Colocation of Potential Parallelism in a Distributed Adaptive Run-time System for Parallel Haskell

Evgenij Belikov, Hans-Wolfgang Loidl, and Greg Michaelson

School of Mathematical and Computer Sciences,
Heriot-Watt University, Edinburgh EH14 4AS, Scotland, UK
{eb120,H.W.Loidl,G.Michaelson}@hw.ac.uk
http://www.macs.hw.ac.uk

Abstract. This paper presents a novel variant of work stealing for load balancing in a distributed graph reducer, executing a semi-explicit parallel dialect of Haskell. The key concept of this load-balancer is *colocating* related *sparks* (potential parallelism) using maximum prefix matching on the encoding of the spark's ancestry within the computation tree, reconstructed at run time, in spark selection decisions. We evaluate spark colocation in terms of performance and scalability on a set of five benchmarks on a Beowulf-class cluster of multi-core machines using up to 256 cores. In comparison to the baseline mechanism, we achieve speedup increase of up to 46% for three out of five applications, due to improved locality and load balance throughout the execution as demonstrated by profiling data. For one less scalable program and one program with excessive amounts of very fine-grained parallelism we observe drops in speedup by 17% and 42%, respectively. Overall, spark colocation results in reduced mean time to fetch the required data and in higher degree of parallelism of finer granularity, which is most beneficial on higher PE numbers.

Keywords: Parallel Functional Programming · Graph Reduction · Load Balancing · Distributed-Memory Work Stealing · Adaptive Parallelism

1 Introduction

Exploiting modern distributed parallel architectures is key for improving application performance and scalability beyond a single machine, for instance for Large-Scale Data Analytics and High-Performance Computing. Additionally, using a high-level programming language is crucial for countering growing software complexity and for increasing programmer productivity by delegating most of the coordination and parallelism management to the run-time system (RTS). Functional Programming offers a high level of abstraction and advanced language features [1, 16, 14], e.g. higher-order functions, polymorphism, and type classes. In particular, functional languages appear suitable for exploitation of fine-grained parallelism as independent sub-expressions can be evaluated in any order without changing the result (known as the Church-Rosser property [9]), facilitating incremental parallelisation and allowing for sequential debugging of parallel programs, whilst avoiding race conditions and deadlocks [13].

Work stealing [5] is a popular passive (i.e. receiver-initiated) decentralised load balancing mechanism, where idle processing elements (PEs) attempt to steal work from busy PEs. Important parameters in this mechanism are the target of the steal attempt and the choice of the (potential) parallel work units, or *sparks*. In our current parallel RTS the target is randomly selected, to avoid hotspots in the communication, and older sparks are preferred, because they typically represent work of larger granularity. Large granularity aims at offsetting the communication costs, especially in computations that use the Divide-and-Conquer (D&C) pattern or are nested and are run on distributed architectures with very high communication costs.

Note that in our system all parallelism is *advisory* rather than *mandatory*. This means that RTS policies can adaptively tune the amount of parallelism, deciding not to generate actual parallelism. This can effectively in-line work into other threads and thereby improve the granularity of the computation.

In this paper we investigate the effect of a modification to the spark (work) selection policy, namely *spark colocation* (SC), on performance and scalability. SC exports the spark that is, according to a specific metric, most closely related to the computation performed by the thief and is aimed at resolving the trade-off between data locality and load balance, instead of exporting the oldest spark. The chosen metric for proximity is the distance in the compute tree, and the RTS is extended to capture a trace of spark sites, representing the path in the tree leading to this spark. On selecting a spark to export to another PE, the one with the longest common prefix is used, as the one that is most closely related to recent work performed on the thief's PE. Compared to the baseline mechanism, SC achieves speedup increase of up to 46%, due to improved locality and load balance throughout the execution as demonstrated by profiling data, whilst for one less scalable application and one with excessive amount of overly fine-grained parallelism we observe drops in speedup of 17% and 42%, respectively.

Next we introduce the GUM RTS for Glasgow parallel Haskell in Section 2 and discuss the design and implementation of spark colocation in Section 3, followed by evaluation of empirical results for five applications based on means-based metrics from per-PE profiles gathered from runs on a 256-PE-cluster in Section 4. A brief discussion of related work follows in Section 5, before our conclusion and future work directions are presented in Section 6.

2 Distributed Graph Reduction in the GUM RTS

Here we briefly introduce the Glasgow parallel Haskell (GpH) language and the underlying GUM (Graph Reduction on a Unified Machine Model) RTS that implements distributed graph reduction [31], including most notably using global addresses to implement virtual shared memory, thread management using *sparks* that efficiently represent potential parallelism, and work stealing, or *fishing*, for passive load distribution.

2.1 Haskell Extension for Semi-Explicit Parallelism

Glasgow parallel Haskell (GpH) [12] extends Haskell [15, 24], a popular non-strict purely functional language, by adding a sequential and a parallel combinator as language primitives (`pseq` and `par`), which allow the specification of evaluation order and identification of potential parallelism, respectively. This high-level programming model is *semi-explicit*. The advisory parallelism identification and optional application-level granularity control are explicit. All other coordination aspects, such as communication and synchronisation, are implicitly controlled by the RTS. Listing 1.1 provides an example.

```haskell
fib 0 = 0                          -- sequential version
fib 1 = 1                          -- NB args of type Integer
fib n = fib (n-1) + fib (n-2)

pfib 0 _ = 0                       -- parallel version
pfib 1 _ = 1
pfib n t | n <= t = fib n     -- threshold for granularity control
         | otherwise = x `par` y `pseq` x + y
           where x = pfib (n-1) t
                 y = pfib (n-2) t
```

Listing 1.1. GpH Example: Sequential and Parallel Fibonacci Functions

Using `par`, the programmer provides a hint to the RTS that the first argument expression can be beneficially evaluated in parallel, thus creating a *spark*, and the RTS decides at run time whether the spark will be turned into a light-weight thread increasing the actual degree of parallelism or ignored. Note that in order to be useful the first expression should be unevaluated, represent a large-enough amount of computation, and be shared with the rest of the program [21]. This mechanism can be viewed as implementing *lazy futures* similar to *lazy task creation* [25]. To cleanly separate the computation and coordination concerns *Evaluation Strategies* [30, 22] were introduced on top of the basic primitives.

