University of Glasgow, School of Computing Science, Technical Report TR-2016-001

Towards an Adaptive Skeleton Framework 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. Irregularly-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 irregularly parallel programs. The approach combines declarative parallelism with JIT technology, dynamic scheduling, and dynamic transformation.

We present the design of an adaptive skeleton library, with a task graph implementation, JIT trace costing, and adaptive transformations. We outline the architecture of the protoype adaptive skeleton execution framework in Pycket, describing tasks, serialisation, and the current scheduler. We report a preliminary evaluation of the prototype framework using 4 micro-benchmarks and a small case study on two NUMA servers (24 and 96 cores) and a small cluster (17 hosts, 272 cores). Key results include Pycket delivering good sequential performance e.g. almost as fast as C for some benchmarks; good absolute speedups on all architectures (up to 120 on 128 cores for sumEuler); and that the adaptive transformations do improve performance.

Keywords parallelism; performance portability; JIT compiler

1. Introduction

The general purpose hardware landscape is dominated by parallel architectures — multicores, manycores, clusters, etc. These architectures have very different performance characteristics e.g. the number of processors or communication costs. A key element of the *multicore software crisis* is a lack of abstraction: most parallel code mixes coordination and computation at the expense of clarity and maintainability. Worse still, coordination often hard-codes assumptions about the architecture. Hence the program requires significant refactoring for 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 size of tasks can be statically predicted. However many important problems exhibit *irregular* parallelism. Examples include sparse matrix operations as used in PDE solvers, algorithms mining large graphs, and core algorithms used in computer algebra and symbolic computation. This large class of problems requires coordination strategies that dynamically adapt 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. The approach we propose combines declarative parallelism with Just In Time (JIT) compilation, dynamic scheduling, and dynamic transformation.

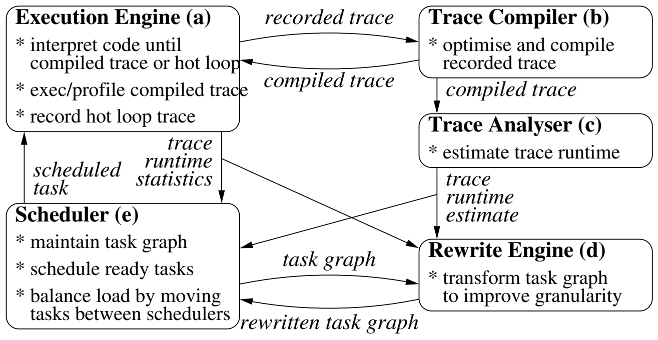

Figure 1. Adaptive Skeletons Execution Framework

We aim to investigate the performance portability potential of an *Adaptive Skeletons* (AS) library based on task graphs, and an associated parallel execution framework that dynamically schedules and adaptively transforms the task graphs. We express common patterns of parallelism as a relatively standard set of algorithmic skeletons [7], with associated transformations. Dynamic transformations, in particular, rely 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 deliver estimates of task granularity by dynamic profiling and/or dynamic trace cost analysis, and these can be exploited by the dynamic scheduler. We chose a trace-based JIT-compiled functional language as functional programs are easy to transform; dynamic compilation allows a wider range of transformations including ones depending on runtime information; and trace-based JIT compilers build intermediate data structure (traces) that may be costed.

Figure 1 shows a functional block diagram of the adaptive skeletons execution framework, showing the interaction between its various components. The *execution engine* and *trace compiler* together make up a traditional trace-based JIT compiler, with the added capability of profiling hot traces. The *trace analyser* performs cost analysis of hot traces as they are compiled to native code. Profiler and trace analyser feed their data to the *scheduler*, which uses the information to decide where to schedule which parallel tasks. If the scheduler cannot find enough parallelism, or if the parallel tasks turn out to be too fine-grain, the scheduler may call the *rewrite engine* to attempt to transform individual tasks or the whole task graph.

The AS framework is being engineered in *Pycket* [2, 5], a very new trace-based JIT compiler for Racket, itself a Scheme dialect. Currently Pycket only supports a strict subset of Racket and does not yet support any of the parallel, concurrent and distributed features of full Racket. Our original intention was to develop the framework for a trace-based JIT compiler for Haskell, but were unable to in the absence of a sufficiently mature Haskell JIT compiler.

To investigate performance portability the AS framework must support a range of different parallel architectures. In particular, the prototype framework currently supports standard multicores, (4 to 16 cores) manycore servers (i. e. NUMA servers with up to 100 cores) and small clusters of multicores (with a few hundred cores in total).

The paper makes the following research contributions after surveying related work (Section 2). We outline a novel approach to delivering portable parallel performance for programs with irregular parallelism in the form of the design of an adaptive skeleton library with a task graph implementation, JIT trace costing, and transformations that adapt skeletons for parallel architectures (Section 3). We outline the Pycket protoype adaptive skeleton execution framework, describing tasks, serialisation, and the current scheduler (Section 4). We report a preliminary evaluation of the prototype framework using 4 micro-benchmarks and a small case study on two NUMA servers (24 and 96 cores) and a small cluster (17 hosts, 272 cores). Key results include the following. Pycket delivers good sequential performance e. g. almost as fast as C for benchmarks with substantial amounts of numerical computation, and never more than 3.5 times slower. The protoype framework achieves good absolute speedups on all architectures for all but one program, e. g. maximum speedups of 117 for SumEuler on 96 cores (with two hardware threads each) and 55 for k-means clustering on 128 cores (spread over 16 hosts). Crucially, the adaptive transformations do improve parallel performance (Section 5).

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. Indeed there is a movement of programming languages towards a stateless "functional second" model as exemplified by languages like C# and JavaScript.

For the purposes of AJITPar functional languages have two key advantages over imperative or object-oriented languages.

- Typically, substantial program fragments are stateless. This ensures determinism even for speculative parallelism where computations may be aborted, restarted or replicated.
- An equational theory of program equivalence that justifies program transformations based on rewriting. For example, the standard Haskell compiler *GHC*, winner of the 2011 ACM SIGPLAN Programming Languages Software Award for (among other achievements) its "efficiency", relies heavily on equational rewriting during its optimisation phase.

We base our coordination constructs on task-parallel languages, e. g. MonadPar [15] or HdpH [14], Haskell DSLs for expressing task parallel computations on shared- and distributed-memory platforms, respectively. On top of the (often) low-level primitives of these DSLs sits a library of *algorithmic skeletons* [7], providing high-level abstraction to the application programmer.

An approach to tuning code to specific architectures by transforming high-level coordination constructs like skeletons into semantically equivalent ones with different coordination behaviour was validated by the PMLS compiler [20]; a similar idea of refactoring skeletons is has also been pursued in *ParaPhrase* (EU FP7-288570). However, neither ParaPhrase nor PMLS transform code at runtime, as we propose to do.

Just-In-Time Compilers. The challenge of portable binaries has arisen in other contexts, e. g. for Java web apps running on a diverse and unpredictable set of architectures. Therefore, Java compilers generate machine-independent bytecode, which is designed to be interpreted by a virtual machine (VM). To gain execution speed, many VMs translate bytecode to native code on-the-fly, a technique called *just-in-time (JIT) compilation*. Over the last 15 years, JIT compilers have been developed for many languages, e. g. Mozilla's TraceMonkey [10] for JavaScript, PyPy [6] for Python, and Microsoft's SPUR project [3] for .NET bytecode. JITs are beginning to emerge for statically typed functional languages. For Haskell, there are already several JIT compilers in prototype state, e. g. PyHaskell [25] and *lambdachine* [21, 22]. The standard Racket VM has long been powered by a JIT compiler. Moreover, Pycket [2, 5] is a novel JIT for Racket (often faster than the Racket VM) based on the PyPy tool chain.

