therefore not ideal), it is important to know how much performance can be gained by it (to understand how well the default chunk size performs). To this end, we run a sensitivity analysis over the chunk size of the OpenMP dynamic scheduler for all manually implemented benchmarks. As shown in Fig. 14, all benchmarks get their performance degraded when keep increasing the chunk size except for one benchmark, *cg*, whose performance is slightly improved. The worse performance is because when tasks get coarsened, chunking results in less balanced execution for irregular workloads.

Parallelizing all DOALL loops. The OpenMP results described in prior sub-sections have been obtained by parallelizing only the loops that the original author of the benchmarks has parallelized (which is only the outermost DOALL loops). However, some nested loops of the target benchmarks could be parallelized as DOALL as well. Enabling their parallelism (by adding more OpenMP pragmas) leads to finer-grained parallelism. To understand how OpenMP compilers handle more (fine-grained) parallelism, we ran an experiment where we exposed fork-join parallelism for all DOALL loops (as we did for HBC) for all manually implemented OpenMP benchmarks. This is achieved by explicitly invoking `omp_set_max_active_levels` routine at the beginning

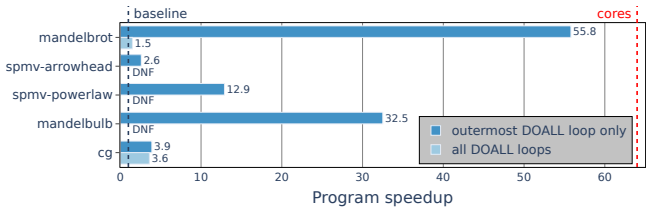

Figure 15. 64-core evaluation of OpenMP dynamic scheduling (using default chunk size) parallelizing the outermost loop only versus all DOALL loops. DNF means the program did not finish within the allowed time frame (2 hours) or crashes.

of the code and tagging all DOALL loops with OpenMP pragmas (using the *dynamic* scheduler and its default chunk size) of all loop nests. As shown in Fig. 15, all benchmarks, except for *cg*, have their performance significantly degraded when all DOALL loops are parallelized. *spmv-arrowhead* and *spmv-powerlaw* did not finish in time (2 hours). *mandelbulb* crashed because the OpenMP runtime failed to allocate necessary resources for 64 workers. The performance degradation is because enabling nested parallelism by parallelizing all DOALL loops in OpenMP generates too many tasks, which overloads the system and wipes out the benefit of parallelism.

6.8 When Heartbeat Scheduling is Inefficient

We compared HBC and OpenMP for regular benchmarks to understand whether heartbeat scheduling can become the solo policy. HBC performs worse than OpenMP in this case, as shown in Fig. 16, because heartbeat scheduling incurs extra overhead that is not justified for workloads that have well-balanced loop iterations. The only exception is *kmeans*, which HBC outperforms OpenMP static scheduler by more than 50%. HBC obtains this performance gain because it is able to reduce an array of elements between all tasks in parallel. Instead, the OpenMP implementation performs the reduction operation over an array sequentially by the main thread, and this adds to the critical path of the computation. We used the unmodified OpenMP implementation *kmeans* from the Rodinia benchmark suite [10], which is the same implementation used by TPAL [42].

The static policy outperforms all dynamic policies (including heartbeat) for most regular benchmarks we studied. This is because a static decision about how to parallelize

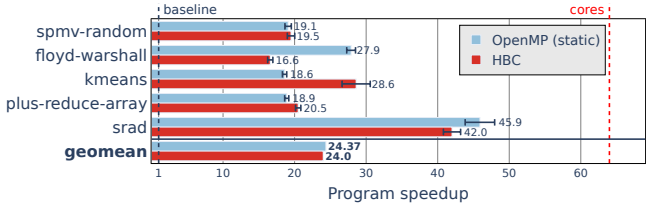

Figure 16. 64-core evaluation comparing OpenMP static scheduling and HBC over regular workloads.

the code generates minimal run-time overhead. Therefore, an ideal compiler should include both heartbeat and static scheduling.

7 Related work

Lazy scheduling and clone optimization. Lazy scheduling (LS) dates back to the proposal for lazy task creation of Mohr et al. [35], and was later adapted to the specifics of the work-stealing scheduler [20], resulting in the *clone optimization*. Our HBC runtime uses the clone optimization to avoid paying synchronization costs (e.g., the execution cost of atomic instructions) between tasks that are executed by the same thread. For HBC, in more detail, when a heartbeat happens at time t, while a task k is executing, k splits into three tasks that cumulatively perform the same computation that k still had to do to complete at time t. These three tasks execute in parallel and the continuation of k (the code to execute after k ends) can execute only when all three tasks end, which requires the three tasks to synchronize. However, when the three tasks are executed by the same thread that was executing k, they will execute sequentially and in the right order (thanks to the thread-local deque). In this case (the fast path), our runtime avoids performing the synchronization. However, if one of these three tasks is stolen by another thread and therefore it will execute on another core, then our runtime performs the necessary synchronization (the slow path).