2.2 Memory Management

GUM implements GpH by supporting distributed graph reduction, where each graph node represents a potentially shared computation, using a combination of a *virtual shared memory* that holds the shared graph nodes and independent local heaps associated with separate GUM instances that run on each PE in parallel. Once a node has been evaluated it is replaced by the result, which is in turn sent to all the PEs that require it.

This design, based on private heaps with some sharing across them, is scalable as most of garbage collection (GC) can be performed *locally* without the need for communication and synchronisation. GUM uses a *generational* garbage collector that is either *copying* or *compacting* depending on the RTS flags set, thus avoiding a stop-the-world design (e.g. as used in GHC-SMP [23]). Heap objects that survive for a long time are promoted from the initial and frequently GC'd heap area (called *nursery*) to a different space that is GC'd less often. This GC scheme assumes that most heap objects will expire after a short period of time allowing the associated memory to be reclaimed. Additionally, GUM uses *distributed weighted reference counting* [4] to manage the virtual shared heap.

2.3 Thread and Parallelism Management

GUM represents sub-computations using light-weight threads that are mapped to relatively few heavy-weight OS threads (often one per core) in an M-to-N fashion for scalability (similar to Green Threads). Each RTS instance maintains a local thread pool for runnable threads and blocked queues for threads waiting on a result of evaluation performed by another potentially remote thread[1].

GUM's scheduler is unfair and non-preemptive. It prioritises handling messages and implements the *evaluate-and-die* evaluation model [28]. In this model a thread picks up a thunk (an unevaluated expression) to evaluate and returns control to the scheduler once either the evaluation to weak-head normal form has completed or thread blocks waiting on another value under evaluation.

Sparks that represent potentially parallel work are created using the `par` primitive and kept in a separate local pool on each PE. Sparking is inexpensive, as it merely adds a pointer to a graph node representing the expression to be evaluated to the pool, which is implemented using an efficient lock-free dequeue [8], which allows the owner to use one end locally for pushing, whilst older sparks are stolen off the other end using a single atomic compare-and-swap operation (FIFO). The overhead is absent unless two threads happen to simultaneously operate on the same item of the dequeue. Sparks are discarded if they have been already evaluated or if the spark pool is full.

2.4 Workload Distribution

Load balancing across PEs is achieved through work stealing (also called *fishing*) and aims at reducing the overall idle time across PEs. The two main decisions include: 1) where to steal from (victim selection by the thief or selection of forwarding destination by victim with no sparks available for export) and 2) which spark to export (decision made by a victim that has exportable sparks). This work is focused on the latter decision.

Figure 1 illustrates the message types and the protocol. A `FISH` message is a request for work and is forwarded to randomly selected PEs until either some work was found or the `FISH` expires by reaching a maximum age (it is incremented with every hop). If the thief was successful, it receives a `SCHEDULE` message containing some work and potentially some related data. The thief responds by sending an `ACK` message with an updated list of pairs of old and new global addresses to the victim to update the virtual shared memory to reflect the change. If the `FISH` expires, it is sent back to the original PE, which then can then send out a new `FISH`.

The default mechanism selects a victim at random. A victim that receives a `FISH`, selects the oldest spark for donation and sends it back to the origin PE. This is where SC differs: it selects a spark from the same source of parallelism using *maximum prefix matching* on the encoding of the path of the spark within

[1] parallelism is exploited over pure functions and I/O is handled orthogonally by a separate thread

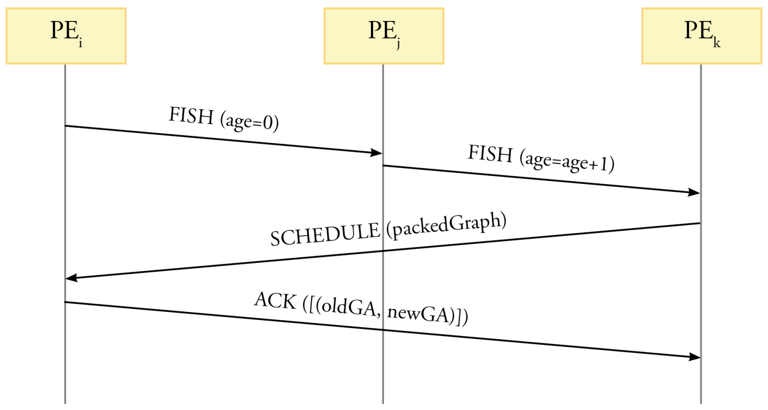

Fig. 1. Multi-Hop Successful FISHing Attempt

the computational graph, rather than using the age of the spark (as described in detail in Section 3).

Fish delay and delay factor as well as a limitation on the number of outstanding fishes (currently one) are used to avoid swamping the network with `FISH`es. Thread migration is not supported in the current implementation.

3 Spark Colocation

Spark Colocation is aimed at improving load balance and locality by exporting the spark that is most closely related to the computation performed by the thief.

3.1 Motivating Example

Consider the example from Figure 2 that illustrates a situation where two PEs work on several tasks and one PE needs to decide which spark to donate.

The tree structure represents computational dependencies, whilst the dashed regions depict which tasks are located on which PE. In particular, both sparks ended up on PE1. As PE2 continues the evaluation it runs out of tasks and sends a `FISH` to PE1. In turn, PE1 can now decide which spark to donate. It would donate B, which we assume is older[2], in the baseline case. Then it would continue to execute the remaining spark A locally. However, the result of A is needed by PE2, which would require additional communication. Similarly, if spark B is exported and turned into a thread on PE2, communication is required to send the result to PE1. If Spark Colocation is used A would be donated as it is more related to the computation on PE2.

