Accelerating Real-Time Applications with Predictable Work-Stealing

Florian Fritz(✉), Michael Schmid, and Jürgen Mottok

Laboratory for Safe and Secure Systems - LaS^3,
Regensburg University of Applied Sciences, Regensburg, Germany
{florian2.fritz,michael3.schmid,juergen.mottok}@oth-regensburg.de

Abstract. Modern compute architectures often consist of multiple CPU cores to achieve their performance, as physical properties put a limit on the execution speed of a single processor. This trend is also visible in the embedded and real-time domain, where programmers are forced to parallelize their software to keep deadlines. Additionally, embedded systems rely increasingly on modular applications, that can easily be adapted to different system loads and hardware configurations.

To parallelize applications under these dynamic conditions, often dispatching frameworks like Threading Building Blocks (TBB) are used in the desktop and server segment. More recently, Embedded Multicore Building Blocks (EMB^2) was developed as a task-based programming solution designed with the constraints of embedded systems in mind.

In this paper, we discuss how task-based programming fits such systems by analyzing scheduler implementation variants, with a focus on classic work-stealing and the libraries TBB and EMB^2. Based on the state of the art we introduce a novel resource-trading concept that allows static memory allocation in a work-stealing runtime holding strict space and time bounds. We conduct benchmarks between an early prototype of the concept, TBB and EMB^2, showing that resource-trading does not introduce additional runtime overheads, while unfortunately also not improving on execution time variances.

Keywords: Real-time · Parallel programming · Work-stealing

1 Introduction

Modern processors rely on multiple cores and accelerating hardware to achieve their performance, as the execution speed of a single processor is physically limited by heat output and power draw. Consequently, developers have to explicitly parallelize their applications to achieve faster execution. Doing this manually can be tedious and error prone, therefore most industries have adopted dispatching frameworks to help with this process. The key idea behind these libraries is that programmers only declare how their work can be split up into individual tasks, while the framework's runtime schedules the work dynamically

A. Brinkmann et al. (Eds.): ARCS 2020, LNCS 12155, pp. 241–255, 2020.
https://doi.org/10.1007/978-3-030-52794-5_18

onto available system resources. Common examples are Intel's Threading Building Blocks (TBB), Microsoft's Parallel Patterns Library (PPL) and the Open Multi-Processing (OpenMP) standard. While task-based programming is a de facto standard in desktop applications and scientific computations, these frameworks are still uncommon for embedded systems due to their highly dynamic nature. A first contender entering this domain is Embedded Multicore Building Blocks (EMB^2), by specifically focusing on requirements like task priorities and static-memory allocation.

In sight of these various implementation variants, we investigate how different scheduling approaches and their concrete implementations affect their suitability for high performance embedded systems. Specifically, we take interest in using task-based programming to gradually parallelize individual real-time applications (Sect. 2). For this, we first recapitulate the commonly used work-stealing algorithm [3], draw implications for the predictability on embedded devices and discuss what challenges practical implementations face. As two examples, we study the internals of TBB, a representative of modern desktop implementations, and EMB^2, a contender specifically for the embedded space (Sect. 3). We find that work-stealing fits our use-case from a theoretical standpoint, but no implementation guarantees static memory usage and strict theoretical bounds. Following this, we introduce a novel resource-trading algorithm that enables us to implement a `C++` work-stealing library with static memory allocation and strict theoretical bounds (Sect. 4). Finally, we analyze how the three libraries perform on an octa-core ARM system. We conduct tests on both an isolated and multiprogrammed system using small problem sizes to evaluate how viable the frameworks are for embedded real-time applications (Sect. 5).

2 System Model and Requirements Context

We consider the acceleration of applications executing on an operating system (OS) scheduling threads onto a symmetric multicore processor using a real-time schedule, e.g. preemptive fixed priority scheduling. Each application[1] τ_i periodically performs work by releasing a sequence of threads to be executed by the OS. To be applicable for real-time and embedded use-cases, the applications must guarantee predictable time and memory bounds, as unexpected deadline misses are not acceptable.

Looking at a single application τ_i, we consider the process of gradually parallelizing it. For this a programmer can use the aforementioned dispatching frameworks to introduce sections of task-based parallelism to speed up compute intensive algorithms. This process breaks down the algorithms into a series of individual tasks, resulting in a Directed Acyclic Graph (DAG) where vertices denote computations and edges represent ordering constraints between them. Figure 1 shows a small example DAG, further discussed in the following section.