An emerging technology in this area is *trace-based* JIT compilation; in fact all of the systems mentioned above apart from the standard Racket VM are trace-based JITs. Rather than compiling whole function or 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. Moreover, traces often span several static scopes, so trace-based JITs can perform inlining and accurate inter-procedural analysis for free.

The idea to build an auto-parallelising runtime system (RTS) on top of a trace-based JIT has been proposed in [23, 27], but only in the context of fine-grained implicit parallelism on small multicores. However, implicit parallelism is unlikely to scale to systems with large numbers of cores or deep memory hierarchies because it does not help the RTS navigate the vast space of potential parallelism. Instead, we propose to limit the search space by relying on the programmer to annotate parallel code (often termed *semi-implicit* parallelism), yet guiding the RTS's efforts to auto-tune parallelism with code transformations supplied by the programmer.

An alternative approach, exemplified by languages like OpenCL and SaC [11], is to perform specialisation of architecture-independent data parallelism at runtime by selecting from a small number of pre-compiled code variants. Instead, we propose a more flexible system that can explore an unbounded space of variants.

Code transformations. Program transformations are central to optimising compilers. The GHC Haskell compiler, for instance, aggressively optimises Haskell code by equational rewriting [17, 18]. Haskell programmers can aid the compiler by supplying hints in the form of rewrite rules; thanks to Haskell's purity many helpful semantic properties of data structures are expressible as simple equational rules.

Besides optimisation, program transformations can be used to tune parallelism. The PMLS compiler [20], 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 approach [24] based on rewrite rules transforming algorithmic skeletons into OpenCL code has been shown to automatically tune regular linear algebra kernels on GPUs to performance level comparable with code hand-tuned by expert library developers.

In Java, bytecode transformations are a common way to extend the language without changing the compiler or the VM. For instance, Java bytecode rewriting has been used to enforce resource bounds [8], or to transparently run applications on distributed-memory architectures [26]. Typically, such transformations preserve bytecode structure, e. g. replacing a class by a class, and can be implemented by simple bytecode traversal. Transformations that alter the bytecode structure, e. g. rewriting loops to optimise database queries [12], are far more complex. However, some complex transformations can be implemented in expressive rule-based rewrite frameworks [1].

Java bytecode rewriting is typically performed ahead of runtime, so it cannot help a program adapt to irregular parallelism. Instead, we propose a novel approach where code is (potentially continually) rewritten at runtime by a JIT rewrite engine, that applies equational rules guided by online profiling and cost analysis.

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 [7] for specifying task-based parallelism within Racket [19]. 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 piggybacks on Pycket [2], a trace-based JIT compiler for Racket, to analyze the cost of tasks as they are executed. The cost information is used both to guide the dynamic task scheduler as well as 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 detailed 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 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*. We use letters $f, g, \dots$ to range over tasks and $a, b, \dots$ to range over futures. 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.

A future a is a storage cell that is either *empty* or *full*; in the latter case, a stores a value. A future can be filled only once, that is, the value stored in a cannot be updated, and any attempts to do so are silently ignored.

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 results of the application fill f's output futures.[1] We generalise the notion of enabledness to subgraphs of the task graph as follows. A subgraph is *enabled* if there is a partial order on its tasks such that no task is blocked when evaluating according in order.

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 3.5.

The only primitive exposed to the programmer (or skeleton developer) is spawn which adds a new task to the task graph. Semantically, spawn f x1 ... xn is the same as function application f x1 ... xn but the implementation

[1] In Racket functions can have multiple return values, analogously tasks can have multiple output futures.

```
parFib :: Int -> Int
parFib = parDivconq
           (\n -> if n < 2 then [] else [n-1,n-2])
           (reduce (+) 0)
           (const 1)

parSumEuler :: Int -> Int -> Int
parSumEuler l u =
  parReduce (+) 0 $ parMap totient [l..u]
    where
      totient :: Int -> Int
      totient n = length [k | k <- [1..n], gcd n k == 1]
```

Figure 3. Examples: parallel Fibonacci and SumEuler.

- Checks whether each input x_i is a future; if not, creates a new future a_i and fills it with x_i, else aliases a_i to x_i.
- Creates new futures $b_1, \dots, b_m$ where m is the output arity of f.
- Adds task f with inputs $a_1, \dots, a_n$ and outputs $b_1, \dots, b_m$ to the task graph.

3.2 Basic skeletons

Figure 2 introduces three basic adaptive skeletons, the well-known data parallel skeletons map and reduce and the control-parallel skeleton divconq, which abstracts the divide-and-conquer paradigm.

The skeletons are specified in a Haskell-inspired equational style over lists despite the fact that they may be defined over any container type. The arguments to reduce must form a monoid, that is, g is an associative operation with neutral element z; more specialised skeletons may also require g to be commutative.

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 builds a binary tree of tasks and returns a single future. Similarly, parDivconq returns a single future from which a tree hangs. The arity of the tree depends on the divide function, its leaves are conquer tasks, and its inner nodes are combine tasks.

Examples. Figure 3 shows two small code examples expressed using the adaptive skeletons. The naive recursive algorithm to compute Fibonacci numbers is a well-known micro-benchmark for measuring the overhead of function calls. It follows the divide-and-conquer pattern, so can naturally parallelised using parDivconq. SumEuler is another micro-benchmark, summing up Euler's totient function over an interval of integers. It is naturally expressed as a map followed by a reduce, hence parallelised using parMap and parReduce.

The examples show that the skeletons can be composed sequentially as in SumEuler. However, skeletons should not be nested, that is, their arguments should not call skeletons themselves. This is restriction is not enforced though; in fact, the system will execute nested skeletons and spawn new subtasks. Yet, due to the distributed architecture of the AS framework (see sections 3.5 and 4), subtasks spawned during the parallel execution of a task on a worker need to be sent to the central scheduler, which increases communication overheads.

3.3 Skeleton families

The examples in Figure 3 also illustrate how the skeletons may generate large numbers of fine grain tasks. For instance, parFib generates an exponential number of conquer and combine tasks, yet

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

reduce :: (a -> a -> a) -> a -> [a] -> a
reduce g z []         = z
reduce g z [x]        = x
reduce g z (xs ++ ys) = g (reduce g z xs)
                          (reduce g z ys)

divconq ::
  (a -> [a]) -> ([b] -> b) -> (a -> b) -> a -> b
divconq div comb conq x =
  case div x of
    [] -> conq x
    ys -> comb (map (divconq div comb conq) ys)
```

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

parReduce :: (a -> a -> a) -> a -> [a] -> a
parReduce g z []         = z
parReduce g z [x]        = x
parReduce g z (xs ++ ys) = spawn g (parReduce g z xs)
                                   (parReduce g z ys)

parDivconq ::
  (a -> [a]) -> ([b] -> b) -> (a -> b) -> a -> b
parDivconq div comb conq x =
  case div x of
    [] -> spawn conq x
    ys -> spawn comb (map (parDivconq div comb conq) ys)
```

Figure 2. Specification of basic skeletons (right) and their sequential counterparts (left).

each combine task only adds two numbers, and each conquer task returns the constant 1. What is needed are skeletons that generate a smaller number of larger tasks.

Each of the three basic skeletons gives rise to a family of *tunable* skeletons (Figure 4) with the same functional semantics but different parallel behaviour. Typically, the tunable skeletons accept an extra parameter that allows to tune the number of parallel tasks generated, thereby reducing the number of tasks while increasing their computational granularity.