[2] this is reasonable as PE1 is the main PE and PE2 starts with no work

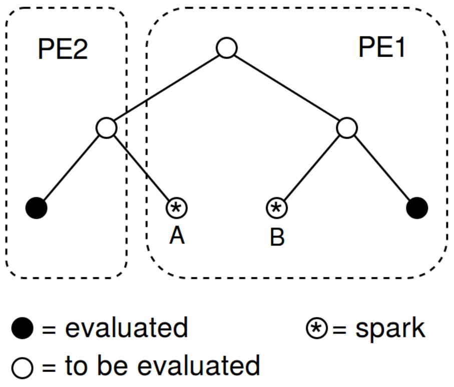

Fig. 2. Example of Potential for Colocation

The main idea is to allocate computations to PEs that have worked on related computations. A related computation is located closely in the same computational sub-tree, because its result or produced data are likely to be required by the other computation. The concept of SC builds on the notion of proximity between computations. Two sparks are defined to be in close proximity if the path in the tree between their nodes is short. In particular, if the root node is on the path, the sparks can be considered unrelated.

3.2 Design

SC is an extension of the baseline work stealing mechanism, investigating the effect of favouring colocation of related sparks, rather than selecting a spark to export based on its age alone. The idea is to allocate computations to PEs that have worked on related computations, i.e. computation located closely in the same computational sub-tree likely to require the result of, or share some data with the other. Using SC, the information on the proximity between sparks that would normally be lost during compilation is forwarded to the RTS, where it can dynamically influence scheduling and load balancing decisions at run time.

Informally, the colocation algorithm behaves as follows: if a PE is idle, it will attempt to steal work from others that will respond with the spark on the path through the compute tree that is most related to the computation performed by the thief, rather than with the oldest. We use the *ancestry* relation with the *maximum prefix* function as the matching function for finding the best match between the encoding of the thief and of the sparks available to the victim. The baseline mechanism is used as a fallback.

Figure 3 illustrates the encoding for two sources of parallelism, thus base 2 is used for the encoding. For example, if spark A with the encoding 01 was turned into a thread and then had the choice between sparks B and C, the latter would be chosen as given its encoding 010 it has a longer common prefix of length two with A as opposed to B with encoding 00, which shares only one symbol with A. We can also see that A requires the result of computation C, whilst it does not require the result of B to proceed. An ancestor of a spark is recursively defined as either the direct creator of the spark (its parent), or as the ancestor of its parent. The ancestry relation is encoded as a path in the computation represented by a string of symbols that encode the branch at each tree level.

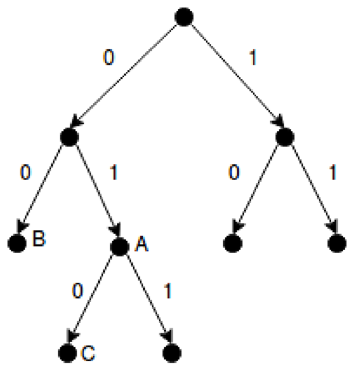

Fig. 3. Spark Ancestry Encoding Example

We select *maximum prefix* as a matching function, because the resulting encoding mirrors closely the actual tree-like computational structure of the application. The ancestry relation defines the distance between a thread's encoding and the encoding of a given spark as the sum of edges traversed on the path from one encoding to the other in the tree. The smaller the distance the more closely related two sub-computations are deemed to be. Investigation of alternative encodings and matching functions remains for future work.

3.3 Implementation

SC is implemented as an explicit language primitive — a version of the `par` combinator we call `parEnc` — that takes additional encoding arguments that are passed to the RTS and used to tag the sparks. The path to the spark constitutes an encoding, where we start from the root and add a symbol for each sub-branch chosen at each level. The symbol corresponds to the `parEnc` site that leads to the creation of the spark and is appended to its inherited parent's encoding.

Spark Selection: In the baseline mechanism, the spark pool is implemented as a lock-free double-ended queue, so that the owning PE can add new sparks at the tail of the deque whilst sparks are exported off the head. This mechanism avoids most of the synchronisation cost as it is only incurred when threads attempt to dequeue the same spark, as the owner turns local sparks into threads by taking them from the tail, which is similar to the Breadth-first-Until-Saturation-then-Depth-first mechanism [6].

By contrast, SC uses spark encodings to select related sparks, if possible. Internally, we use hash tables to store and efficiently access the information on threads and sparks using their respective identifiers as lookup keys. This mechanism enables the RTS to distinguish sparks based on their source of parallelism and location within the compute tree of the application for a given input. Each time a spark is created it stores its encoding in the hash table. This encoding is compared to the encoding carried by an incoming `FISH` message, extended with information about the encoding of the thief. The spark pool is traversed and a spark with a maximum prefix match is donated. To trade precision for overhead, the maximum traversal length can be specified as an RTS option.

Matching Function: We have chosen to encode ancestry as a string of symbols to the base needed to encode the maximum number of branches at a level of the tree, reflecting the dynamic relationship that arises at run time.

As a natural choice, maximum prefix string matching is used to determine the spark for export, since it represents the closest relation between the computations in the graph. Nevertheless, the matching may potentially lead to more communication than in the baseline case and increased amount of inter-PE sharing as implicated by the number of global addresses. Therefore an empirical evaluation is needed.

Packet Format: To propagate ancestry information, the packet format is extended for the `FISH` and the `SCHEDULE` protocol messages. `FISH` is extended to carry the requesting PE's encoding, whilst `SCHEDULE` includes the exported spark and its encoding. When turned into a thread, the spark's encoding is used as the thread's encoding, which is in turn passed on to the sparks it may generate.

Profiling: To facilitate comparison between SC and the baseline mechanism, the event-based profiling sub-system is extended to record thread granularities, i.e. the run time elapsed from start to termination of a thread, and fetch times, i.e. run time spent in the state waiting for data to arrive, in addition to the already available profiling information such as per-PE load over time, message counts, and number of global addresses.