[1] Usually in real-time literature applications are referred to as tasks and threads are called jobs, however, this conflicts with the notation in task-based programming.

The dispatching framework creates a pool of P_i worker threads within the application τ_i to execute the tasks. A variety of techniques exist to schedule the DAG cooperatively among this thread pool. A simple method is to distribute the work statically. However, this can result in poor load balancing for irregular workloads, or in multiprogrammed systems if single workers are preempted. On the other hand, there exist a variety of dynamic dispatching algorithms which aim to improve load balancing. Variations include, among others, list scheduling (GNU OpenMP), work-sharing (EMB^2) and work-stealing (TBB).

From the point of view of the OS, applications are therefore following a fork-join structure. Each application starts with a single, main thread, until a parallel algorithm is executed on a pool of P_i worker threads (fork). When the workers have completed all tasks, the application joins back into a single serial thread. The fork-join procedure can be repeated multiple times. This leads to a two level scheduler: the OS preemptively schedules jobs onto physical processor cores, the application internally executes tasks cooperatively on the worker threads.

3 State of Task-Based Programming

In theory, list schedulers provide optimal bounds for distributing task-based programs. However, they suffer from memory contention in real implementations which can lead to bad average case runtimes. Because of this, work-stealing variants have prevailed instead. The idea is to associate each processor with its own deque (double ended queue) in which tasks are pushed and popped locally as long as the worker thread does not run out of tasks. The processor only interacts with other deques when it has no more work, in which case it tries to steal work from another processors deque.

Work-stealing therefore acts mostly decentralized, avoiding contention on shared data structures like a central task queue. This makes it perform well on modern microarchitectures, both in theory [3] and practice [2,10]. However, implementation details can significantly affect memory usage and runtime properties of work-stealing frameworks. To assess the use of such libraries in real-time systems, we first recapitulate the proven bounds of classic work-stealing and discuss how it maps to practical libraries implementing it. Next, we analyze the schedulers used in TBB and EMB^2, to show where practical task-parallel libraries are heading in general and in the embedded space.

3.1 Classic Work-Stealing

Blumofe proves the first good space and time bounds for fully strict computations [3]. Figure 1 shows part of a strict DAG with three potentially parallel computation strands shaded in gray. Dotted edges are called spawn edges and allow the control flow to diverge, curved edges are data dependencies between strands of execution, enforcing ordering of tasks. Parallel strands form a parent-child relationship, where a parent spawns a child. For a computation to be fully strict, data dependency edges must only go from child to parent.

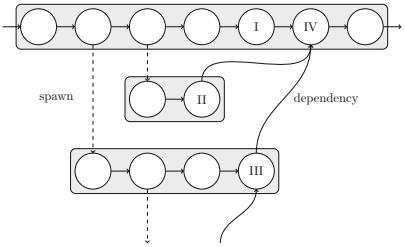


Fig. 1. DAG of computation in classic work-stealing

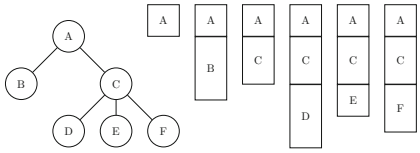


Fig. 2. Invocation tree (left) and resulting call-stacks (right)

While at first seeming restrictive, strict computations are those that are intuitively well-formed, i.e. spawning a child strand corresponds to an asynchronous subroutine call. Listing 1 shows an example program that could result in a DAG similar to Fig. 1, spawning two potentially parallel sub procedure calls in lines 5 and 6, then synchronizing to wait for their completion in line 8. The API shown in the example is known as nested fork-join parallelism [7] and implemented in libraries like Cilk [5] and TBB [8]. Interpreting parallelism as asynchronous subroutine calls allows us to view the execution as walking over an invocation tree (Fig. 2), where a serial execution is a preorder walk of the tree and parallel execution schedules walk the tree asynchronously[2].

In the randomized work-stealing algorithm there are four main events when worker threads interact with the scheduler and diverge from this serial execution [3]: (1) spawning, i.e. pushing a task to the bottom of their local deque; (2) enabling a blocked task, i.e. being the last predecessor in the DAG to finish; (3) executing the last vertex in a string of execution; (4) stealing, i.e. running out of local work and stealing uniformly at random from the top of other deques.