The skeletons in the map family reduce the number of tasks by having each task apply function f to a "sublist" of the input list sequentially. This sublists are constructed by dividing the input list into chunks of size k (function chunk k), or by traversing the input list with stride k (function transpose . chunk k). In either case, parallelism is actually generated by calling the parMap skeleton. The output list (of futures) is constructed by reversing the transformation of the input (using concat or concat . transpose).

The reduce family admits two approaches to control granularity. Sublists may be generated by chunking or striding as for map, with sequential reduction of the sublist results. Alternatively, the input list can be subdivided (typically in the middle) 2^k times by unfolding the call tree of reduce to depth k and spawning 2^k sequential reduction tasks. As parReduceStride reorders the input list, it is only safe to use with commutative reduction operators g.

The divide-and-conquer family offers a depth-bounded skeleton, similar to the depth-bounded reduce skeleton, and a thresholding skeleton. The latter also unfolds the recursive call tree of divide-and-conquer but spawns a sequential divide-and-conquer task as soon as the input reaches a certain threshold (i. e. the predicate thresh becomes true). Both skeletons also spawn conquer tasks encountered while unfolding the top levels of the call tree but they don't spawn combine tasks. This rests on the assumption that conquer tasks are computationally heavier than combine tasks; the family could be extended to cater for cases when this assumption does not hold.

3.4 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 [4] in the 1980s. The right column relates the skeletons in each family to each other and to their sequential counterpart. The left column presents various laws about the interaction between the sequential map and reduce skeletons and the auxiliary list operations chunk, concat and transpose , e. g. map fusion (3). Note that the dot (.) in these equations denotes function composition.

While the equations on the right of Figure 5 are generally sufficient to replace a single instance of an inefficient basic skeleton with a more efficient tunable one, the equations on the left are needed to rewrite composite skeleton expressions. Figure 6 demonstrates this by means of a rewrite derivation of a more efficient implementation of the SumEuler benchmark (Figure 3) rewriting modulo associativity of function composition. The derived implementation devides the input list into "sublists" by striding, then parMaps the sequential SumEuler function over the list of sublists and sequentially reduces the results by summation.

Another transformation of SumEuler might use rules (11) and (12) to arrive at the simpler **reduce** (+) 0 . **parMapStride** k totient, which also controls task granularity by striding. However, this expression is not as effcient as the one derived previously because the derivation in Figure 6 uses map fusion to combine computing the totient of a sublist with a reduction. Thus, each task in Figure 6 returns a single integer instead of a list of integers, thereby reducing communication overheads.

3.5 Adaptive execution framework

This section sketches how 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 transformations whereas the workers simply execute work as assigned by the master. Section 4 discusses the implementation of workers in detail, and we now focus on the master detailing how it adapts task granularity to suit the hardware architecture.

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

- The *evaluator* thread evaluates the main program sequentially. When it evaluates skeletons, it expands the task graph by spawning new tasks as described in section 3.2 and 3.3. When the evaluator attempts to access an empty future, it will block until that future is filled.
- The *scheduler* thread schedules enabled tasks or task subgraphs to idle workers. Decisions on the size of the subgraphs scheduled are guided by cost models for computation and communication (see below). The scheduler also monitors the execution time and communication overheads of scheduled tasks to establish (i) whether the cost models predict accurately, (ii) how regular tasks are, and (iii) whether most of the tasks fall within

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

parMapStride :: Int -> (a -> b) -> [a] -> [b]
parMapStride k f xs = concat $ transpose $ parMap (map f) $ transpose $ chunk k xs

parReduceChunk :: Int ->  (a -> a -> a) -> a -> [a] -> a
parReduceChunk k g z xs = reduce g z $ parMap (reduce g z) $ chunk k xs

parReduceStride :: Int ->  (a -> a -> a) -> a -> [a] -> a
parReduceStride k g z xs = reduce g z $ parMap (reduce g z) $ transpose $ chunk k xs

parReduceDepth  :: Int -> (a -> a -> a) -> a -> [a] -> a
parReduceDepth 0 g z xs         = spawn (reduce g z) xs
parReduceDepth k g z []         = z
parReduceDepth k g z [x]        = x
parReduceDepth k g z (xs ++ ys) = g (parReduceDepth (k-1) g z xs) (parReduceDepth (k-1) g z ys)

parDivconqDepth :: Int -> (a -> [a]) -> ([b] -> b) -> (a -> b) -> a -> b
parDivconqDepth 0 div comb conq x = spawn (divconq div comb conq) x
parDivconqDepth k div comb conq x = case div x of
                                      [] -> spawn conq x
                                      ys -> comb (map (parDivconqDepth (k-1) div comb conq) ys)

parDivconqThresh :: (a -> Bool) -> (a -> [a]) -> ([b] -> b) -> (a -> b) -> a -> b
parDivconqThresh thresh div comb conq x = if thresh x
                                           then spawn (divconq div comb conq) x
                                           else case div x of
                                                  [] -> spawn conq x
                                                  ys -> comb (map (parDivconqThresh p div comb conq) ys)

chunk :: Int -> [a] -> [[a]]             concat :: [[a]] -> [a]             transpose :: [[a]] -> [[a]]
```

Figure 4. Tunable skeletons (bottom line: signatures of auxiliary functions chunk, concat, transpose).

```
Cancellation:
(1) concat . chunk k = id
(2) transpose . transpose = id

Fusion:
(3) map g . map f = map (g . f)

Distributivity:
(4) chunk k     . map f   = map (map f) . chunk k
(5) map f       . concat = concat       . map (map f)
(6) reduce g z  . concat = reduce g z   . map (reduce g z)

Reordering:
(7) transpose . map (map f) = map (map f) . transpose
(8) transpose . reduce g z . map (reduce g z)
      = reduce g z . map (reduce g z)
      = reduce g z . map (reduce g z) . transpose,
      if g is commutative
```

```
Map family:
 (9) map = parMap
(10) parMap = parMapChunk k
(11) parMap = parMapStride k

Reduce family:
(12) reduce = parReduce
(13) parReduce   = parReduceChunk k
(14) parReduce g = parReduceStride k g, if g is comm.
(15) parReduce   = parReduceDepth k

Divide-and-conquer family:
(16) divconq = parDivconq
(17) parDivconq = parDivconqDepth k
(18) parDivconq = parDivconqThresh p
```

Figure 5. Equational laws about lists and skeleton transformations.

