JIT Costing Adaptive Skeletons for Performance Portability

Patrick Maier
University of Glasgow
Patrick.Maier@glasgow.ac.uk

John Magnus Morton
University of Glasgow
j.morton.2@research.gla.ac.uk

Phil Trinder
University of Glasgow
Phil.Trinder@glasgow.ac.uk

Abstract

The proliferation of widely available, but very different, parallel architectures makes the ability to deliver good parallel performance on a range of architectures, or performance portability, highly desirable. Irregular parallel problems, where the number and size of tasks is unpredictable, are particularly challenging and require dynamic coordination.

The paper outlines a novel approach to delivering portable parallel performance for irregular parallel programs. The approach combines JIT compiler technology with dynamic scheduling and dynamic transformation of declarative parallelism.

We specify families of algorithmic skeletons plus equations for rewriting skeleton expressions. We present the design of a framework that unfolds skeletons into task graphs, dynamically schedules tasks, and dynamically rewrites skeletons, guided by a lightweight JIT trace-based cost model, to adapt the number and granularity of tasks for the architecture.

We outline the system architecture and prototype implementation in Racket/Pycket. As the current prototype does not yet automatically perform dynamic rewriting we present results based on manual offline rewriting, demonstrating that (i) the system scales to hundreds of cores given enough parallelism of suitable granularity, and (ii) the JIT trace cost model predicts granularity accurately enough to guide rewriting towards a good adaptive transformation.

Keywords parallelism; performance portability; cost model

1. Introduction

The hardware landscape is dominated by parallel architectures — multicores, manycores, clusters, etc. These architectures have very different performance characteristics, e. g. number of processors or communication costs. Applications often hard-code assumptions about the characteristics of their development platform, and thus require significant refactoring when ported to a new parallel architecture. The challenge of *performance portability* is to deliver good parallel performance on a range of architectures with minimal refactoring.

The performance portability challenge is already hard for problems with regular parallelism, i. e. where the number and granularity of tasks is predictable statically. However many important problems exhibit *irregular* parallelism. Examples include sparse matrix operations as used in PDE solvers, algorithms mining large graphs, and core algorithms in computer algebra and symbolic computation. This large class of problems requires dynamic adaptation as the amount of parallelism changes during the computation.

The aim of the *Adaptive Just-In-Time Parallelisation (AJITPar)* project is to investigate a novel approach to deliver *portable parallel performance* for programs with irregular parallelism across a range of architectures. To this end, AJITPar combines declarative parallelism with Just In Time (JIT) compilation, dynamic scheduling, and dynamic transformation.

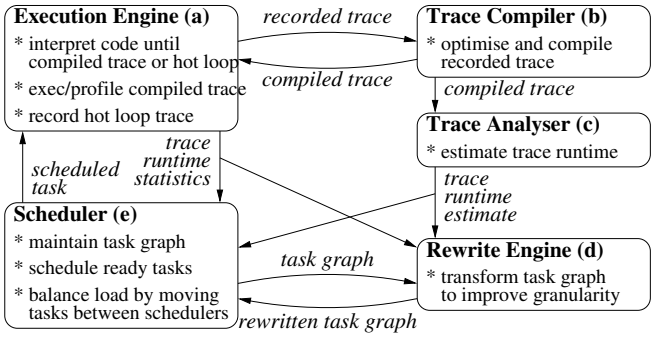

Figure 1. Adaptive Skeletons Execution Framework

AJITPar is based on a library of *Adaptive Skeletons (AS)* for expressing task parallelism. These skeletons [5] are organised into families of parallel patterns, e. g. map, reduction, divide-and-conquer. Crucially for adaptivity, family members may be transformed into each other by means of rewriting according to a set of equations expressing semantic equivalences. As program transformation by equational rewriting is well-established in functional languages, the AS library is implemented in *Racket* [12], a dialect of Scheme.

Dynamic skeleton transformation relies on the ability to dynamically compile code, which is the primary reason for basing the framework on a JIT compiler. Moreover, a trace-based JIT compiler can be extended to estimate task granularity based on lightweight dynamic trace cost analysis, and these estimates can inform both dynamic scheduling and transformation.

Contributions. The paper describes the design of the Adaptive Skeletons (AS) framework (Section 3). Figure 1 shows a block diagram of the components of the framework and their interactions. The *execution engine* and *trace compiler* together make up a traditional trace-based JIT compiler. The *trace analyser* performs cost analysis of hot traces as they are compiled to native code. Trace statistics and costs are fed to the *scheduler*, which uses the information to decide how to schedule tasks. If there is not enough parallelism, or if tasks turn out to be too fine-grain, the scheduler may ask the *rewrite engine* to transform the skeleton in order to adapt parallelism. The rewrite engine uses dynamic trace cost estimates to guide the rewriting towards tasks of suitable granularity and to rank alternative transformations. We detail the unfolding of skeletons into task graphs, the design of the task graph scheduler, and the trace-based cost model.