To analyze the time and space requirements of invocation trees under work-stealing variants, three properties are of interest: (1) S_1 is the space required for a serial execution, which is the peak memory usage when run on a single thread. In the invocation tree on the left side of Fig. 2, this corresponds to the deepest stack on the right side (A, C, D); (2) T_1 is the total work in the DAG and equals execution time on a single thread; (3) T_∞ is the critical, or longest path in the DAG. It is equal to the execution time on unlimited workers, as the ordering constraints of the critical path force it to execute serially.

[2] Fork-Join parallel APIs and invocation trees hide details compared to fully-strict DAGs and have not the same expressive power. However, we use them to simplify our arguments and all proofs hold on the DAG, too.

```
int fib(int n) {
  if (n <= 1) return n;

  int a, b;
  spawn([&]() { a = fib(n - 1); });
  spawn([&]() { b = fib(n - 2); });

  sync();
  return a + b;
}
```

Listing 1: Example of a nested fork-join parallelism API.

Following the above rules and definitions, one can show that the active part of the parallel invocation tree, i.e. all tasks that are executing, waiting for predecessors or enqueued in a deque, have the busy-leaves property [3]: each leaf of the active invocation tree has a processor working on it. When run on P_i worker threads, at most P_i branches of the tree can be active, as each branch has a leaf and therefore one of the processors working on it. Each branch uses a maximum of S_1 memory, leading to the space bound in Eq. (1).

$$S_P \leq S_1 P_i \tag{1}$$

The proof leading to the time bound of randomized work-stealing uses a more complicated delay sequence argument [3]. Intuitively, the proof shows that it is very unlikely that there are many steal attempts without stealing a task that makes progress on the critical path. The expected number of time steps used to perform steals is $T_\infty P_i$. To finish a computation the steals $T_\infty P_i$ and the work T_1 are added up and divided by the number of workers, leading to the expected time bound in Eq. (2). Additionally, the execution time can be bounded to a fixed value with a high probability. Similar bounds hold for multiprogrammed environments [1].

$$T_P = O(T_1/P_i + T_\infty) \tag{2}$$

A framework holding the busy-leaves property and the time bounds can therefore be practical for a soft real-time application requiring a bounded memory footprint and a certain quality of service.

3.2 Work-Stealing Implementation Challenges

The main challenge for work-stealing implementations is to adapt the programming language's serial semantics to a parallel execution, as compiled languages like `C/C++` are designed with a stack based, linear execution in mind. For example while work-stealing both D and F in Fig. 2 can execute concurrently, having one thread observe the stack (A, C, D) and a second one observe (A, C, F) at the same time. Building such a diverging stack is known as a cactus-stack in language design. In order to keep time and space bounds, as well as stick close

to serial semantics, a work-stealing scheduler has to maintain a cactus-stack and make sure that tasks are never blocked by the implementation. If for example a thread executing Listing 1 encounters the sync (vertex I in Fig. 1) and has to wait for children to complete (vertex II and III), it must make the rest of the function (vertex IV) executable by another worker. Additionally, implementations must amortize their scheduling overheads against the work T_1/P and span T_∞ of the computation, e.g. if the stealing process incurs overheads they must be considered as a factor in the T_∞ term.

To understand how implementations can approach this challenge, we present some common variants found in frameworks below.

Heap Allocated Stack Frames – One solution to build a cactus-stack is to allocate each function frame on the heap instead of the stack. This allows for non-blocking execution and strict space bounds, as only the active stack frames of the invocation tree are kept allocated and stacks are independent of worker threads. Cilk [5] implements this principle and therefore holds both theoretical time and space bounds. The drawback to this approach is that it requires compiler support or exhaustive manual code transformations to adopt a heap-stack-frame calling convention, making it less interoperable with existing software and incurring overheads on every function call.

Execute on Worker Stacks – Another approach is to execute tasks directly on the linear stacks of each worker thread. This method requires no special language constructs, but problems occur when a synchronization point is reached. If a thread reaches e.g. the `sync()` in line 8 of Listing 1 and has to wait for children to finish, the function frame lies on top of the stack. To stay greedy, the worker has to start stealing, but executing the stolen task directly on top of the worker's stack leads to two problems: (1) the stack can grow unbounded, as the worker can pile up multiple stolen stacks, (2) the task waiting at the `sync()` is blocked until all stolen tasks above have finished, as it is buried in the call stack.