```
      parSumEuler
    = parReduce (+) 0 . parMap totient
(12)= reduce (+) 0 . parMap totient
 (9)= reduce (+) 0 . map totient
 (1)= reduce (+) 0 . map totient . concat . chunk k
 (5)= reduce (+) 0 . concat . map (map totient) . chunk k
 (6)= reduce (+) 0 . map (reduce (+) 0) . map (map totient) . chunk k
 (8)= reduce (+) 0 . map (reduce (+) 0) . transpose . map (map totient) . chunk k
 (7)= reduce (+) 0 . map (reduce (+) 0) . map (map totient) . transpose . chunk k
 (3)= reduce (+) 0 . map (reduce (+) 0 . map totient) . transpose . chunk k
 (9)= reduce (+) 0 . parMap (reduce (+) 0 . map totient) . transpose . chunk k
```

Figure 6. Example: Derivation of efficient strided parallel SumEuler.

the target granularity range suitable for the hardware architecture. After a warm up period the scheduler reacts to granularity being persistently out of range by signaling the transformer.

- The *transformer* thread repeatedly rewrites the main program's skeletons following a randomized rewrite strategy; a similar strategy has recently been used successfully to compile skeletons to high-performance OpenCL code [24]. Random rewriting produces several different skeleton expressions that are semantically equivalent to the original. The transformer 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 transformer picks the best task graph and signals the scheduler and evaluator to restart the program.

Pragmatics of skeleton transformation. 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. The following heuristics limit the cost.

The cost of computing transformations can be kept in check by grouping rewrite steps into phases and limiting the number of steps per phase, as in [24]. Expressions that expand to very large task graphs can be discarded even before cost analysis on the grounds that the huge number of tasks would imply low granularity. Cost analysis itself ought to 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.

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 subgraphs that minimize 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.

Complementary to the work reported here, the AJITPar project is developing 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. Predicting the cost of individual traces requires a single pass over the trace computing a weighted sum of all instructions. The cost of a task (typically consisting of one or more loops and spanning several traces) can be inferred from the cost of its traces and the value of its trace counters (which can be obtained from the Pycket runtime). 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 expressiions [16].

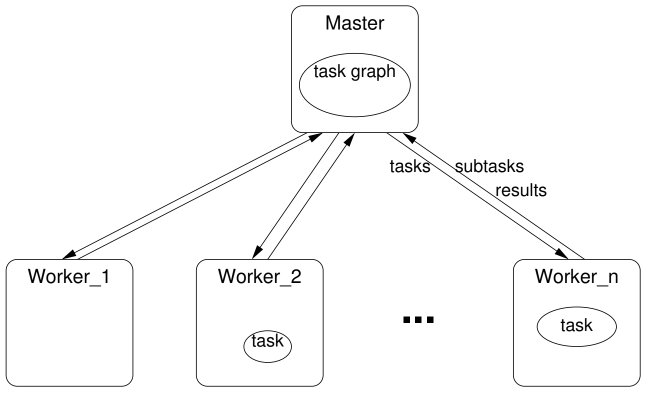

Figure 7. Prototype Adaptive Skeleton execution framework.

The overheads of communication are proportional to the size of the data communicated, i. e. to the size of data in futures. This is easy to establish for full futures (i. e. for task inputs). To estimate the size of task outputs, the AS framework will rely on extrapolation based on observations during the warm up phase. Static size analysis might improve predictions (though such an analysis is complicated by the fact that Racket is not statically typed).

Having described the design vision of the adaptive skeleton execution framework, the next sections describes the current state of the implementation.

4. Prototype Framework Implementation

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

The current prototype executes task-parallel computations on shared or distributed-memory architectures using TCP-based message passing, serialisation, futures, tasks, basic skeletons, task evaluation, dynamic task scheduling, and monitors task runtimes and communication overheads. This implementation forms the basis for the preliminary performance evaluation in Section 5. Cost modelling for Pycket JIT traces has also been implemented, but not yet integrated into the prototype framework [16].

Not yet implemented are cost analysis of task graphs, cost-model guided scheduling of subgraphs of the task graph, cost-model guided transformation of skeletons by rewriting, and support for cancellation and restart of computations.

4.1 System Architecture

Figure 7 shows the scheduling architecture of the adaptive skeleton framework. It consists of a central master and n workers; each of these is realised as a separate OS process, possibly on different hosts. The master runs a standard Racket VM, the workers run Pycket. Since trace-based cost models (which depend on Pycket) are not yet integrated workers can currently also run Racket; we use this feature in Section 5 to compare scaling of Racket vs Pycket.

The master maintains the current task graph and schedules enabled tasks to idle workers. Each worker executes tasks, one at a time, and returns the result to the master. Upon receiving a result the master updates the task graph, which may unblock previously blocked tasks. Workers may also generate subtasks (e. g. following a divide-and-conquer pattern) which they pass back to the master for scheduling.

The master and workers behave much like actors, i. e. they do not share state, are single threaded and communicate by sending messages over TCP connections. In part, these design choices are born out of the restrictions of Pycket, which does not (yet) support concurrency. However, they also simplify the implementation of workers, which execute a simple receive-eval-send loop. Nonetheless, there are drawbacks compared to a shared-memory design:

- TCP-based message passing can add significant latency, particularly for large messages.
- All messages need to be serialised by the sender side and deserialised by the receiver, which can cause significant overhead for large messages. (In fact, Section 4.3 demonstrates that serialisation dominates the cost of message passing.)

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 whether to transform skeletons. Moreover, distributed schedulers tend to produce more random schedules, which makes it harder to evaluate whether performance gains are down to good skeleton transformations or lucky scheduling.

4.2 Tasks and Closures

As a dynamic language Racket supports code mobility (e. g. tasks moving from master to workers) through the ability to dynamically load and execute code. However, one of the many current restrictions of Pycket rules out dynamic code loading — Pycket expects a fixed program at startup. As a result mobile code has to be realised by using a crutch, namely *explicit closures* similar to the Haskell DSLs HdpH [13] and CloudHaskell [9]. An explicit closure is a pair consisting of a uniquely named (hence serialisable) global function pointer and list of serialisable arguments.