The extension is small as it requires mainly adding calls to a timer function in places where a thread enters a particular state (e.g. fetching) and recording the difference when a transition to another state occurs. The extension does not impede scalability as it only involves keeping an additional per-thread counter adding little to the existing profiling overhead, whilst the events are written out to file as they occur using a separate asynchronous thread responsible for buffered I/O.

4 Evaluation

We compare SC and the baseline mechanism using empirical measurements.

4.1 Methodology

We run each of the five applications five times for each PE-count both with and without event-based profiling and compare the median runs with and without SC[3]. The elapsed (wall-clock) run time is measured in milliseconds and includes both the mutation time and the garbage collection time. We don't have exclusive access to the cluster, so that although it is usually lightly loaded, we can't fully rule out some variation due to interference with other processes running on the machines. As PVM is used as a communication library [11], processes are placed onto nodes in a round robin fashion as specified in a hostfile that is read in top-to-bottom order.

Using ends-based metrics such as run time and speedup alone doesn't provide sufficient insight into why the observed effects of SC take place, for instance with respect to load balance over time. Therefore, we also collect profiling data for several means-based metrics: per-PE numbers of threads over time as a measure of load balance and degree of parallelism, thread sizes reflecting granularity, numbers of transmitted messages of different types, as well as the numbers inter-PE pointers to assess data locality, and fetch times and counts for data-carrying messages.

4.2 Target Platform

The applications are run on a 32-node Beowulf cluster of multi-cores using up to 256 PEs. The cluster comprises a mix of 8-core Xeon 5504 nodes with two sockets with four 2GHz cores, 256 KB L2 cache, 4MB shared L3 cache and 12GB RAM, and 8-core Xeon 5450 nodes with two sockets with four 3GHz cores, 6MB shared L2 cache and 16GB RAM. The machines are connected via Gigabit Ethernet with an average latency of 0.23 μs, measured using the Linux `ping` utility (average round-trip time of 100 packets). We use the CentOS 6.7 operating system, the GHC 6.12.3 Haskell compiler, the GCC 4.4.8 C compiler, and the PVM 3.4.6 communication library. The optimisations are turned on (`-O2`).

4.3 Applications

We use five D&C benchmark applications adopted from the parallel part of the established *nofib* benchmarking suite [26] and from a recent study of Evaluation Strategies [22]. In particular, we use `parfib` which is the standard parallelism microbenchmark, `parpair` with calls to `sumeuler` and `parfib` nested within the pair and evaluated in parallel, interval-based `sumeuler` version reformulated

[3] median is used as it is more robust to outliers

using the D&C pattern that calculates the sum of Euler Totient[4] functions in a given range, `worpitzky` that calculates the Worpitzky identity[5] and `minimax` that implements a game using alpha-beta pruning.

Table 1. Applications Overview

application	parallelism pattern	regularity	input parameters
`parfib`	D&C	regular	50 35
`parpair`	nested D&C	irregular/regular	100000 10 50 35
`sumeuler`	D&C	irregular	100000 10
`worpitzky`	D&C	irregular	27 30 18
`minimax`	D&C	irregular	4 8 2

4.4 Results

The results summarised in Table 2 demonstrate that substantial speedups can be reached for both the baseline and for the colocation case over sequential run time, achieving speedup improvement of up to 46% with SC over the baseline for three of the programs. However, we also observe a drop in speedup for SC, for the less scalable `minimax`, and for `worpitzky` with excessively fine-grained parallelism and parallelism degree of 17% and 42%, respectively. We focus on load balance and granularity profiles for `sumeuler` as they most clearly depict the differences between the mechanisms.

Table 2. Applications' Speedups on 256 PEs

application	sequential run time (sec)	baseline speedup	colocation speedup	change in %
`parfib`	1609	204	219	+7
`parpair`	2870	200	231	+16
`sumeuler`	1450	142	207	+46
`worpitzky`	3269	175	101	−42
`minimax`	160	95	79	−17

Load Balance: Figures 4 and 5 show the detailed per-PE profiling data for `sumeuler` indicating load balancing behaviour change resulting from SC use.

[4] `http://mathworld.wolfram.com/TotientFunction.html`
[5] `http://mathworld.wolfram.com/WorpitzkysIdentity.html`