This technique therefore violates both strict time and space bounds. To prevent unbounded space usage sometimes **restricted stealing** approaches are used, limiting work-stealing attempts to a subset of tasks. Examples for this are leapfrogging [4,15] and depth-restricted stealing [8]. However, restricting steals can potentially lead to near serial execution times [13].

One Stack per Steal – To keep the portability of execution on regular stacks but not block in tasks, an option is to always execute a stolen task on a new stack. This technique holds strict time bounds, but uses $S_1 N_{steals}$ memory proportional to the number of active stolen tasks. As with restricted stealing, some implementations like Cilk Plus limit the parallelism in favor of bounded memory usage, setting a fixed amount of stacks and stopping stealing if they are exhausted.

Memory Mapped Cactus-Stack – This solution uses an OS modification for thread local memory mapping, allowing the runtime system to give each worker thread the illusion of having a linear stack [9], holding strict bounds.

The main drawbacks are frequent memory mappings and either OS support or tricks around processes and virtual memory.

3.3 Case Study: TBB and EMB²

Looking at actual implementations of task-based programming, we first examine TBB, the industry leading task-parallel library in `C++`. The framework offers a low level fork-join task-parallel API, high level parallel patterns, concurrent data-structures and includes an optional scalable memory allocator. Overall, TBB's goals are to achieve high throughput by offering a composable, portable task-based API that does not require compiler support and can be gradually incorporated to existing applications.

Internally, TBB [8,14] resembles mostly classic work-stealing for scheduling tasks, with decentralized LiFo deques and randomized stealing. It uses a combination of the 'execute on worker stacks' and 'heap allocated stack frames' strategy for task execution. When using the high level fork-join APIs that are easily added to existing code (like in Listing 1), it executes the tasks on the worker threads, loosing strict bounds. Alternatively, one can re-write code with explicit task and continuation objects, manually building up a heap-allocated cactus-stack. This looses normal call-stack semantics, but in return mostly keeps classic bounds. The parallel patterns internally use this style for efficiency. In case a worker stack becomes too deep, TBB stops stealing, restricting parallelism in favor of application stability.

In contrast to this general purpose library, the EMB² [12] task scheduler specifically targets the embedded market. It is based on the Multicore Task Management API (MTAPI) [6] specification, an industry standard for lightweight task scheduling on resource constrained embedded systems with heterogeneous hardware. Specifically, EMB² implements a MTAPI standard compliant task scheduling environment in C, which can be used directly, but is also utilized by parallel patterns offered by EMB²'s high level `C++` API. The framework offers support for acceleration hardware, supports core affinity as well as task priorities and allocates all runtime resources exclusively during startup. Overall, EMB² offers a portable solution to dispatch tasks onto different components of a resource constrained system, acting more like a 'whole system scheduler' similar to an OS and is not restricted to strict fork-join parallelism on the CPU.

Looking at EMB²'s scheduling and task management, we only discuss execution on the CPU. Each thread is associated with multiple FiFo queues for different task priorities, with workers pulling tasks in either local-first or priority-first order. Task execution uses the 'run on workers stacks' technique and tasks are blocked in the stack while stolen tasks are executed above. Newly spawned tasks are distributed to threads in a round robin fashion. The combination of work-sharing and FiFo queues leads to a fair task execution, i.e. old tasks are executed first, ensuring that no single task is buried in queues. This fits the 'whole system scheduler' style of MTAPI, i.e. a system that continuously spawns mostly independent tasks needing to finish in a timely manner. However, no formal bounds can be provided and especially tree-like computations use much memory as the

schedule results in a breadth first execution of the invocation tree. To make use of the static resource allocation, the programmer has to manually find the maximum number of spawned tasks.

In summary, TBB sticks as close to classic work-stealing as possible, while focusing on portability and average system throughput. When necessary, it diverges from theoretical bounds in favor of a portable and simple implementation. This trend can be seen in most general purpose implementations. EMB^2 in contrast offers a task-based API, similar to TBB, but tunes its scheduler for fair task execution and offers specialized features like static resource allocation at startup.

4 Work-Stealing with Static Memory Allocation

EMB^2 shows that static, predictable resource usage and clean task-based parallel APIs are in demand on embedded platforms. However, their fair scheduling model is better suited for controlling the whole system rather than gradually introducing parallel sections in individual applications. Classic work-stealing on the other hand is a good fit for this purpose and can also ease the reasoning on upper memory bounds. A combination of work-stealing and static memory allocation at startup can therefore lead to predictable application behavior. Unfortunately, modern frameworks like TBB intentionally hurt the tight bounds of randomized work-stealing and make liberal use of a general purpose memory allocator. On a desktop machine occasional usage of more memory or longer execution times are well worth the trade-off. For an embedded system, in contrast, unpredictability can cause major issues.