Tasks are layered on top of closures, linking closures to input and output futures. Thus evaluating an enabled task amounts to reading its input futures, evaluating the closure and writing the results to the output futures; this is the essence of a worker's receive-eval-send loop.

4.3 Serialisation

Tasks and futures (results) must be serialised to byte strings that can be transmitted over TCP sockets. Racket offers a serialisation library for this purpose but the library does not work in Pycket. Hence we have implemented our own serialisation library, specifically designed to serialise tree-like data structure fast. (In fact, the library relies on data being acyclic; attempting to serialise cycles will likely result in the system live-locking.)

We have benchmarked the serialisation library on a number of typical data structures, including binary trees, vectors and matrices. We have also compared the performance of serialisation in Pycket to the performance in other languages, including Haskell and Java. Here, we reproduce the throughput measurements for serialising triangular matrices of 64-bit integers, see Figure 8.

We observe that the Pycket throughput is consistently the best — it is only beaten by Java throughput when serialising very large data structures. (This is likely a result of Java's memory manager being better able to handle very large objects.) Yet, we also observe that serialisation throughput never exceeds 600 MBit/s (measured on a 5 year old Intel Xeon CPU at 2GHz). That is, serialisation throughput is an order of magnitude lower than the throughput of modern networking hardware (10 Gbit/s). Hence, for large messages communication latency will be dominated by the time taken to serialise and deserialise messages.

platform	network	RAM/host	cores/host	cores total
AMD NUMA server	-	512GB	24	24
AMD NUMA cluster	1Gbit	512GB	32	96
Intel cluster	10Gbit	64GB	16	272

Table 1. Benchmark platforms.

5. Preliminary Evaluation

We evaluate the performance of the prototype AS execution framework on a number of micro-benchmark and a small case study application. The goals of this evaluation are to demonstrate

- that the framework scales on different architectures (Sections 5.2 and 5.4),
- that the framework scales to hundreds of workers provided task granularity is suitable (Sections 5.2 and 5.3), and
- that skeleton transformations can improve performance (Section 5.5).

5.1 Benchmarks Applictions and Platforms

We consider the following (micro-)benchmarks applications.

1. *Fibonacci.* Naively computes the n-th Fibonacci number:
$$fib(n) = fib(n-1) + fib(n-2) \ .$$
2. *SumEuler.* Computes the sum of Euler's totient function φ over an interval of n integers:
$$\mathit{SumEuler}(n) = \Sigma_{i=1}^{n}\varphi(i) \ .$$
3. *Mandelbrot.* Computes a hi-res black-and-white image of the Mandelbrot set according to the escape time algorithm with an iteration limit of 1000 and where complex numbers are emulated by pairs of double precision floating point numbers.
4. *Matrix multiplication.* Multiplies two square matrices of double precision floats using the naive cubic time algorithm.
5. *K-means clustering* Classifies a large data set high-dimensional points (represented as vectors of double precision floating point numbers) according to Lloyd's iterative refinement algorithm for k-means clustering.

Fibonacci is implemented using a divide-and-conquer skeleton, the other benchmarks rely on data-parallel skeletons. K-means clustering exhibits a sequence of parallel phases, the other benchmarks consist of a single parallel phase. Fibonacci, matrix multiplication and k-means clustering are regular problems whereas SumEuler and Mandelbrot show moderate irregularity.

Table 1 summarises characteristics of the three evaluation platforms: a 24-core NUMA server, a cluster of three 32-core NUMA server, and a cluster of 17 16-core servers. The latter platform is based on Intel Xeon CPUs, the former two on AMD Opterons.

The experiments reported here were performed using a snapshot of Pycket obtained on 10 June 2015, extended with TCP bindings and built on Racket 6.2 and RPython 2.6.0. All experiments were repeated 7 times; we base speedup calculations on mean runtimes.

5.2 Strong Scaling Experiments on a Distributed Platform

To demonstrate scaling of the AS framework up to hundreds of workers, we investigate strong scaling of the micro-benchmarks Fibonacci, SumEuler and Mandelbrot on the AMD NUMA cluster. For these experiments we pick input parameters to yield sufficiently large sequential runtimes (between 137 and 499 seconds) and tune the skeletons to produce a suitable number (about 4000) of tasks. To be precise, we compute

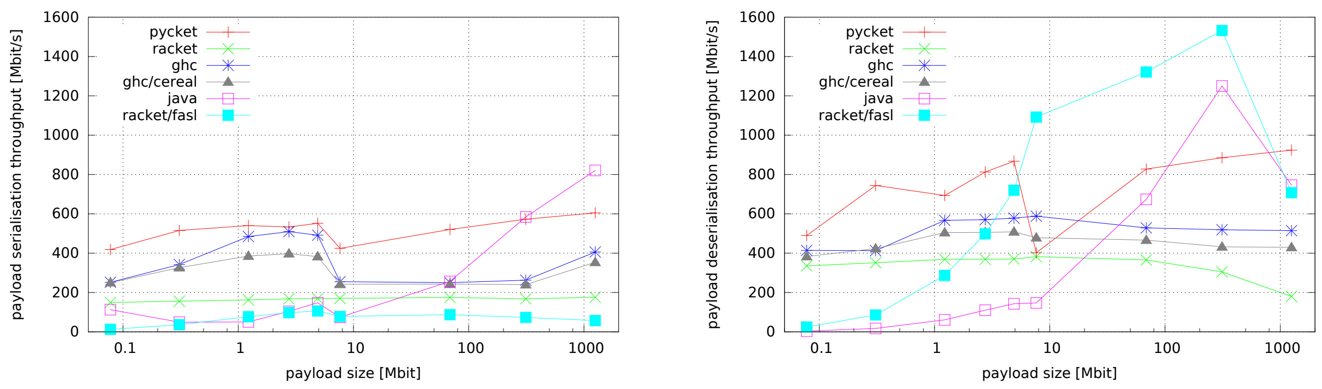

Figure 8. Triangular matrices of integers: serialisation throughput (left) and deserialisation throughput (right).

- Fibonacci of $n = 50$ using a thresholding skeleton cutting off at 33.
- SumEuler of $n = 2^{16}$, striding through the interval in steps of 4000.
- Mandelbrot at a resolution 10000×8000, chunking the image into 4000 horizontal stripes (of resolution 10000×2).

The top row of Figure 9 shows log/log plots of the runtimes and absolute speedups of these benchmarks in the prototype AS framework. All problems scale similarly up to 90 workers, with speedups of 63 to 73 and with moderate variability (as witnessed by the smallness of the error bars). Only once we move to 180 workers (i. e. making full use of the platform's two hardware threads per core) do the graphs diverge. SumEuler, and to a lesser extent Mandelbrot, scale quite well to 180 workers; Fibonacci, however, does not. However all benchmarks still improve performance beyond 90 workers, with a top speedup of 98.7 for SumEuler on 180 workers. This shows that hardware threads can alleviate some of the drawbacks of our centralised work stealing architecture, keeping workers busy that would otherwise have been idle waiting for the master to assign more tasks.

For reference, we performed the same experiments with workers running Racket instead of Pycket, shown in the bottom row of Figure 9. We found the sequential performance of Racket and Pycket comparable; Racket outperforms Pycket by a margin of at most 20% on Fibonacci and Mandelbrot, however Pycket beats Racket by about 15% on SumEuler. Noteworthy is that Racket generally scales better than Pycket when using all available hardware threads, achieving a top speedups of 116.7 for SumEuler on 180 workers.

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