Other instantiations of LS include backtracking-based load balancing [24] for recursive programs, library-based implementations of work stealing [17, 53], and lazy binary splitting [47, 49] for parallel loops.

Granularity control. A classic alternative to LS and heartbeat scheduling is granularity control (GC), a family of approaches that operate in a proactive manner to amortize task overheads. Like with LS, in GC, the program switches between serial and parallel modes of execution. However, switching in GC is guided by *predicted* amounts of *future* of work (LS and heartbeat scheduling are guided by *measured* amounts of *past* work). Manual granularity control [27] remains commonplace in spite of its limitations [47]; there have also been various proposals for automatic granularity control [15, 26, 32, 38, 44, 51]. Oracle-guided GC [2, 3] is the first to be backed by formal guarantees that bound task-related overheads and guarantee preservation of parallelism. However, these bounds require certain assumptions on the dynamic behavior of the program, which may be difficult to know in general, and the approach is not fully automatic. In particular, it requires application programmers to write annotations at fork points in the program that specify *abstract cost functions*, which require manual effort and are sometimes not practical.

Prior work has performed granularity control applied to a single program to dynamically adapt on changes to the hardware resources available while the parallel program executes (e.g., due to having multiple programs running on the same machine at the same time) [9, 16, 25, 45]. Compared to heartbeat scheduling, these approaches react to change in available resources rather than changes of available parallelism of the target program.

Compiler support for parallelism. Tapir [43] is a recent proposal to embed fork-join parallelism into the LLVM IR. The motivation is to unlock conventional middle-end optimizations (i.e., optimizations that target LLVM IR code) in LLVM to work within parallel constructs (e.g., loop invariant code motion across parallel loops). These optimizations are otherwise only applicable to serial regions of programs. Although our HBC extends the LLVM IR, our focus is to enable granularity control rather than to unlock existing optimizations of LLVM's middle end. Finally, notice that HBC can be extended to target TAPIR rather than LLVM IR. HBC and Tapir should compose well.

8 Conclusion

Obtaining efficient parallel execution still requires programmers to manually control the parallelism granularity of their programs. This leads to either platform-specific and hard-to-maintain codebases or inefficient programs. Heartbeat scheduling can solve this problem, but it requires the software to fit to an unconventional structure. This paper introduces HBC, the first compiler that automatically transforms C/C++ programs with nested fork-join constructs into such unconventional structure, unlocking heartbeat scheduling for a wide programmer base. HBC outperforms the clang-based OpenMP compiler for irregular workloads, where even programmers (not tools) struggle to optimize.

Acknowledgements

We thank members of the ARCANA Lab for their support and feedback on this work. We also thank the anonymous reviewers for their insightful comments and feedback, which made this work stronger, and especially Jean-Pierre Lozi, who significantly helped finalize the writing of this paper. This effort is based upon work supported by the U.S. Department of Energy under contract number DE-SC0022268. It is also based upon work supported by the National Science Foundation under Grants CCF-1901381, CCF-2107241, CCF-2115104, NSF-2119069, CCF-2119352, NSF-2107042, NSF-2028851, and NSF-1908488, and via the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration, by the U.S. Department of Energy, Office of Science, under Contract DE-AC02-06CH11357.

References

[1] Umut A. Acar, Arthur Charguéraud, Adrien Guatto, Mike Rainey, and Filip Sieczkowski. Heartbeat scheduling: Provable efficiency for nested parallelism. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 769–782, 2018.

[2] Umut A. Acar, Arthur Charguéraud, and Mike Rainey. Oracle scheduling: Controlling granularity in implicitly parallel languages. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 499–518, 2011.

[3] Umut A. Acar, Arthur Charguéraud, and Mike Rainey. Oracle-guided scheduling for controlling granularity in implicitly parallel languages. *Journal of Functional Programming (JFP)*, 26:e23, 2016.

[4] V Aho Alfred, S Lam Monica, and D Ullman Jeffrey. Compilers principles, techniques & tools, 2007.

[5] Sotiris Apostolakis, Ziyang Xu, Greg Chan, Simone Campanoni, and David I. August. Perspective: A sensible approach to speculative automatic parallelization. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 351–367, New York, NY, USA, 2020. Association for Computing Machinery.