To bring work-stealing closer to the embedded domain, we explore how a `C++` library implementation can provide tight and predictable memory bounds while also keeping the runtime properties of classic work-stealing and a natural fork-join API. The core of the prototype – called Predictable Parallel Patterns Library for Smart and Scalable Systems (P^{3}LS^3) – is a novel resource-trading scheme that integrates memory management into the stealing procedure. This allows the implementation to allocate all memory statically at startup, guaranteeing a maximum application footprint after a single, serial measurement run.

4.1 Resource-Trading Algorithm

Existing implementations like Cilk and TBB allocate all resources used during scheduling on the heap. This adds a multithreaded memory allocator as an abstraction layer to be considered in a pessimistic analysis. The different orders of allocation that can happen must be taken into account for exact memory requirements, as well as the sporadic work involved in balancing memory between threads or requesting new pages from the OS.

To avoid this issue, the resource-trading algorithm incorporates the balancing of memory into the stealing process. This amortizes the management overhead into the T_∞ term of the time bound and allows for a strict space bound. The

starting point is the maximum amount of resources R_S a serial invocation tree can allocate. Figure 3 shows R_S exemplary as the dark shaded areas, indicating the active part of the deepest invocation tree (we use R_S instead of S_1 to more accurately describe our implementation later on).

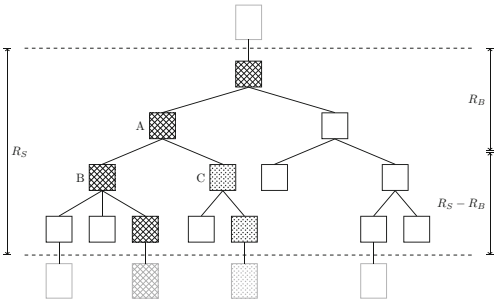


Fig. 3. Invocation tree with resource-trading

Resource-trading has the same linear growing bound $R_S P_i$ as work-stealing. Each worker thread is associated with R_S resources at startup, with no additional allocations during runtime. By proofing that a worker thread never runs out of resources with this initial configuration, the bound $R_S P_i$ follows trivially. Specifically, we show that a thread always starts stealing with R_S resources and these are enough to run until returning to the stealing state.

A serial execution of the invocation tree has by definition enough resources, therefore the interactions during the work-stealing algorithm are of interest. The critical point where a thread loses resources of its initial R_S pool are synchronization points where another thread stole part of the work and is not yet finished. Figure 3 shows this situation. The first, dark shaded thread eventually returns from task B, requiring task C to be finished before continuing working on task A. However, task C is currently being executed by the second, light shaded thread. To not idle the first worker has to start stealing. Unfortunately, all resources including A and upwards must be kept allocated, making the first thread lose R_B blocked resources.

To solve this, each thread trades in resources to compensate for potentially blocked resources of another worker on a steal. In the example in Fig. 3 the second thread trades R_B of its initial R_S resources to task A when stealing C. This leaves the worker with $R_S - R_B$ resources for the remaining invocation tree, which are sufficient for executing it, as R_S equals the longest branch. Following this trade-in rule and the busy-leaves property, each task t with n active child tasks has $n - 1$ traded resources associated with it. The first $n - 1$ children finishing can not execute t, but can combine their free resources with one of the traded in resources to enter the stealing state with the initial R_S resources. The last child finishing does not require spare resources, as it can continue working on the parent task, freeing its resources when finishing it.

Following this simple trade-in rule, resources can be balanced between the workers only on steal and synchronization events, leading to strict space bounds. The trading affects the time bounds by adding the work to trade resources to the steal procedure. When the work required for stealing is proportional to c_∞, the expected time bound in Eq. (3) follows.

$$T_P = O(T_1/P_i + c_\infty T_\infty) \tag{3}$$

4.2 Prototype Implementation

In order to keep the busy-leaves property and strict time bounds, the prototype must implement a cactus-stack and not block in threads. The system model suggests that parallel sections are clearly defined and should be predictable in resource usage, having the execution switch from a serial to a parallel section explicitly. Figure 3 shows this with a switch from the serial stack (shaded) to the parallel invocation tree at the top, dotted line. We decide to build a cactus stack by executing each spawned task on a small stackful coroutine[3], as spawns tend to be dense in a parallel section, requiring only a small stack per task. Calls into purely serial code that potentially uses more stack space are run on a separate, bigger stack, as indicated with the bottom, dotted line in Fig. 3. By doing this, P^3LS^3 holds strict theoretical bounds and implements the API in Listing 1.