To assess the performance of the system on a realistic benchmark application, we implement the iterative refinement algorithm (also known as 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.

Parallelism is introduced by sub-dividing the data sets into equal sized chunks such that each chunk can be classified in parallel; the resulting tasks are highly regular. Since each iteration depends on the results of the previous one, the parallel algorithm is required to synchronise after each iteration, before generating a new set of parallel tasks.

We run the algorithm on the Intel cluster, scaling up to 8 workers per node distributed over 16 nodes; the 17th node is reserved exclusively for the master.

As inputs we used 6 synthetic data sets with N points in d dimensions, where N varies from $250,000$ to $1,000,000$ and d varies from 10 to 40. We classify each data set into 250 clusters. To obtain comparable results, we run the clustering algorithm for a fixed 20 iterations (instead of until convergence). We measure wall-clock time taken for clustering, excluding system startup time and time taken to read the input. To measure scaling we vary the number of workers per node (up to 8 workers/node, i. e. 128 workers in total).

Figures 10 shows log/log plots of runtimes and speedups depending on the total number of workers available. The algorithm scales well initially, reaching a speedup of 29.4 on 32 workers on the biggest problem (based on a sequential runtime of 294.7 seconds). However, it does not scale as well as we increase the number of workers per node to 4 and higher, reaching a top speedup of 54.9 on 128 workers. One reason for the diminished speedup may be problem size; smaller problems reach their scaling limits already at 64 workers. Another reason may be Amdahl's law. Because of the synchronisation after each iteration the algorithm essentially alternates between sequential and parallel phases. Yet another reason may be effects of the parallel hardware. A separate experiment comparing the runtime of the biggest problem on 64 workers spread over 16 nodes versus 64 workers spread over 8 nodes reveals a slowdown of about 16% when doubling the number of workers per node.

We also briefly compare the performance of Racket vs Pycket on k-means clustering. On this numerically intensive application, Racket is about 2 to 2.5 times slower than Pycket in sequential execution. The system does scale better with Racket workers, reaching a top speedup of 75.5 for the biggest problem. However, improved scaling is not able to fully compensate for slower sequential execution; at peak performance the system with Racket workers is still 30% slower than with Pycket workers.

5.4 Strong Scaling on a NUMA Server

An earlier version of the AS framework did not support communication via TCP. Instead, this version relied on Unix IPC constructs (named pipes) and was hence limited to run the master and all workers on the same physical host. We used this version to systematically explore strong scaling of the micro-benchmarks Fibonacci, SumEuler, Mandelbrot and matrix multiplication across a range of 5 to 6 different values for the respective skeleton tuning parameters. These experiments were performed on the 24-core AMD NUMA

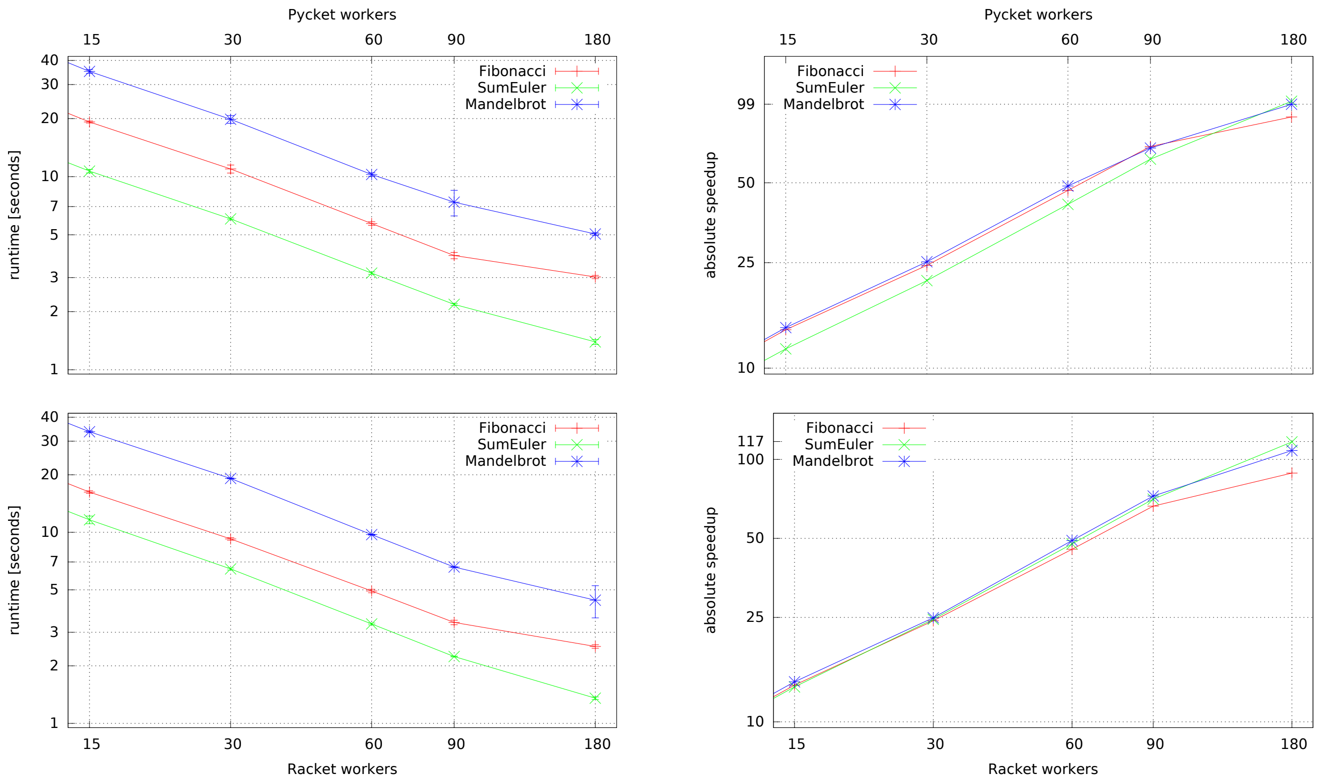

Figure 9. Runtimes (left) and speedups (right) on AMD NUMA cluster; workers run Pycket (top row) or Racket (bottom row).

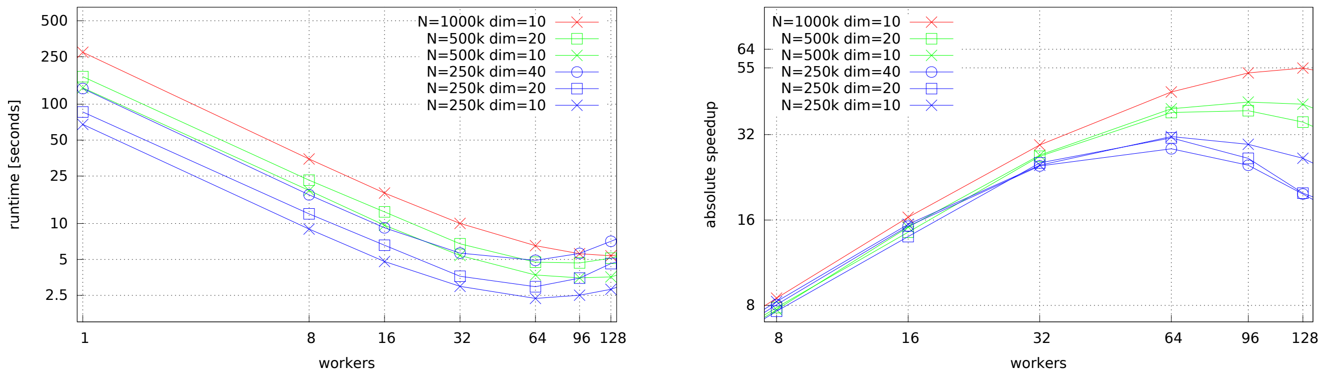

Figure 10. Runtimes (left) and speedups (right) of 20 iterations of k-means on Intel cluster.

platform. The snapshot of Pycket was obtained on 14 May 2015 and built on Racket 6.1.1 and RPython 2.5.1.

To establish a baseline, we compare the sequential runtime of each benchmark on Pycket with the runtime of a sequential implementation in C. To determine strong scaling, we compare sequential runtime on Pycket with parallel runtimes on the AS framework (which runs Pycket on workers), scaling the number of workers (and hence the number of cores utilised). Runtimes measured exclude system startup and shutdown but include JIT warmup. During the experiments the server was not entirely unloaded, which is why we limit scaling to 21 cores maximum (1 core for the master, up to 20 cores for workers).

Fibonacci Pycket takes 60.8 seconds to compute the 47th Fibonacci number. The C implementation is about 3.5 times faster at 17.8 seconds.

Figure 11 shows the speedup graphs for parallel Fibonacci computations using a divide-and-conquer skeleton with different thresholds for falling back on sequential computations. The thresholds were chosen such that the granularity of the sequential tasks varies by more than two orders of magnitude, from less than 10 milliseconds to more than one second.

Scaling is best for tasks in the medium granularity range, though overall the system does not scale well with top speedup of only 14.3 on 20 workers, despite the very low communication overheads of the Fibonacci computation. One reason for this might be the centralised master in the AS framework architecture. It is likely

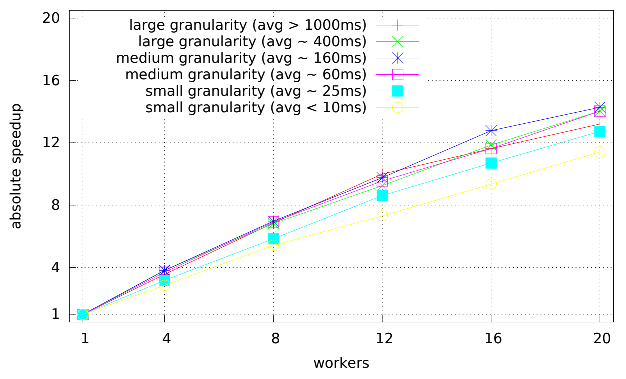

Figure 11. Speedup of Fibonacci on AMD NUMA server.

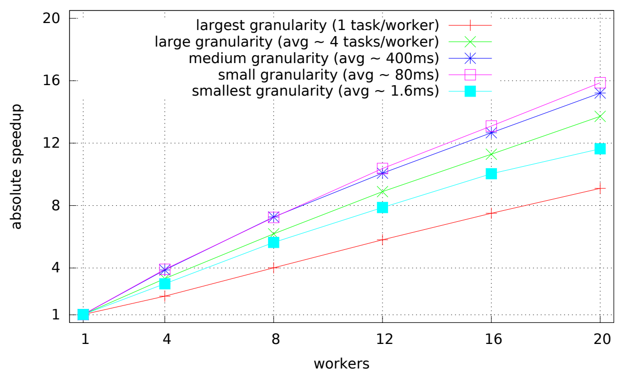

Figure 12. Speedup of SumEuler on AMD NUMA server.

that distributed work stealing would improve the performance of divide-and-conquer parallelism.

SumEuler Pycket takes 82.5 seconds to compute the SumEuler of $n = 50,000$. The C implementation is about 30% faster at 62.5 seconds.

Figure 12 shows the speedup graphs for the parallel computations using a parallel-map task farm skeleton and dividing the input interval into even sized chunks. The chunksize (and hence the task granularity) varies. At one extreme, each chunk contains only a single integer, resulting in 50,000 tasks with an average granularity of 1.6 milliseconds. At the other extreme there are exactly as many chunks as there are workers, resulting in an average task granularity of several seconds.

Since the problem is irregular — the runtime of each task depends on the length of its input interval as well as on the size of the numbers in the interval — very large tasks result in poor scaling due to imbalanced load. Very small tasks scale better but not optimally due to the scheduling overheads (which are around 0.1 milliseconds per task even for tasks with very small communication overhead). Tasks with a medium granularity scale best with a top speedup of 15.9 on 20 workers.

Mandelbrot Pycket takes 115.7 seconds to compute the Mandelbrot image with a resolution of 5000×4000 pixels. The C implementation is about 30% faster at 90.9 seconds.