[6] Simone Campanoni, Kevin Brownell, Svilen Kanev, Timothy M. Jones, Gu-Yeon Wei, and David Brooks. HELIX-RC: An architecture-compiler co-design for automatic parallelization of irregular programs. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 217–228, Piscataway, NJ, USA, 2014. IEEE Press.

[7] Simone Campanoni, Glenn Holloway, Gu-Yeon Wei, and David Brooks. HELIX-UP: Relaxing program semantics to unleash parallelization. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '15, pages 235–245, Washington, DC, USA, 2015. IEEE Computer Society.

[8] Simone Campanoni, Timothy Jones, Glenn Holloway, Vijay Janapa Reddi, Gu-Yeon Wei, and David Brooks. HELIX: Automatic parallelization of irregular programs for chip multiprocessing. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 84–93, New York, NY, USA, 2012. ACM.

[9] Simone Campanoni, Timothy Jones, Glenn Holloway, Gu. Y. Wei, and David Brooks. The helix project: Overview and directions. In *DAC Design Automation Conference 2012*, pages 277–282, June 2012.

[10] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, page 44–54, USA, 2009. IEEE Computer Society.

[11] Ding-Kai Chen, Hong-Men Su, and Pen-Chung Yew. The impact of synchronization and granularity on parallel systems. *ACM SIGARCH Computer Architecture News*, 18(2SI):239–248, 1990.

[12] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1), dec 2011.

[13] SK Debray, Manuel V Hermenegildo, and Pedro López García. A methodology for granularity-based control of parallelism in logic programs. *Journal of symbolic computation*, 21(4-6):715–734, 1996.

[14] Enrico A. Deiana, Vincent St-Amour, Peter A. Dinda, Nikos Hardavellas, and Simone Campanoni. Unconventional parallelization of nondeterministic applications. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 432–447, New York, NY, USA, 2018. ACM.

[15] A. Duran, J. Corbalan, and E. Ayguade. An adaptive cut-off for task parallelism. In *2008 SC - International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2008.

[16] Murali Krishna Emani, Zheng Wang, and Michael F. P. O'Boyle. Smart, adaptive mapping of parallelism in the presence of external workload. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–10, 2013.

[17] Karl-Filip Faxén. Wool-a work stealing library. *SIGARCH Comput. Archit. News*, 36(5):93–100, June 2009.

[18] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, jul 1987.

[19] Elias Frantar and Dan Alistarh. Sparsegpt: Massive language models can be accurately pruned in one-shot. 2023.

[20] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI*, pages 212–223, 1998.

[21] Souradip Ghosh, Michael Cuevas, Simone Campanoni, and Peter Dinda. Compiler-based timing for extremely fine-grain preemptive parallelism. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020.

[22] Milind Girkar and Constantine D Polychronopoulos. Automatic extraction of functional parallelism from ordinary programs. *IEEE transactions on parallel and distributed systems*, 3(2):166–178, 1992.

[23] Kyle C. Hale, Conor Hetland, and Peter A. Dinda. Automatic hybridization of runtime systems. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '16, page 137–140, New York, NY, USA, 2016. Association for Computing Machinery.

[24] Tasuku Hiraishi, Masahiro Yasugi, Seiji Umatani, and Taiichi Yuasa. Backtracking-based load balancing. *Proceedings of the 2009 ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, 44(4):55–64, February 2009.

[25] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. Dynamic knobs for responsive power-aware computing. *ACM SIGARCH computer architecture news*, 39(1):199–212, 2011.

[26] Lorenz Huelsbergen, James R. Larus, and Alexander Aiken. Using the run-time sizes of data structures to guide parallel-thread creation. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, LFP '94, pages 79–90, 1994.

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

[28] Fredrik Kjolstad. Taco github, 2017. https://github.com/tensor-compiler/taco.

[29] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–29, 2017.

[30] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, page 591–600, New York, NY, USA, 2010. Association for Computing Machinery.

[31] Viktor Leis, Kan Kundhikanjana, Alfons Kemper, and Thomas Neumann. Efficient processing of window functions in analytical sql queries. volume 8, page 1058–1069. VLDB Endowment, jun 2015.

[32] Hans-Wolfgang Loidl and Kevin Hammond. On the granularity of divide-and-conquer parallelism. In *Proceedings of the 1995 Glasgow Workshop on Functional Programming*, pages 1–10, 1995.

[33] Jiacheng Ma, Wenyi Wang, Aaron Nelson, Michael Cuevas, Brian Homerding, Conghao Liu, Zhen Huang, Simone Campanoni, Kyle C. Hale, and Peter A. Dinda. Paths to openmp in the kernel. In Bronis R. de Supinski, Mary W. Hall, and Todd Gamblin, editors, *SC '21: The International Conference for High Performance Computing, Networking, Storage and Analysis, St. Louis, Missouri, USA, November 14 - 19, 2021*, pages 65:1–65:17. ACM, 2021.