The paper outlines the design and implementation of the current AS prototype framework (Section 4). By adopting a distributed memory architecture the system can be deployed on a full range of architectures, including multicore, NUMA, and clusters. The AS prototype is built on top of *Pycket* [1], a new trace-based JIT compiler for Racket.

The paper evaluates the AS prototype to discover (1) whether the system scales given enough parallelism of suitable granularity, and (2) whether the trace cost model can predict granularity with sufficient accuracy to guide adaption for a specific architecture using skeleton rewriting (Section 5). For a standard irregular benchmark (SumEuler) on 64 cores, we report good quality task cost predictions for three parallel versions. We demonstrate that the predictions allow the selection of appropriate transformations that improve parallel performance from poor to close to optimal, e. g. improving speedup by a factor of more than 10. On a regular real-world application (k-means clustering) we report scaling to 176 cores on a 256 core cluster, delivering a top speedup of 101. Elsewhere we report good speedups for two out of three further benchmarks on cluster and NUMA architectures [9].

2. Background

Stateless Declarative Parallelism. Many language designs reflect the importance of statelessness for parallelism, e. g. the semantics of a *parallel for loop* in OpenMP is typically only deterministic if the body is stateless. Hence functional languages, where computations are stateless by default, provide an excellent frame into which to embed a stateless parallel coordination language.

Due to AJITPar's range of target architectures, we base our coordination constructs on task-parallel languages for shared- and distributed-memory parallelism, e. g. the Haskell DSL HdpH [8]. Unlike pure task-parallel DSLs, we do not expose the task layer directly. Instead, we provide higher-level abstractions in the form of a library of *algorithmic skeletons* [5]. Importantly, the library does not just provide skeleton implementations but also a framework for transforming skeletons by rewriting.

Others have combined functional skeleton languages with transformation, e. g. [14] rewrites data-parallel skeletons and compiles to OpenCL, Skel [4] is a task-parallel Erlang skeleton library integrated into a re-factoring tool, and the PMLS compiler [13] combined profiling with feedback-directed skeleton rewriting. None of these systems transform skeletons at runtime, as we propose to do.

Just-In-Time Compilers. Dynamic *just-in-time (JIT) compilation* is an established technology for speeding up the execution of virtual machine (VM) interpreters like the HotSpot Java VM.

Trace-based JIT compilers are an emerging technology in this area, particularly popular for dynamic languages, e. g. TraceMonkey [6] for JavaScript, LuaJIT for Lua, or PyPy [3] for Python. Rather than compiling whole method bodies with their complex control structure, trace-based JITs detect, compile and optimise only hot *traces*, e. g. the common path through a loop body. Because traces are straight-line pieces of code without complex control, trace-based JITs aggressively optimise based on highly accurate analyses that aren't feasible in static compilers.

There aren't many trace-based JITs for functional languages yet. Among the few is Pycket [1], a novel tracing JIT for Racket (and often faster than the Racket VM, which runs on a traditional JIT). Being based on PyPy's RPython meta-tracing tool chain [3], Pycket is also one of the most mature functional tracing JITs.

Tracing JITs have not been used much for parallelisation. A note-worthy exception is [15], which explores an auto-parallelising runtime system based on tracing and dynamically transforming binary executables. However, the approach is limited to small multicores and not addressing performance portability.

The simple linear structure of traces makes them ideal targets for lightweight dynamic cost analysis. Augmenting a tracing JIT with such an analysis and investigating its use in guiding the dynamic scheduling and transformation of parallelism is a key novelty of AJITPar.

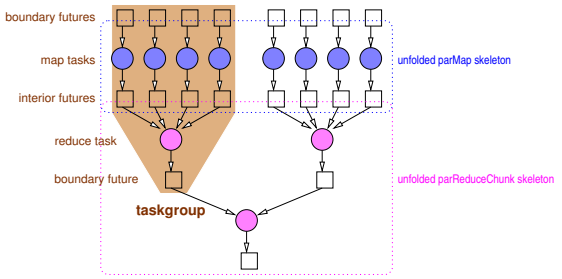

Figure 2. Task graph example.

Transformations. Program transformations are central to optimising compilers. The GHC, for instance, aggressively optimises Haskell code by equational rewriting [11]. Programmers can aid the compiler by supplying hints in the form of rewrite rules; thanks to Haskell's stateless nature many helpful semantic properties of data structures are expressible as simple equations.