Figure 13 shows the speedup graphs for the parallel computations using a parallel-map task farm skeleton to compute horizontal stripes of the Mandelbrot image. The width (i.e. number of rows of pixels) of the stripes varies, and with it the task granularity. The smallest stripes (1 single row of pixels) result in an average

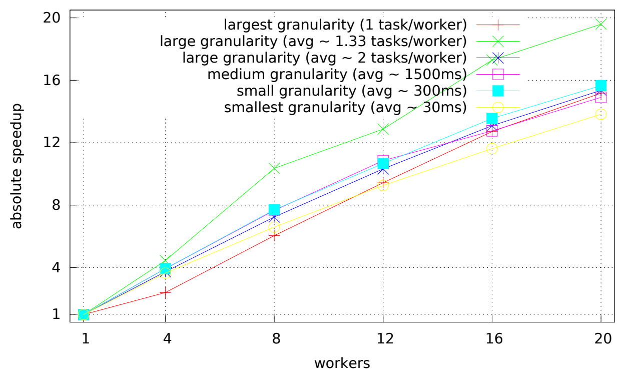

Figure 13. Speedup of Mandelbrot on AMD NUMA server.

task granularity of about 30 milliseconds. At the other extreme, the number of stripes matches the number of workers, resulting in an average task granularity of several seconds. We note that the input of each task is small: 5 integers describing the position and size of the stripe plus the iteration limit. However, Mandelbrot tasks produce a moderate amount of output data, namely at least 1 bit per pixel. That is, even the smallest tasks produce about 5000 bits, or 625 bytes. The total amount of output data that needs to be transmitted from workers to the master is $5000 \cdot 4000$ bits, or about 2.5 megabytes.

Mandelbrot is an irregular problem as stripes at the top and the bottom of the picture are generally faster to compute than stripes in the middle. Nonetheless, scaling appears to be fairly unaffected by task granularity, with a few exceptions. One exception is the scaling of tasks of the largest granularity (i. e. exactly one task per worker) at 4 and 8 workers, where irregularity does cause noticeable load imbalances. The other exception is deterioration of scaling when task granularity is smallest (i. e. a single row of pixels) due to increased scheduling overheads. Overall, tasks with a moderate average granularity of around 300 milliseconds appear to scale best, achieving a speedup of 15.7 on 20 workers.

The speedup graphs in Figure 12 also show an outlier soaring above all others. According to the data, dividing the Mandelbrot image into $\frac{4}{3}$ as many stripes as there are workers results in a speedup of 19.6 on 20 workers; the speedups are even super-linear at lower numbers of workers, e. g.. 10.3 on 8 workers. We do not currently have a convincing explanation for this phenomenon.

Matrix multiplication Pycket takes 75.3 seconds to multiply two matrices of dimension 3200. The C implementation is about 20% faster at 62.5 seconds.

Figure 14 shows the speedup graphs for parallel matrix multiplication, computing the rows of the product matrix in parallel using a parallel-map task-farm skeleton. Computing a single row of the output takes on average about 25 milliseconds time, i. e. the smallest possible task granularity is 25 milliseconds. To increase granularity, tasks may compute blocks of rows; Figure 14 also shows the speedups for tasks computing blocks of 10 rows, blocks of 50 rows, and blocks of a size such that there are 4 tasks per worker, or 1 task per worker.

Tasks need to communicate only a small amount of input data, like in the Mandelbrot benchmark, since the input matrices are read from file by each worker simultaneously (and the time for reading inputs is excluded from the runtime). However, tasks produce a substantial amount of output that must be communicated back to the master. In total, all tasks together produce about $8 \cdot 3200^2$ bytes, or about 80 megabytes. The current prototype system struggles with the throughput at these data sizes, taking about 5 seconds

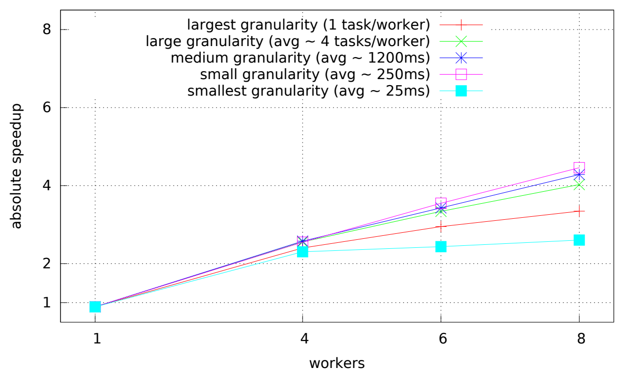

Figure 14. Speedup of matrix multiplication on AMD NUMA server (up to 8 workers).

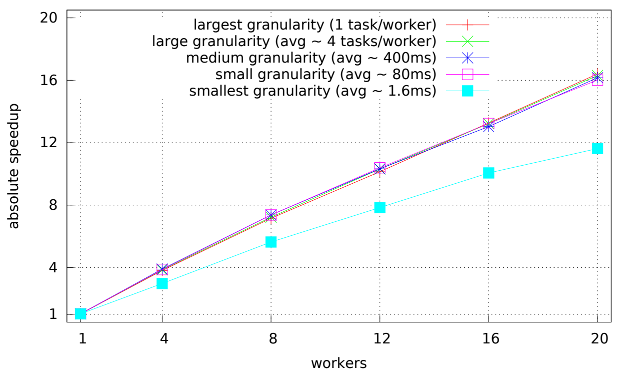

Figure 15. Speedup of strided SumEuler on AMD NUMA server.

to serialise, transmit and deserialise the product matrix. Besides slowing parallel computations down, this overhead also appears to cause significant noise in parallel runtimes, varying by as much as 50%, which is much higher than in the other benchmarks.

We limit the experiments for scaling matrix multiplication to 8 workers — beyond that parallel runtimes are too noisy to draw any conclusion from the data. We observe that tasks with middling granularities of around 250 milliseconds scale best, with a top speedup of 4.4 on 8 workers. The speedup is depressed by the time taken to transmit the product matrix back to the master, which accounts for a third of the parallel runtime on 8 workers. If transmitting the final product were omitted (or transparently relegated to a high-performance distributed key value store) the speedup on 8 workers would rise to 6.7.

As in the other benchmarks, low granularity tasks suffer from increased scheduling overheads. Surprisingly, very high granularity tasks also scale badly, despite matrix multiplication being a very regular problem. This may be explained by noise in the data — fewer tasks per worker appear to significantly increase the variance in parallel runtimes of matrix multiplication.

5.5 Impact of Transformations on a NUMA Server

Each of the benchmarks above evaluates the scaling of several versions of the same algorithm, differing only in the number of parallel tasks and their granularity. These versions were generated by (manually) transforming the original parallel algorithm into one that takes an extra parameter controlling the parallelism:

- a threshold in the case of the divide-and-conquer benchmark (Fibonacci),
- a chunk or block size in the case of the data parallel benchmarks.

The results of Section 5.4 show that transformations do have an impact on the scalability of parallel code. In the case of data parallel problems, in particular, the untransformed code often generates tasks that are too fine-grain and, hence incurring high overheads.

As Section 3.4 details there are other transformations that might be applicable to the benchmarks in Section 5.4. For instance, data parallel computations might subdivide their input by striding rather than chunking, effectively re-ordering the computation. Figure 15 shows the effect such a re-ordering transformation on SumEuler. In this case, each task iterates over the input interval with a given *stride*, computing Euler's φ function on all integers encountered. That is, the transformation groups together integers whose difference is a multiple of the stride; the total number of tasks equals the stride. The effect of this transformation on SumEuler is to render tasks more regular because the magnitude of the numbers in each task is approximately the same (and the cost of computing φ is determined by the magnitude of its argument). The benefit of more regular task sizes is evident in Figure 15 which shows that all strided versions of SumEuler scale virtually the same way,[2] regardless of task granularity. Moreover, all strided versions achieve higher speedups (between 16.0 and 16.4 on 20 workers) than even the best of the chunked versions.