[34] Angelo Matni, Enrico Armenio Deiana, Yian Su, Lukas Gross, Souradip Ghosh, Sotiris Apostolakis, Ziyang Xu, Zujun Tan, Ishita Chaturvedi, David I. August, and Simone Campanoni. NOELLE Offers Empowering LLvm Extensions. In *International Symposium on Code Generation and*

Optimization, 2022. CGO 2022., 2022.

[35] Eric Mohr, David A. Kranz, and Robert H. Halstead Jr. Lazy task creation: a technique for increasing the granularity of parallel programs. In *Conference record of the 1990 ACM Conference on Lisp and Functional Programming*, pages 185–197, New York, New York, USA, June 1990. ACM Press.

[36] David Mosberger, Peter Druschel, and Larry L Peterson. Implementing atomic sequences on uniprocessors using rollforward. *Software: Practice and Experience*, 26(1):1–23, 1996.

[37] Niall Murphy, Timothy Jones, Robert Mullins, and Simone Campanoni. Performance implications of transient loop-carried data dependences in automatically parallelized loops. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, pages 23–33, New York, NY, USA, 2016. ACM.

[38] Joseph Pehoushek and Joseph Weening. Low-cost process creation and dynamic partitioning in Qlisp. In Takayasu Ito and Robert Halstead, editors, *Parallel Lisp: Languages and Systems*, volume 441 of *Lecture Notes in Computer Science*, pages 182–199. Springer Berlin / Heidelberg, 1990.

[39] Omni Compiler Project. Nas-c-openmp3.0, 2014. https://benchmark-subsetting.github.io/cNPB/.

[40] Mike Rainey. Tpal github, 2021. https://github.com/mikerainey/tpal/tree/master.

[41] Mike Rainey. Tpal matrix generator, 2021. https://github.com/mikerainey/tpal/blob/master/runtime/bench/spmv.hpp#L659.

[42] Mike Rainey, Kyle Hale, Ryan R. Newton, Nikos Hardavellas, Simone Campanoni, Peter Dinda, and Umut 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.

[43] Tao B. Schardl, William S. Moses, and Charles 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*, PPoPP '17, page 249–265, New York, NY, USA, 2017. Association for Computing Machinery.

[44] Kish Shen, Vitor Santos Costa, and Andy King. Distance: A new metric for controlling granularity for parallel execution. *Journal of Functional and Logic Programming*, 1999:1–23, 1999.

[45] Filippo Sironi, Davide B. Bartolini, Simone Campanoni, Fabio Cancare, Henry Hoffmann, Donatella Sciuto, and Marco D. Santambrogio. Metronome: Operating system level performance management via self-adaptive computing. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 856–865, New York, NY, USA, 2012. ACM.

[46] Shaden Smith, Jee W. Choi, Jiajia Li, Richard Vuduc, Jongsoo Park, Xing Liu, and George Karypis. FROSTT: The formidable repository of open sparse tensors and tools, 2017.

[47] Alexandros Tzannes, George C. Caragea, Rajeev Barua, and Uzi Vishkin. Lazy binary-splitting: a run-time adaptive work-stealing scheduler. In *Symposium on Principles & Practice of Parallel Programming*, pages 179–190, 2010.

[48] Alexandros Tzannes, George C. Caragea, Uzi Vishkin, and Rajeev Barua. Lazy scheduling: A runtime adaptive scheduler for declarative parallelism. *ACM Trans. Program. Lang. Syst.*, 36(3), sep 2014.

[49] Alexandros Tzannes, George C. Caragea, Uzi Vishkin, and Rajeev Barua. Lazy scheduling: A runtime adaptive scheduler for declarative parallelism. *TOPLAS*, 36(3):10:1–10:51, September 2014.

[50] A. van Heukelum, G. T. Barkema, and R. H. Bisseling. DNA electrophoresis studied with the cage model. *Journal of Compututational Physics*, 180:313–326, July 2002.

[51] Joseph S. Weening. *Parallel Execution of Lisp Programs*. PhD thesis, Stanford University, 1989. Computer Science Technical Report STAN-CS-89-1265.

[52] Daniel White. 3d mandelbrot generator, 2008. https://www.fountainware.com/Funware/Mandelbrot3D/Mandelbrot3d.htm.

[53] Chaoran Yang and John Mellor-Crummey. A practical solution to the cactus stack problem. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 61–70, 2016.

[54] Yunming Zhang. Graphit github, 2018. https://github.com/GraphIt-DSL/graphit.

[55] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. Graphit: A high-performance graph dsl. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–30, 2018.