Program transformations can also be used to tune parallelism. The PMLS compiler [13], for example, tunes parallel ML code by transforming skeletons based on offline profiling data. While this works well for regular problems, PMLS cannot help with irregular parallelism because it transforms code at compile time rather than at runtime. A similar feedback-directed compile-time approach for rewriting skeletons into OpenCL code [14] has been shown to automatically tune regular linear algebra kernels on GPUs to performance levels comparable with code hand-tuned by expert library developers.

3. Adaptive Skeletons

This section presents the design of the *Adaptive Skeleton* library for expressing and transforming parallelism, and outlines how the framework adapts parallelism to the current execution architecture. Adaptive *skeletons* are based on a standard set of algorithmic skeletons [5] for specifying task-based parallelism within Racket [12]. The AS framework expands skeletons to task graphs and schedules tasks to workers; expansion and scheduling happen at runtime to support tasks with irregular granularity. The AS framework is based on the Pycket [1] trace-based JIT compiler for Racket, and we have developed cost models for Pycket JIT traces and technologies to extract and analyse the costs of the traces after the warm up period. The cost information will be used to guide both the dynamic task scheduler, and a skeleton transformation engine. The latter *adapts* the task granularity of the running program to suit the current architecture by rewriting skeletons according to a standard set of equations.

Although the skeletons are implemented in Racket we choose to present their type signatures and semantics in a Haskell-like syntax for brevity, and in our opinion readability. This section glosses over serialisability, assuming that all types are serialisable, including function types; how serialisation is realised is outlined in Section 4.

3.1 Task graph

Programs are expressed in terms of skeletons rather than individual tasks, thus the functional semantics of programs can be understood without knowledge of its task dependencies. However, the parallel behaviour of a skeleton is best described by providing a translation to a graph making explicit its tasks and their dependencies.

A *task graph* is an acyclic directed bipartite graph, the vertices of which are alternately *tasks* and *futures*, and the edges of which are *dependencies*. Given a task f and futures a and b, an edge from a to f indicates that a is an *input* of f, whereas an edge from f to b indicates that b is an *output* of f.

```
parSumEuler :: [Int] -> Int
parSumEuler = parReduce sum . parMap totient

totient :: Int -> Int
totient n = length [k | k <- [1..n], gcd n k == 1]
```

Figure 4. Parallel SumEuler benchmark.

A future is a storage cell that is initially empty and can be filled once. A task is essentially a function call. A task f is *enabled* if all of its input futures are full, otherwise f is *blocked*. When enabled, f may be *evaluated* by applying the function to the values stored in the input futures; the result of the application fills f's output future.

The AS scheduler distributes *task groups*, i. e. connected subgraphs of the task graph, as depicted in Figure 2. A task group is *enabled* if there is a partial order on its tasks such that no task is blocked when evaluating in order. We call a future *interior* if it is accessed (written or read) exclusively by tasks in a single task group; futures that are shared between task groups are called *boundary*. In our distributed memory execution model we expect that interior futures are read and written using shared memory, while reads and writes of boundary futures require communication. The goal of the AS rewrite engine is to transform the skeletons in the program so that the associated task graph has a sufficient number of task groups, while minimising the number and size of boundary futures.

The AS framework hides the task graph almost completely from programmers. Accessing futures is handled transparently by the system, simply producing their value in case they are full. In case an empty future is accessed, the system suspends evaluation until that future is filled, as elaborated in Section 3.4.

The only primitive exposed to the programmer (or skeleton developer) is spawn which adds a new task to the task graph. Semantically, spawn f a1 ... an is the same as function application f a1 ... an but the implementation transparently promotes the inputs a_i to futures, creates a new future b, and adds a task f with inputs $a_1, \ldots, a_n$ and output b to the task graph.

3.2 Skeletons

Figure 3 introduces some well-known data parallel map and reduce skeletons. The skeletons are specified in a Haskell-inspired equational style over lists yet can be defined over any container type. The function argument to reduce must be an associative reduction operator, i. e. distribute over list concatenation according to the equation g (xs_1 ++ ... ++ xs_n) = g [g xs_1, ..., g xs_n].

The parallel behaviour of parMap and parReduce is defined in terms of spawn. The parMap skeleton produces a flat task graph, that is, no task depends on any other task, and returns a list of futures. In contrast, parReduce only generates a single reduction task, taking a list of futures and returning a single future.

Figure 4 shows a small example expressed using these skeletons, summing up Euler's totient function over a list of integers. This is naturally expressed as a map followed by a reduce, hence parallelised by composing parMap with parReduce.

The example illustrates how skeletons may generate large numbers of fine grain tasks, in this case one task calling totient for each element of the input list. This may hamper performance as the overheads of distributed scheduling dominate task runtime.