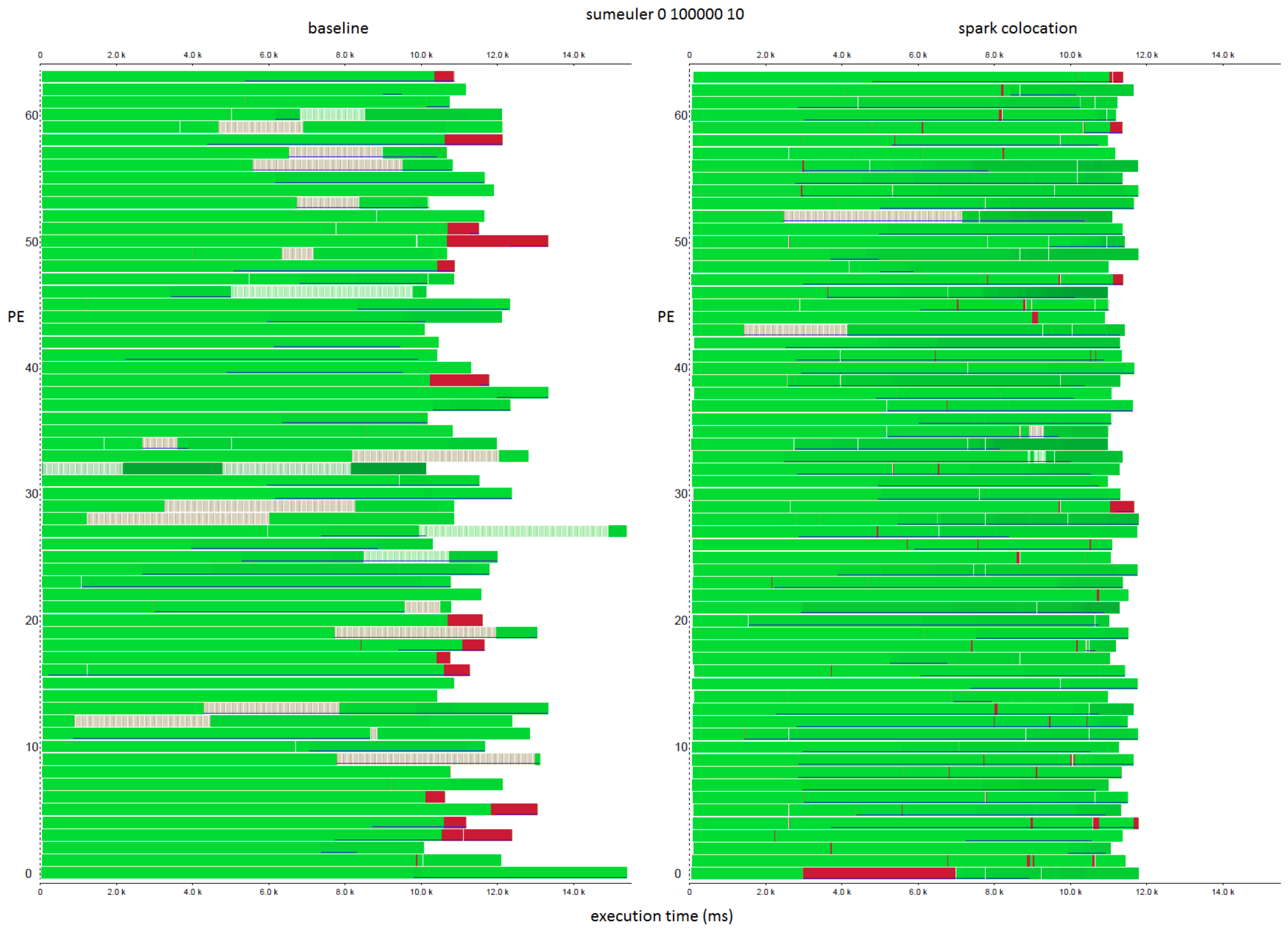

Fig. 4. Event-Based Load Balancing Per-PE Profile Comparison for `sumeuler` **PEs 1-64** out of 128

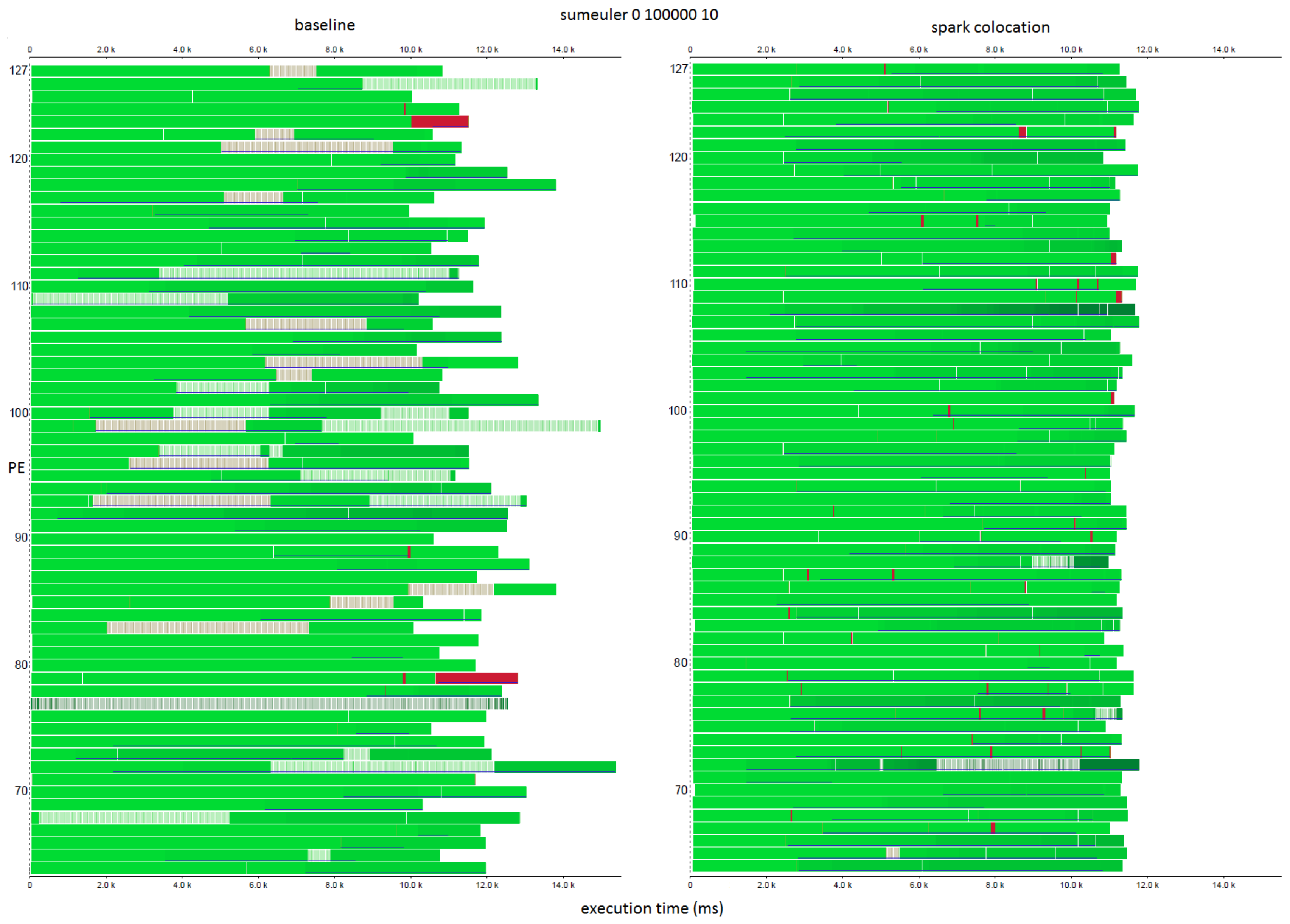

Fig. 5. Event-Based Load Balancing Per-PE Profile Comparison for `sumeuler` **PEs 65-128** out of 128