The previous section on the resource-trading algorithm introduced abstract resources R_S that can be split and united at any point. However, memory can not be split and united at will, as computers rely on continuous blocks of memory in the virtual address space. Our first prototype therefore trades fixed size memory blocks managed in a linked list, i.e. it trades the stackfull coroutines to execute tasks. Each thread starts with D blocks equal to the deepest spawn depth and trades are performed by slicing and concatenating parts of the lists. The time c_∞ to perform a steal is therefore $c_\infty = O(D)$.

The stealing procedure integrating resource-trading is implemented in a non-blocking manner, thus holding bounds on a multiprogrammed environment [1]. During stealing, a flag is atomically updated from a thief to acquire a task, similar to Wool [4]. The new value indicates both the stolen state of the task and contains the traded in resources, making the action of stealing and trading in resources atomic. Each task additionally holds a stack of currently traded in resources, which is also used to implicitly synchronize as the last finishing child encounters an empty resource stack.

During development the program must be executed once to measure the maximum size of the coroutines and the computation depth D by triggering the biggest possible invocation tree. These measured values are then used to configure the scheduler, which during startup acquires the $S_P = O(P_i(S_1 + D))$ memory required for the execution. This way of finding the static memory

[3] The resource-trading algorithm can work with any other choice of cactus-stack and non-blocking scheduler. We choose coroutines as we are interested in exploring a pure library solution with a clean API.

footprint is very accessible for the developer, as it only depends on few metrics and the model of an invocation tree is simple to reason about.

5 Performance Analysis

We evaluate the performance of P^3LS^3 by comparing it to TBB (2019, interface version 11000) and EMB^2 (v1.0.0). All benchmarks are executed on a Banana Pi M3 as an example of a high performance embedded system. The board is equipped with an A83T ARM SoC housing a Cortex-A7 octa-core processor clocked at 1.6 GHz and runs the vendor supplied linux operating system, which is based on the 3.4.39 smp preemp kernel. The benchmark applications are compiled with GCC v5.4 using optimization level `-O3`.

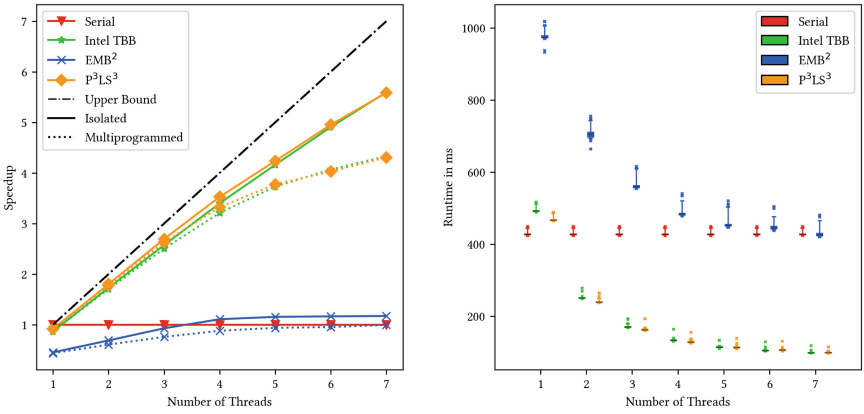
We first analyze the scheduling overhead of the frameworks using the synthetic load of unbalanced tree search [11], followed by an embarrassingly parallel row wise matrix multiplication. Further benchmarks where conducted, but are not shown for brevity. However, they all show the same trend as seen in the following evaluation. Each benchmark is discussed in one of the following subsections, with diagrams showing the resulting speedups and full execution time distributions. The box plots indicate the 95th and 5th percentile execution times with whiskers and all fliers are included.