The parMap and parReduce skeletons both give rise families of *tunable* skeletons with the same functional semantics yet different parallel behaviour. Typically, these skeletons take extra parameters that govern the number of tasks generated, and hence the task granularity. Figure 3 shows two sample tunable skeletons. Both split their input lists into segments of size k (using function chunk). The parMapChunk skeleton generates tasks that map f over each segment, thereby increasing the task granularity. In contrast, parReduceChunk decreases the task granularity, as it generates a two-level tree of reduce tasks by reducing each segment with g, and then reducing the list of reduction results with g again.

```
      parReduce sum . parMap totient
(6,7)= reduce sum . map totient
  (5)= reduce sum . map totient . concat . chunk 20
  (3)= reduce sum . concat . map (map totient) . chunk 20
  (4)= reduce sum . map (reduce sum) . map (map totient) . chunk 20
  (1)= reduce sum . map (reduce sum . map totient) . chunk 20
(6,7)= parReduce sum . parMap (reduce sum . map totient) . chunk 20
```

Figure 6. Derivation of parallel SumEuler with fused reduction.

The AS framework is designed to be extensible, and a richer set of skeletons with associated transformations is given in [9]. Most importantly, the set of tunable skeletons is not fixed as more tunable skeletons are derivable by equational reasoning (Section 3.3).

3.3 Skeleton transformation by rewriting

The AS framework is designed to adapt the granularity of tasks by transforming the underlying skeletons and Figure 5 presents a set of fairly standard equations for rewriting skeleton-based code. This style of program transformation goes back to Bird's work on algebraic identities [2] in the 1980s. Equations (6) and (7) relate the skeletons in each family to each other. The remaining equations state various laws about the interaction between the sequential map and reduce skeletons and the list transformations chunk and concat, e. g. map fusion (1).

Equations (6) and (7) are derivable and generally sufficient to replace a basic skeleton with a corresponding tunable one. However, rewriting composite skeleton expressions using the remaining equations may produce better results. Figure 6 demonstrates this by deriving a more efficient implementation of the SumEuler benchmark from Figure 4. The derived implementation partially fuses the reduce and map skeletons, and effectively computes the sequential SumEuler function on list segments of size 20 in parallel before summing the results.

3.4 Adaptive execution framework

The adaptive skeleton execution framework combines dynamic scheduling of tasks with adaptive skeleton transformation. The framework employs a master/worker architecture, where the master occupies a single core and each worker occupies a single core. The master is responsible for scheduling and transformation whereas the workers simply execute tasks. Section 4 discusses the implementation of workers in detail; here we focus on how the master adapts task granularity for the current architecture.

Master threads. The master executes three threads, potentially concurrently.

The *evaluator* evaluates the main program sequentially. When evaluating skeleton expressions, it expands the task graph by spawning new tasks as described in Section 3.2. When the evaluator attempts to access an empty future, it will block until that future is filled.

The *scheduler* distributes enabled task groups to idle workers. Decisions on the size of the groups are guided by cost models for computation and communication (see below). The scheduler also monitors execution time and communication overheads of scheduled task groups to establish (i) whether the cost models predict accurately, (ii) how regular tasks are, and (iii) whether most of the tasks fall within the target granularity range. After a warm up period the scheduler reacts to granularity being persistently out of range by signalling the transformer.

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs

parMap :: (a -> b) -> [a] -> [b]
parMap f []     = []
parMap f (x:xs) = spawn f x : parMap f xs

parMapChunk :: Int -> (a -> b) -> [a] -> [b]
parMapChunk k f =
  concat . parMap (map f) . chunk k
```

```
List transformations:
chunk  :: Int -> [a] -> [[a]]
concat :: [[a]] -> [a]
```

```
reduce :: ([a] -> a) -> [a] -> a
reduce g xs = g xs

parReduce :: ([a] -> a) -> [a] -> a
parReduce g xs = spawn g xs

parReduceChunk :: Int -> ([a] -> a) -> [a] -> a
parReduceChunk k g =
  parReduce g . parMap (reduce g) . chunk k
```

Figure 3. Sample skeletons in the map (left) and reduce (right) families; the tunable *Chunk skeletons are derivable by equational reasoning.