We visualise data using 128 PEs for readability, but the difference is stronger for higher numbers of PEs. Figure 4 visualises PEs 1–64 as horizontal bars, Figure 5 PEs 65–128, baseline being on the left and SC on the right. A per-PE profile shows PEs on the y-axis and execution time in milliseconds on the x-axis, thus depicting load-balance across PEs over time. The darkness of the green value at each point in time shows the utilisation (i.e. the number of runnable threads) as an average over a fixed time window, whilst idle time is shown in red. Additionally, the small blue stripes embedded in the lines for each individual PE reflect the number of communicating (blocked-on-fetch) threads.

Overall, we observe better load balance for SC, as almost all of the bars are green, as opposed to the baseline case, where there are substantially more gaps and areas with a reduced number of threads visible. In particular, most of the blocking time is at the end of execution for the baseline (we can distinguish the execution and the waiting for termination as two distinct phases), but it is more spread out and more evenly distributed across more PEs for SC, which exhibits fewer blocking hotspots. We can see noticeably more short green stripes for baseline reflecting the need to fetch data, which appears less often for SC as either the data is readily available or the waiting can be overlapped with computation performed by another thread.

Additionally, the data show good load balance for SC, with very similar total run times on each PE, whilst for the baseline the run times are more variabile, with differences of over 30% of the total run time in some cases.

Granularity: We use event-based profiling to record execution time for each thread. Figure 6 depicts the granularity of `sumeuler` on 256 PEs, with number of threads on the y-axis and thread granularity in milliseconds on the x-axis. Light-red represents the baseline case, light-blue SC, and a darker shade shows the overlap between both. The granularity profiles are overlapping but distinct.

We observe fewer threads and coarser granularity for the baseline case[6], which results from exporting older and likely larger sparks, which are then turned into threads on arrival at the thief PE. Note that the RTS cannot re-balance threads, as opposed to sparks, between PEs, and therefore this behaviour can lead to load imbalance. By contrast, SC exports sparks that are closer to a thief's encoding, but of smaller granularity, which allows more flexibility in saturating larger number of PEs. Although finer granularity is associated with additional overhead, in this case the advantage of improved load balance out-weighs this overhead. Note that due to thread subsumption, which allows a thread to evaluate a potentially parallel child computation sequentially, not all of the fine-grained sparks will be turned into threads, thus reducing the overhead.

Degree of Parallelism: Complementing the granularity profiles, Tables 3 and 4 present the measured total (across PEs) and calculated median (per PE) spark and thread counts, representing the *potential* and *actual* degree of parallelism, respectively. We report data from the median run profiled on 256 PEs for each benchmark, comparing the baseline against SC.

[6] for other benchmarks SC consistently leads to more and smaller threads

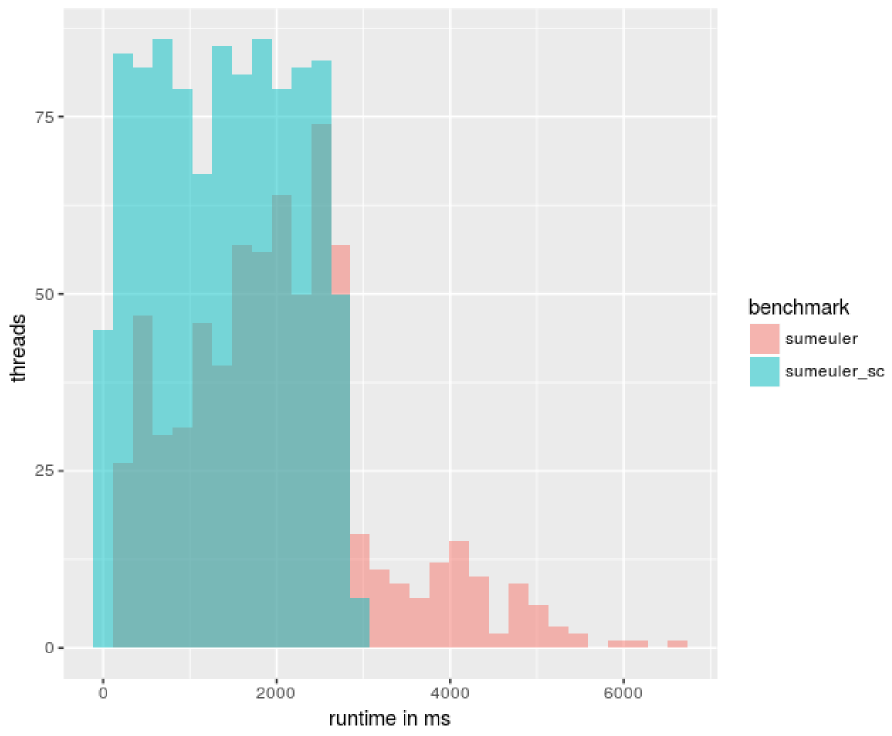

Fig. 6. Granularity of `sumeuler` on 256 PEs

Table 3. Spark Counts for Benchmarks on 256 PEs

application		median		total	**change**
	baseline	SC	baseline	SC	**in %**
`parfib`	11	12	2755	3172	+15
`parpair`	14	19	3840	5045	+31
`sumeuler`	6	7	1854	1983	+7
`worpitzky`	1322	1927	337116	488550	+45
`minimax`	7	5	2466	2525	+2

Overall, we observe consistently higher potential parallelism in the range between 2% and 45% for SC, which translates into proportionally higher increase in the number of threads of up to 197%. This can be attributed to the export of related sparks rather than the oldest, which may reduce potential for subsumption once the computation is shared across the PEs. Sparks are inexpensive as they are pointers to sub-graphs and can be maintained with low overhead and allow more flexibility for load balancing, potentially increasing utilisation. Threads

Table 4. Thread Counts for Benchmarks on 256 PEs

application	median		total		**change**
	baseline	SC	baseline	SC	**in %**
`parfib`	4	6	1127	1584	+41
`parpair`	5	10	1195	2508	+110
`sumeuler`	3	4	802	955	+19
`worpitzky`	322	979	82065	243709	+197
`minimax`	4	4	1092	1055	−3

are more expensive as they require the creation of data structures in the heap to hold thread state and related information, which may increase the memory management overhead.