For these measurements, we manually transformed the parallel code and manually selected suitable threshold, chunksize and stride parameters to tune the transformed code for the particular architecture. The aim of the full AS execution framework is to effectively automate this process.

- Admissible transformations are specified by the programmer Since transformations are often tied to particular skeletons, it is most likely that the transformations are specified by the skeleton library developer. Transformations are expressed in a domain specific language (e. g. as skeleton transforming rewrite rules, Section 3.4).
- The AS framework will select applicable transformations and rank the transformed code according to its cost models (Section 3.4). Where transformations require extra parameters (e. g. thresholds, strides, etc.) cost analysis will also aid in picking suitable values.
- The AS framework will abort the current computation and restart a transformed program if it finds the granularity of the current task graph unsuitable (i. e. either too large or too small) and if the framework has produced a graph with better (predicted) granularity.

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 improve 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 outlined the Pycket prototype AS execution framework, describing tasks, serialisation, and the current scheduler (Section 4).

[2] The exception is the smallest granularity. However, this is essentially the untransformed code, where each task computes φ of just one integer, and scales exactly as the smallest granularity of chunked SumEuler.

The results of the preliminary performance evaluation of the prototype AS framework are encouraging (Section 5). Pycket delivers good sequential performance e.g. almost as fast as C for some benchmarks, and never more than 3.5 times slower. The protoype framework achieves good absolute speedups on all architectures for all but one program. Crucially, the adaptive transformations do improve parallel performance.

The evaluation does, however, reveal several limitations of the prototype AS framework. (1) The centralised master scheduler in the framework can become a bottleneck and limit scalability beyond a hundred workers; it is also a poor fit for some skeletons like divide-and-conquer. This could be addressed by adding distributed work stealing. (2) Communication costs are high due to serialisation and Unix IPC overheads (Section 4.3), restricting the performance of programs with high communication volumes e. g. matrix multiplication. This could be addressed by maintaining large data structures in some distributed store, e.g. a key-value store.

Performance issues aside, the key future objectives are to engineer and evaluate the full AS framework as follows. To build the dynamic skeleton transformation engine into the framework. To use the JIT trace costs to guide the transformations of the skeleton programs to adapt to the underlying architecture. The potential of the approach can then be assessed by evaluating the performance of the full AS framework on both benchmarks and case studies.

Acknowledgments

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

References

[1] E. Balland, P.-E. Moreau, and A. Reilles. Bytecode rewriting in Tom. In *Bytecode 2007*, ENTCS 190(1), pages 19–33, 2007.

[2] S. Bauman, C. F. Bolz, R. Hirschfeld, V. Kirilichev, T. Pape, J. G. Siek, and S. Tobin-Hochstadt. Pycket: A tracing JIT for a functional language. In *ICFP '15*, 2015.

[3] M. Bebenita, *et al.* SPUR: a trace-based JIT compiler for CIL. In *OOPSLA 2010*, pages 708–725. ACM Press, 2010.

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

[5] C. F. Bolz, T. Pape, J. Siek, and S. Tobin-Hochstadt. Meta-tracing makes a fast Racket. In *Workshop on Dynamic Languages and Applications (DYLA '14)*, 2014.

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

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

[8] G. Czajkowski and T. von Eicken. JRes: A resource accounting interface for Java. In *OOPSLA '98*, pages 21–35. ACM Press, 1998.

[9] J. Epstein, A. P. Black, and S. Peyton-Jones. Towards Haskell in the cloud. In *Haskell '11, Tokyo, Japan*, pages 118–129. ACM Press, 2011.

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

[11] C. Grelck and S.-B. Scholz. SAC – a functional array language for efficient multi-threaded execution. *International Journal of Parallel Programming*, 34(4):383–427, 2006.

[12] M.-Y. Iu and W. Zwaenepoel. Queryll: Java database queries through bytecode rewriting. In *Middleware 2006*, LNCS 4290, pages 201–218. Springer, 2006.

[13] P. Maier and P. W. Trinder. Implementing a high-level distributed-memory parallel Haskell in Haskell. In *IFL 2011*, LNCS 7257, pages 35–50. Springer, 2012.

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

[15] S. Marlow, R. Newton, and S. L. Peyton-Jones. A monad for deterministic parallelism. In *Haskell 2011, Tokyo, Japan*, pages 71–82. ACM Press, 2011.

[16] J. M. Morton, P. Maier, and P. Trinder. Costing JIT traces. Technical Report TR-2015-001, School of Computing Science, University of Glasgow, 2015.

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

[18] S. L. Peyton Jones, A. Tolmach, and T. Hoare. Playing by the rules: Rewriting as a practical optimisation technique in GHC. In *Haskell 2001*, pages 203–233, 2001.

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

[20] N. Scaife, S. Horiguchi, G. Michaelson, and P. Bristow. A parallel SML compiler based on algorithmic skeletons. *J. Funct. Program.*, 15 (4):615–650, 2005.

[21] T. Schilling. Challenges for a trace-based just-in-time compiler for Haskell. In *IFL 2011*, LNCS 7257, pages 51–68. Springer, 2012.

[22] T. Schilling. *Trace-based Just-in-time Compilation for Lazy Functional Programming Languages*. PhD thesis, University of Kent, 2013.

[23] W. Schulte and N. Tillmann. Automatic parallelization of programming languages: Past, present and future. In *IWMSE10*, 2010. Keynote.

[24] M. Steuwer, C. Fensch, C. Dubach, and S. Lindley. Generating performance portable code using rewrite rules. In *ICFP '15*, 2015.

[25] E. W. Thomassen. Trace-based just-in-time compiler for Haskell with RPython. Master's thesis, NTNU, Trondheim, Norway, Jan. 2013.

[26] E. Tilevich and Y. Smaragdakis. Portable and efficient distributed threads for Java. In *Middleware 2004*, LNCS 3231, pages 478–492. Springer, 2004.

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