```
Fusion:
(1) map g . map f = map (g . f)

Distributivity:
(2) chunk k . map f    = map (map f) . chunk k
(3) map f   . concat   = concat      . map (map f)
(4) reduce g . concat  = reduce g    . map (reduce g)
```

```
Cancellation:
(5) concat . chunk k = id

Skeleton families:
(6) map    = parMap    = parMapChunk k
(7) reduce = parReduce = parReduceChunk k
```

Figure 5. Equational laws about lists and skeleton transformations.

The *transformer* thread executes the rewrite engine that employs a randomised rewrite strategy on the skeletons; a similar strategy has been used successfully to compile skeletons to high-performance OpenCL code [14]. Random rewriting produces several alternative skeleton expressions that are semantically equivalent to the original. The rewrite engine expands (in a similar way to the evaluator) each expression into a task graph in order to predict its runtime using the computation and communication cost models. Finally, the rewrite engine picks the best task graph and signals the scheduler and evaluator to restart the program.

Pragmatics of skeleton rewriting. Transformation is potentially costly, both in terms of time and memory spent on rewriting, task graph expansion and cost analysis, and in terms of work lost due to restarts. Rewriting costs can be limited by grouping rewrite steps into phases and limiting the number of steps per phase as in [14]; expressions that expand to very large task graphs are likely to perform poorly and can be discarded even before cost analysis. Moreover the cost analysis will often be cheap as task graphs usually contain many replicated tasks so the time to analyse individual tasks can be amortised.

The cost of restarts can be controlled by restricting restarts to a warm up phase of a few seconds, and by limiting the number of restarts. The cost of restarts can also reduced if the parallelism is divided into a sequence of phases, e. g. simulation steps, or iterations of k-means clustering. In these cases only the currently active phase needs to be restarted, preserving the work of previous phases.

3.5 Cost models

It may appear that both scheduler and transformer require accurate task computation and communication time predictions *before* a task runs. In fact the scheduler tolerates inaccuracy. Irregularly sized tasks can be scheduled dynamically without accurate information of their expected runtime at the expense of some (usually moderate) overheads — work stealing schedulers typically manage this scenario well. What matters for the scheduler is to capture the ratio of computation to communication in order to select task groups that minimise this ratio.

Similarly, the transformer requires only relative, rather than absolute measures like actual runtimes or latencies. Relative cost predictions will be used to compare alternative transformations of the same skeleton expression. Since the transformations mainly affect the parallel coordination and leave the sequential code largely untouched, consistency of predictions (e. g. two tasks executing almost the same code will have very similar costs) is more important than accuracy.

We have developed simple cost models for predicting the runtime of tasks. These cost models hook into the trace-based JIT compiler (Pycket), intercepting traces after the optimisation phase and just before compiling to native code. The cost of a trace, $\gamma(Tr)$, is computed as the weighted sum of the k instructions in the trace The cost of a task (typically consisting of several loops spanning several traces) can be inferred from the cost of its traces and the value of its trace counters from the Pycket runtime. The weights for each instruction class are determined once for each target architecture by running a suite of benchmarks. The equation below shows the trace cost model parameterised for the cluster measured in section 5.

$$\gamma(Tr) = \sum_{i=1}^{k} \begin{cases} 4.884 \times 10^{-4}, & \text{if } op_i \in numeric \\ 4.797 \times 10^{-3}, & \text{if } op_i \in alloc \\ 4.623 \times 10^{-4}, & \text{if } op_i \in guard \\ 0, & \text{otherwise} \end{cases}$$

The resulting computational cost models are not very accurate in absolute terms but they are consistent and accurately reflect the costs of pre- and post-transformed skeleton expressions [10].

To predict communication overheads, the AS framework employs a simple linear cost model that depends on the type and size of the data communicated, i. e. on the contents of boundary futures. The model reflects that the overheads of TCP dominate the cost of sending small messages, whereas for large messages serialisation dominates (and is linear in the size of the payload). We established the parameters for this model in a series of serialisation experiments reported elsewhere [9]. We omit the formal model here due to lack of space; Section 5.1 compares some of its predicted overheads against overheads measured in the SumEuler benchmark.