Using SC turns out to be particularly beneficial for larger numbers of PEs as the number of threads per PE is increased in all but one case, whilst the amount of total heap available grows with the number of PEs reducing the pressure on the garbage collector. The `worpitzky` benchmark is an example of worst-case behaviour, demonstrating that having a higher number of threads may become counterproductive when there are already more than enough threads in the baseline case, due to additional overhead, reducing scalability.

Fetching Behaviour: Another distinguishing characteristic and the *most direct indicator of SC's efficacy* is the *fetch time* threads spend waiting for data required by the computation to arrive. Table 5 compares the baseline and SC across applications for the median run on 256 PEs (no data available for `minimax`).

Table 5. Overview of Fetching on 256 PEs (in ms)

application	baseline mean fetch time across PEs	colocation mean fetch time across PEs	**mean fetch time change in % across PEs**	total fetch time change in %	total fetch count change in %
`parfib`	829	637	−23	+8	+35
`parpair`	1109	566	−49	−5	+78
`sumeuler`	594	290	−51	−29	+49
`worpitzky`	19	12	−40	+81	+163

In some cases it is possible that the data is already available or fits into the same packet, resulting in fetch time of zero, as for many `sumeuler` threads, and in other cases the fetch time may exceed the time the thread spends performing the computation.

We observe that *SC has consistently a smaller mean fetch time across PEs* than the baseline, with drops in the range between 23% and 51%. This indicates that the threads in SC case are 'more useful' in the sense that they spend less time waiting on data to arrive. Thus, despite finer granularity, SC threads have higher average utilisation as can be seen from the load balancing results, and the degree of parallelism is increased, which allows more overlap between communication and computation. Although the total number of fetch messages is increased due to the larger number of threads, for `parpair` and `sumeuler`, the benchmarks that benefit most from SC, the total fetch times are still lower than for the baseline due to reduction in individual fetch times.

5 Related Work

Although popularised by Cilk [5], work stealing was used in earlier parallel implementations of functional languages [6, 18, 27], whilst remaining popular in contemporary implementations (e.g. [10]), as reviewed in a recent survey [32], with locality-awareness being a popular current research direction.

Table 6. Overview of GUM and Related Systems

RTS (Language)	parallelism identification	scheduling	architecture	synchronisation	load balancing
Cilk [5] (C ext.)	explicit (`cilk_spawn`)	LIFO	shared	explicit	work stealing
GHC-SMP [23] (GpH)	annotations (advisory)	FIFO unfair	shared	implicit	work stealing
Manticore [10] (NESL/CML-alike)	impl. data par. expl. task par.	FIFO nestable	shared	implicit	work pushing
X10 [7] (X10)	impl. data par. expl. task par.	PGAS	shared	implicit	work stealing
GUM [31] (GpH)	annotations (advisory)	FIFO unfair	virtual shared	implicit	work stealing
DREAM [20] (Eden)	explicit process instantiation	round robin fair	shared-nothing	implicit	work pushing

Table 6 provides an overview of GUM compared to the most related systems, which together span a wide spectrum of parallel language run-time systems. For more detailed and broader comparisons refer to further literature [3, 2]. With respect to parallelism identification GUM and SMP occupy a unique place in the design space as the annotations provide hints that are advisory rather than mandatory, as is e.g. process instantiation performend in an Eden program, which will lead to a creation of a remote process. Eden and GUM are similar in the architectural respect that unlike other systems they enable distributed execution. On the other hand they differ in the implementation as GUM provides a Global Indirection Table for inter-PE pointers implementing the virtual shared memory abstraction, whilst DREAM uses shared-nothing design and sends data once it is in normal form. Manticore and X10 are somewhat similar in chosing to incorporate both implicit data parallelism and explicit task parallelism, whilst GUM makes no special arrangements for data parallelism and treats expressions requiring data as tasks. There is no agreement on the scheduling style among the systems, Manticore allowing nested schedulers and X10 following PGAS distribution style. GUM and SMP follow the evaluate-and-die model that leads to an unfair design, but helps improve performance by avoiding some overhead.

In all systems thread and memory management are implicit as well as synchronisation, with an exception of Cilk. This allows for a high level of expressiveness, compared to explicit synchronisation and parallelism managment. Despite the popularity of work stealing, some systems have chosen to use work pushing to reduce the amount of communication. This diversity exacerbates the difficulty of directly comparing these systems and languages.

Granularity control is another key consideration for execution of non-strict parallel functional programs [19], both through thread subsumption [25] and explicit application-level specification using thresholding and sophisticated fuel-based algorithms [29] at application or library level. Moreover, work stealing was also shown to benefit from granularity awareness [17].

6 Conclusion

We have introduced *spark colocation,* a work stealing variant that maintains dynamic information about ancestry throughout the execution and uses this information to select sparks that are more closely related to a thief's computation, rather than picking the oldest spark. We report results from five Glasgow parallel Haskell benchmark programs running on a cluster of multi-cores using an extended version of the GUM RTS on up to 256 cores, showing speedup improvements of up to 46% for three of the programs. Examining profiling data suggests that the gain is due to improved load balance and reduced average fetching time, suggesting that related tasks were indeed colocated.