All measurements are performed in both isolated and multiprogrammed system conditions. Isolated tests are executed with minimal influences from other processes running on the system, by isolating the benchmark processor cores and using the round robin real-time scheduler. These isolated measurements are most common in other benchmarks, and thus can be used for comparison. To simulate a multiprogrammed system, we intentionally run one higher priority process per CPU core, potentially preempting the currently running benchmark. The processes are periodically performing work and memory access, resulting in a measured per core utilization of an average 25%.

5.1 Unbalanced Tree Search

Unbalanced tree search [11] constructs and traverses a highly unbalanced tree by calculating a hash value at each node. The benchmark spawns a task for each node, resulting in very unpredictable load with many synchronization points, revealing the frameworks scheduling overhead. For our test, we choose to spawn an initial 140 nodes at the tree root and eight children with a probability of $q = 0.124875$. This leads to a tree with about 71,000 nodes and therefore the same amount of spawned tasks. We repeat each benchmark 50 times.

Figure 4a shows that TBB and P^3LS^3 both achieve a near identical speedup in this test, while EMB^2 can not accelerate the computation, even introducing a significant slowdown at low core counts. This confirms the assumption that decentralized work-stealing and its depth-first tree traversal is superior to the work-sharing scheduler used in EMB^2 for this kind of fine-grained, recursive tasks. Turning to the time distributions in a multiprogrammed environment,



(a) average case speedups (b) multiprogrammed time distribution

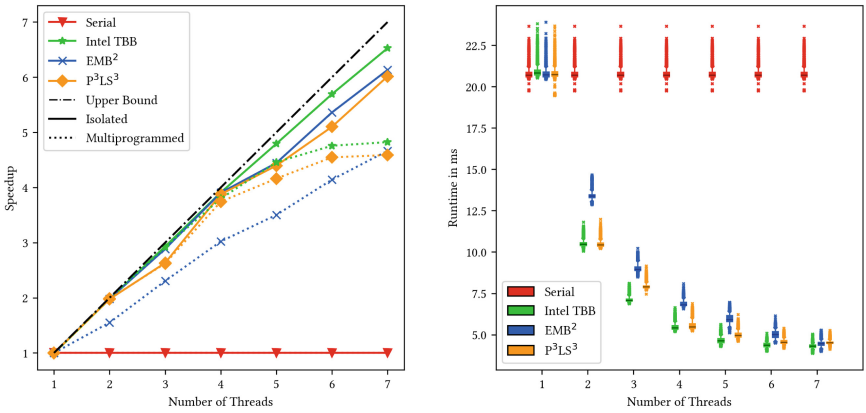
Fig. 4. Results of the unbalanced tree search benchmarks.

shown in Fig. 4b, we can see generally low dispersion for TBB and P^3LS^3, close to the serial measurements. EMB^2 in contrast shows by far the biggest irregularity in execution times. This trend is also visible in non synthetic divide and conquer algorithms like a fast Fourier transform, although it is less pronounced.

5.2 Matrix Multiplication

We implement a row wise matrix multiplication as an example of an embarrassingly parallel algorithm. Using the frameworks `parallel_for` constructs, rows are executed concurrently, making the libraries handle splitting up data and load balancing internally. We choose a matrix size of 128×128 and repeat each benchmark 5000 times.

The measured average speedups compared to the serial implementation are shown in Fig. 5a. All three frameworks nearly reach the theoretical upper bound of a perfect, linear speedup when being executed on an isolated system, with TBB being the fastest by a slight margin. When looking at the multiprogrammed environment, all frameworks show worse speedups. Interestingly, EMB^2 slows down more drastically than the other two frameworks on lower core counts, suggesting problems with either load balancing or synchronization. Looking closer at the time distributions in Fig. 5b, we notice a trend of more consistent execution times with increasing thread count and observe that the libraries do not introduce more variance than a serial execution, suggesting that the overall system jitter by the higher priority workers dominates.



(a) average case speedups (b) multiprogrammed time distribution

Fig. 5. Results of the matrix multiplication benchmarks.

6 Conclusion

We analyzed how task-parallel programming and dispatching frameworks fit into the embedded and real-time domain. We argue that classic work-stealing offers good theoretical bounds and our tests verify that it leads to consistently fast execution times even on small problem sizes. Specifically, the benchmarks show that EMB^2 suffers from load balancing issues, while TBB and P^{3}LS^3 perform almost equally well on all tests. This behavior results from P^{3}LS^3 and TBB using a lock-free work-stealing scheduler, while EMB^2 uses a mostly work-sharing implementation and therefore, find the former to be superior for parallelizing individual applications. Furthermore, all benchmarks show that the use of work-stealing frameworks does not increase the dispersion of the execution times compared to a sequential execution. The uncertainty from randomized stealing is dominated by other system effects.