4. Prototype Framework Implementation

A prototype adaptive skeleton execution framework to implement the design in Section 3 is under development. This section outlines some key design decisions and the current implementation status.

The current prototype executes task-parallel computations on shared- or distributed-memory architectures using TCP-based message passing. It implements dynamic scheduling and monitors task runtimes and communication overheads. The current system is a fairly conventional distributed-memory parallel functional language implementation; a more detailed discussion can be found in [9].

The prototype extracts task costs using dynamic trace cost analysis based on the model of Section 3.5. That is, we have implemented functional blocks (a), (b), (c) and (e) in Figure 1, but have yet to implement the rewrite engine — block (d). The current implementation is the basis for the preliminary performance evaluation in Section 5, but the transformations are performed manually following the equations in Figure 5.

System Architecture. The runtime environment consists of a central master and multiple workers; each being a separate OS process, possibly on different hosts. The master runs a standard Racket VM, the workers run Pycket. The master maintains the current task graph and schedules enabled task groups to idle workers. Each worker executes task groups, one task at a time, and returns the results to the master. Upon receiving results the master updates the task graph, which may unblock previously blocked task groups.

The decision to adopt a centralised scheduler rather than distributed work stealing was taken with transformations in mind. Distributed schedulers typically lack an accurate global view of current system load and performance, hence it is harder to decide when and how to transform skeletons. Moreover, distributed schedulers tend to produce more random schedules, making it harder to determine whether performance gains are down to good transformations or lucky scheduling.

Tasks and Closures. In contrast to Racket, Pycket currently expects a fixed program at startup and cannot (yet) load code dynamically. To provide the code mobility required in a distributed system, we resort to *explicit closures*, which are essentially static global function pointers, similar to closures in distributed Haskell DSLs like HdpH [8]. Tasks are layered on top, linking closures to input and output futures. Thus, evaluating a task amounts to reading its input futures, evaluating the closure and writing the result to its output future.

Serialisation. Tasks and results are serialised to byte strings for transmission over TCP. We have implemented a fast serialisation library for Pycket and shown that it delivers good performance [9], e. g. consistently beating Racket's own serialisation library.

5. Preliminary Evaluation

This section evaluates the AS prototype implementation investigating (i) whether the lightweight JIT trace based cost model can predict the cost of transformations with reasonable accuracy (Section 5.1), and (ii) whether (a previous version of) the prototype can scale to hundreds of workers (Section 5.2). The performance of further benchmarks (Fibonacci, SumEuler, Mandelbrot, matrix multiplication) on a cluster and two NUMA servers is reported in [9].

All measurements are repeated 7 times and we base speedup calculations on mean runtimes. The experiments are performed on a cluster of 17 16-core servers (2GHz Xeon E5 v2 CPU, 64 GByte RAM, Ubuntu 14.04) connected by 10 Gbit Ethernet. The AS prototype is based on a Pycket snapshot from 23 May 2016[1] built on Racket 6.5 and RPython 5.1.2, except for Section 5.2 evaluating an earlier version based on a snapshot from 10 June 2015[2] built on Racket 6.2 and RPython 2.6.0.

[1] `https://github.com/samth/pycket/commit/89464a4`

[2] `https://github.com/samth/pycket/commit/2f9dc62`

Version	Runtime	Speedup
(1) parSumEuler (Figure 4)	121.0s	4.1
(2) chunked map (chunksize $k = 20$)	10.2s	49.2
(3) chunked map + fused reduction (Figure 6)	10.1s	50.0

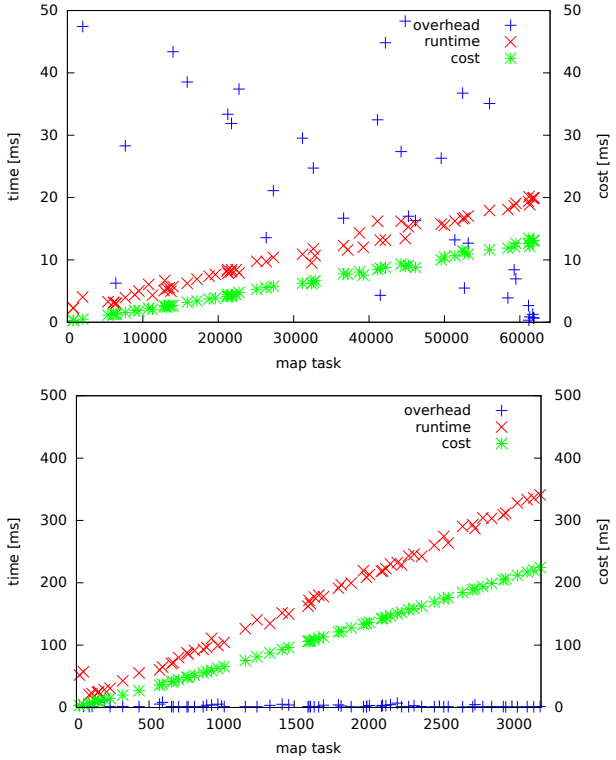

Figure 7. Cost, runtime and overheads of 64 randomly selected SumEuler tasks: version 1 (top) versus version 2 (bottom).

5.1 Effectiveness of the JIT trace cost model

To evaluate the usefulness of the trace cost model (Section 3.5) for predicting how task granularity changes when transforming skeletons, we perform a series of experiments computing the sum of Euler's totient function over an interval of 64000 integers. The skeletons are transformed manually offline, and not automatically during execution. The sequential runtime is 503s; we display parallel runtimes and speedups of three parallel versions on 64 workers on top of Figure 7.