However, the drop in speedup for one less scalable application and one with excessive amounts of overly fine-grained parallelism, suggests that a heuristic could be developed to switch between the baseline and spark colocation depending on both application and architectural characteristics such as the number and computational capability of PEs.

Our mechanism requires minimal programmer overhead, and we argue that it is possible to automatically place annotations by enumerating `par`s and replacing each `par` with `parEnc`, with the corresponding encoding as an argument. As further future work, we would like to investigate different encodings and matching functions to effect granularity in the opposite direction towards a more coarse-grained setting, which becomes useful if the number of PEs is small or parallelism degree is excessive.

Acknowledgements

We are grateful to the anonymous reviewers for comments that have substantially improved the presentation of this paper.

References

1. J. Backus. Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs. *CACM*, 21(8):613–641, 1978.
2. E. Belikov. Language Run-time Systems: an Overview. In *Proc. of Imperial College Computing Student Workshop*, volume 49 of *OpenAccess Series in Informatics (OASIcs)*, pages 3–12. Leibniz-Zentrum fuer Informatik, 2015.
3. E. Belikov, P. Deligiannis, P. Totoo, M. Aljabri, and H.-W. Loidl. A survey of high-level parallel programming models. Technical Report HW-MACS-TR-0103, Department of Computer Science, Heriot-Watt University, December 2013.
4. D. Bevan. An efficient reference counting solution to the distributed garbage collection problem. *Parallel Computing*, 9(2):179–192, 1989.
5. R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Symp. on Principles and Practice of Parallel Programming (PPoPP'95)*, pages 207–216, 1995.
6. F. W. Burton and M. R. Sleep. Executing functional programs on a virtual tree of processors. In *Proceedings of the 1981 conference on Functional programming languages and computer architecture*, pages 187–194. ACM, 1981.
7. P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *ACM SIGPLAN Notices*, volume 40, pages 519–538. ACM, 2005.
8. D. Chase and Y. Lev. Dynamic circular work-stealing deque. In *Proc. of the 17th ACM Symp. on Parallelism in Algorithms and Architectures*, pages 21–28, 2005.
9. A. Church and J. B. Rosser. Some Properties of Conversion. *Transactions of the American Mathematical Society*, 39(3):472–482, 1936.
10. M. Fluet, M. Rainey, J. Reppy, A. Shaw, and Y. Xiao. Manticore: A heterogeneous parallel language. In *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, pages 37–44. ACM, 2007.
11. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine: A User's Guide and Tutorial for networked Parallel Computing*. MIT Press, 1994.
12. K. Hammond. Glasgow parallel Haskell (GpH). In *Encyclopedia of Parallel Computing*, pages 768–779. Springer, 2011.
13. K. Hammond. Why parallel functional programming matters: Panel statement. In *Reliable Software Technologies Ada-Europe*, pages 201–205. Springer, 2011.
14. Z. Hu, J. Hughes, and M. Wang. How functional programming mattered. *National Science Review*, 2(3):349–370, 2015.
15. P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler. A history of Haskell: being lazy with class. In *Proceedings of the third ACM SIGPLAN Conference on History of programming languages*, pages 1–12. ACM, 2007.
16. J. Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, 1989.
17. V. Janjic and K. Hammond. Granularity-aware work-stealing for computationally-uniform grids. In *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, pages 123–134. IEEE, 2010.

18. D. A. Kranz, R. H. Halstead Jr, and E. Mohr. Mul-T: A high-performance parallel Lisp. In *ACM SIGPLAN Notices*, volume 24, pages 81–90, 1989.
19. H.-W. Loidl, P. Trinder, and C. Butz. Tuning task granularity and data locality of data parallel GpH programs. *Parallel Processing Letters*, 11(04):471–486, 2001.
20. R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí. Parallel Functional Programming in Eden. *Journal of Functional Programming*, 15(3):431–475, 2005.
21. S. Marlow. *Parallel and Concurrent Programming in Haskell: Techniques for Multicore and Multithreaded Programming.* O'Reilly, 2013.
22. S. Marlow, P. Maier, H.-W. Loidl, M. Aswad, and P. Trinder. Seq no more: better strategies for parallel Haskell. In *Proc. of the 3rd ACM Symposium on Haskell*, pages 91–102, 2010.
23. S. Marlow, S. L. Peyton Jones, and S. Singh. Runtime support for multicore Haskell. In *ACM SIGPLAN Notices*, volume 44, pages 65–78, 2009.
24. S. Marlow (Ed.). Haskell 2010 language report. 2010. `http://www.haskell.org/onlinereport/haskell2010`.
25. E. Mohr, D. Kranz, R. Halstead Jr, et al. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, 1991.
26. W. Partain. The nofib benchmark suite of Haskell programs. In *Functional Programming, Glasgow 1992*, pages 195–202. Springer, 1993.
27. S. L. Peyton Jones. Parallel implementations of functional programming languages. *The Computer Journal*, 32(2):175–186, 1989.
28. S. L. Peyton Jones, C. Clack, and J. Salkild. High-performance parallel graph reduction. In *International Conference on Parallel Architectures and Languages Europe*, pages 193–206. Springer, 1989.
29. P. Totoo and H.-W. Loidl. Lazy data-oriented evaluation strategies. In *Proc. of 3rd ACM Workshop on Functional High-Performance Computing*, pages 63–74, 2014.
30. P. Trinder, K. Hammond, H.-W. Loidl, and S. L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, January 1998.
31. P. Trinder, K. Hammond, J. Mattson Jr, A. Partridge, and S. Peyton Jones. GUM: a portable parallel implementation of Haskell. In *Proc. of PLDI*, pages 79–88, 1996.
32. J. Yang and Q. He. Scheduling parallel computations by work stealing: A survey. *International Journal of Parallel Programming*, 46(2):173–197, 2018.