Our remaining concern with existing dispatching frameworks is their liberate use of dynamic memory management and occasional deviation from classic work-stealing bounds in favor of mainstream usability. As embedded systems can require static resource allocation, we implemented a prototype work-stealing library in `C++`, offering both static memory allocation and strict theoretical bounds. Our time measurements show that our early prototype P^{3}LS^3, implementing the proposed resource-trading approach, can keep up with the industry leading TBB. Unfortunately, P^{3}LS^3 can not improve execution time variances compared to TBB, even though TBB implements blocking style work-stealing and uses dynamic memory management. Under our current measurement conditions, we can therefore not detect any sporadic, negative effect on execution times resulting from dynamic memory allocations.

Currently, we focus on the performance of individual applications, in future work we would like to investigate the behavior of multiple task-parallel applications running concurrently on a real-time OS. We would also like to refine our measurements, by including memory usage and by looking at smaller problem sizes. Lastly, we want to explore if resource-trading can be integrated into parallel patterns that require structured memory allocations, like e.g. divide and conquer algorithms with temporary buffers.

References

1. Arora, N.S., Blumofe, R.D., Plaxton, C.G.: Thread scheduling for multiprogrammed multiprocessors. In: Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA 1998, Puerto Vallarta, Mexico, pp. 119–129. ACM (1998). https://doi.org/10.1145/277651.277678
2. Atkinson, P., McIntosh-Smith, S.: On the performance of parallel tasking runtimes for an irregular fast multipole method application. In: 13th International Workshop on OpenMP, IWOMP 2017. LNCS, vol. 10468, pp. 92–106. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-65578-9_7
3. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. In: Proceedings 35th Annual Symposium on Foundations of Computer Science, pp. 356–368, November 1994. https://doi.org/10.1109/SFCS.1994.365680
4. Faxen, K.: Efficient work stealing for fine grained parallelism. In: 39th International Conference on Parallel Processing, pp. 313–322, September 2010. https://doi.org/10.1109/ICPP.2010.39
5. Frigo, M., Leiserson, C.E., Randall, K.H.: The Implementation of the Cilk-5 multithreaded language. In: Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation, PLDI 1998, Montreal, Quebec, Canada, pp. 212–223. ACM (1998). https://doi.org/10.1145/277650.277725
6. Gleim, U., Levy, M.: MTAPI: parallel programming for embedded multicore systems. Technical report (2013)
7. Halpern, P.: Strict fork-join parallelism. Technical report N3409, September 2012
8. Kukanov, A., Voss, M.J.: The foundations for scalable multi-core software in Intel® threading building blocks. Intel Tech. J. **11**, 309–322 (2007). https://doi.org/10.1535/itj.1104.05
9. Lee, I.T.A., Boyd-Wickizer, S., Huang, Z., Leiserson, C.E.: 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 2010. Vienna, Austria, p. 411. ACM Press (2010). https://doi.org/10.1145/1854273.1854324
10. Li, J., Dinh, S., Kieselbach, K., Agrawal, K., Gill, C., Lu, C.: Randomized work stealing for large scale soft real-time systems. In: IEEE Real-Time Systems Symposium, RTSS, pp. 203–214, November 2016. https://doi.org/10.1109/RTSS.2016.028
11. Olivier, S., et al.: UTS: an unbalanced tree search benchmark. In: International Workshop on Languages and Compilers for Parallel Computing, LCPC 2006. LNCS, vol. 4382, pp. 235–250. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-72521-3_18
12. Schuele, T.: Embedded multicore building blocks - parallel programming made easy. In: Embedded World 2015 (2015)

13. Sukha, J.: Brief announcement: a lower bound for depth-restricted work stealing. In: Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures, SPAA 2009, Calgary, AB, Canada, pp. 124–126. ACM (2009). https://doi.org/10.1145/1583991.1584025
14. Voss, M., Asenjo, R., Reinders, J.: Pro TBB. Apress, Berkeley (2019)
15. Wagner, D.B., Calder, B.G.: Leapfrogging: a portable technique for implementing efficient futures. In: Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 1993, San Diego, California, USA, pp. 208–217. ACM (1993). https://doi.org/10.1145/155332.155354