The table shows that version 1, the untransformed parSumEuler, a parallel map followed by a reduction, performs poorly. The scheduler can detect poor performance early on. It starts by scheduling a random selection of enabled tasks, and continuously monitors task runtimes, costs and overheads, i. e. the time to serialise data, sent it over TCP and deserisalise it at the other end.

The top graph of Figure 7 shows a distribution of runtimes, costs and overheads of some 64 tasks, typical of what the scheduler might encounter for version 1. Here we have discarded runtime and cost data of the very first tasks (which are tainted by JIT warm up effects). Tasks appear on the x-axis in the order they were created during the unfolding of the parMap skeleton. Runtimes appear to increase linearly (if noisily) by up to an order of magnitude, as do costs, showing a reasonable correlation between the two quantities. The trace cost model systematically underestimates measured runtime, as it does not account for some overheads. A linear cost correction based on observed runtimes could improve accuracy. In contrast, the measured overheads are extremely variable, and many are off the scale, indicating that the scheduler is overwhelmed by

small tasks. Using a simple statistical analysis that is robust against outliers, e. g. taking the median, the scheduler decides whether to transform version 1 in the hope of finding a better one. In this case, median cost is 5.7ms and median runtime 9.8ms, but median overhead is an excessive 44ms (despite tasks only communicating small amounts of data). Hence the scheduler triggers a skeleton rewrite by waking the transformer.

As a first attempt, the transformer may rewrite the skeleton expression parReduce sum . parMap totient by replacing parMap with parMapChunk k to produce version 2. What value of k should the transformer choose? Aiming for an average granularity of 100 to 200ms, the transformer picks $k = 20$ and predicts a median granularity of 114 to 196ms. The transformation does not change the amount of data communicated, yet it does reduce the total number tasks by a factor 20, which should reduce pressure on the scheduler and bring down overheads.

The second row of the table in Figure 7 demonstrates how this simple transformation has improved performance ten-fold. The bottom graph of Figure 7 shows the distribution of runtimes, costs and overheads of a random sample of 64 tasks of this transformed version. The data looks less noisy and the overheads have come down. This is confirmed by statistical analysis: median cost is 108ms, median runtime 174ms and median overhead 1.4ms. At this point the scheduler may well decide that the ratio of computation to communication is appropriate and run to completion.

Further improvements are possible however. If the transformer performs more elaborate rewrites, e. g. the ones detailed in Figure 6, it will find a third version that partially fuses the reduction with the map. Comparing the task graphs for versions 2 and 3 (both with chunksize $k = 20$), the transformer finds both appear similar. Both feature 3200 map tasks, yet they differ in the final reduce task. Version 2 reduces 64000 integers whereas version 3 reduces only 3200 integers. Consequently, our communication cost model predicts communication costs for the reduce tasks of 66ms for version 2 and 5.4ms for version 3. This is broadly consistent with observed reduce task overheads of 124ms and 10.3ms, respectively. As a result of the lower communication overhead, version 3 exhibits a marginally better speedup (see Figure 7).

Systematic experiments with chunksizes show that $k = 50$ is optimal for this instance of SumEuler, resulting in a speedup of 53.6. This demonstrates that the cost models can effectively guide the rewrite engine to a version whose performance is close to optimal.

5.2 Small Case Study: K-means on a Distributed Platform

To assess the performance of an earlier AS prototype on a realistic benchmark application, we implement the iterative refinement algorithm, or Lloyd's algorithm, for k-means clustering. Given a classification of N d-dimensional data points into k clusters, an iteration of the algorithm refines the classification by re-classifying each data point to the cluster whose centroid is nearest. Then the algorithm re-computes the cluster centroids based on the refined classification, and starts the next iteration. The algorithm terminates when the classification becomes stable.

Each iteration consists of a parallel map (re-classifying data points) and reduce (re-computing centroids). The map is chunked and partially fused with the reduction, similar to the SumEuler example. Since each iteration depends on the centroids computed in the previous round, the parallel algorithm synchronises after each iteration by broadcasting the new centroids (as a $k \times d$ matrix).

We benchmark on a data set of 4.9 million 40-dimensional data points, derived from a data set of KDD-99 data mining competition [7]. We classify into $k = 250, 500, 1000$ clusters. To compare timings, we stop after 20 iterations (instead of running to convergence). We measure wall-clock time, excluding system startup and time taken to read the input. To measure scaling we vary the number of workers from 16 to 256.

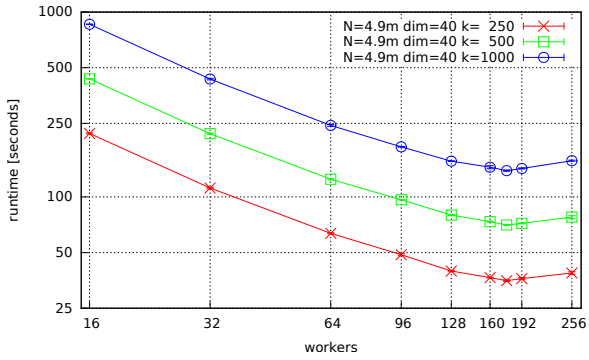

Figure 8. Runtimes of 20 iterations of k-means.

Figure 8 shows a log/log plot of parallel runtimes depending on the number of workers. Sequential runtime is linear in k and is 14039 seconds for $k = 1000$ (which is 2.4× slower than a sequential implementation in C). For each k, the system scales to 160 and peaks at 176 workers, with a top speedup of 101×, and 42× over sequential C. Performance drops upwards of 192 workers due to too many Pycket instances (more than 12 per 16-core node) competing for the memory bus.

6. Discussion

We have outlined a novel approach to delivering portable performance for irregularly parallel programs that combines declarative parallelism with JIT technology, dynamic scheduling, and dynamic transformation. If the approach is successful it may help improve the performance portability of many languages with tracing JIT-compilers.

We have presented the design of an adaptive skeleton (AS) library with a task graph implementation, JIT trace costing, and transformations that adapt skeletons for parallel architectures (Section 3). We have sketched the current state of the prototype implementation (Section 4).

The results of the preliminary evaluation of the prototype are encouraging (Section 5). The system scales given enough parallelism of suitable granularity, e. g. Figure 7 reports a maximum speedup of 50 on 64 workers for SumEuler, and Figure 8 a maximum speedup of 101 on 176 workers for the k-means case study. Moreover Figure 7 shows that the trace cost model can predict granularity with sufficient accuracy to guide program transformation, and hence adaption, for a specific architecture.

The prototype AS implementation has some performance issues, as detailed in [9], but these are readily addressed. Given that the skeletons were manually transformed to produce the SumEuler versions reported in Figure 7, the key remaining technical challenge is to implement the skeleton rewrite engine to automate the transformations. If effective this will enable fully automatic adaption to the underlying architecture during program execution. The potential of the approach can then be assessed by evaluating the performance of the full AS framework on both benchmarks and case studies, and on a range of parallel architectures.

Acknowledgments

This work is funded by UK EPSRC grant AJITPar (EP/L000687/1).

References

[1] S. Bauman, *et al.* Pycket: A tracing JIT for a functional language. In *ICFP '15*, pages 22–34. ACM, 2015.

[2] R. S. Bird. Algebraic identities for program calculation. *Comput. J.*, 32(2):122–126, 1989. doi: 10.1093/comjnl/32.2.122.

[3] C. F. Bolz, *et al.* Tracing the meta-level: PyPy's tracing JIT compiler. In *ICOOOLPS '09*, pages 18–25. ACM, 2009.

[4] C. Brown, *et al.* Cost-directed refactoring for parallel Erlang programs. *Int. J. Parallel Prog.*, 42(4):564–582, 2014.

[5] M. I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989.

[6] A. Gal, *et al.* Trace-based just-in-time type specialization for dynamic languages. In *PLDI 2009*, pages 465–478. ACM, 2009.

[7] KDD. KDD cup 1999 data, 1999. URL `https://archive.ics.uci.edu/ml/machine-learning-databases/kddcup99-mld/kddcup99.html`.

[8] P. Maier, R. Stewart, and P. W. Trinder. The HdpH DSLs for scalable reliable computation. In *Haskell 2014*, pages 65–76. ACM, 2014.

[9] P. Maier, J. M. Morton, and P. Trinder. Towards an adaptive skeleton framework for performance portability. Technical Report TR-2016-001, School of Computing Science, University of Glasgow, 2016. URL `http://www.dcs.gla.ac.uk/~pmaier/AJITPar/`.

[10] J. M. Morton, P. Maier, and P. Trinder. JIT-based cost analysis for dynamic program transformations. In *RAC2016, Eindhoven, The Netherlands*, 2016.

[11] S. L. Peyton Jones. Compiling Haskell by program transformation: A report from the trenches. In *ESOP '96*, LNCS 1058, pages 18–44. Springer, 1996.

[12] Racket. Racket programming language. URL `http://racket-lang.org/`.

[13] N. Scaife, *et al.* A parallel SML compiler based on algorithmic skeletons. *J. Funct. Program.*, 15(4):615–650, 2005.

[14] M. Steuwer, *et al.* Generating performance portable code using rewrite rules. In *ICFP '15*, pages 205–217. ACM, 2015.

[15] E. Yardimci and M. Franz. Dynamic parallelization and vectorization of binary executables on hierarchical platforms. *Journal of Instruction-Level Parallelism*, 10:1–24, 2008.