Scheduling light-weight parallelism in ArTCoP

J. Berthold[1], A. Al Zain[2], and H-W. Loidl[3]

[1] Fachbereich Mathematik und Informatik
Philipps-Universität Marburg, D-35032 Marburg, Germany
berthold@mathematik.uni-marburg.de
[2] School of Mathematical and Computer Sciences
Heriot-Watt University, Edinburgh EH14 4AS, Scotland
ceeatia@macs.hw.ac.uk
[3] Institut für Informatik, Ludwig-Maximilians-Universität München, Germany
hwloidl@tcs.ifi.lmu.de

Abstract. We present the design and prototype implementation of the scheduling component in ArTCoP (architecture transparent control of parallelism), a novel run-time environment (RTE) for parallel execution of high-level languages. A key feature of ArTCoP is its support for deep process and memory hierarchies, shown in the scheduler by supporting light-weight threads. To realise a system with easily exchangeable components, the system defines a *micro-kernel*, providing basic infrastructure, such as garbage collection. All complex RTE operations, including the handling of parallelism, are implemented at a separate system level. By choosing *Concurrent Haskell as high-level system language*, we obtain a prototype in the form of an executable specification that is easier to maintain and more flexible than conventional RTEs. We demonstrate the flexibility of this approach by presenting *implementations of a scheduler for light-weight threads* in ArTCoP, based on GHC Version 6.6.
Keywords: Parallel computation, functional programming, scheduling.

1 Introduction

In trying to exploit the computational power of parallel architectures ranging from multi-core machines to large-scale computational Grids, we are currently developing a new parallel runtime environment, ArTCoP, for executing parallel Haskell code on such complex, hierarchical architectures. Central to the design of ArTCoP is the concept of *deep memory and deep process hierarchies*. The system uses different control mechanisms at different levels in the hierarchy. Thus, data access and presence of parallelism can be transparent to the language level. For the memory management this provides a choice of using explicit data distribution or virtual shared memory. For the process management this means that units of computation are very light-weight entities, and we explicitly control the scheduling of these units. In this paper we focus on the scheduling component of the system.

Our modular design defines a minimal *micro-kernel*. More complex operations are implemented in a high-level system language (Concurrent Haskell)

outside this kernel. As a result, this design provides an *executable specification* and all code presented in this paper has been tested in the context of a modified runtime-environment (RTE) of the Glasgow Haskell Compiler (GHC) Version 6.6.

Immediate benefits of this design are the ease of prototyping and of replacing key components of the RTE — issues of particular importance in complex parallel systems such as *computational grids* [6], incorporating thousands of machines on a global scale. Supporting such global architectures, as well as emerging multi-core machines, requires support for deep memory and process hierarchies, which use different implementations, depending on the underlying architecture or other system features. Additionally the system needs to be *adaptive* in the sense that it dynamically adapts its behaviour to dynamically changing characteristics of the parallel machine.

In this sense, ArTCoP provides a generic and adaptive system for parallel computation, combining features of our existing parallel RTEs for GpH [19] and Eden [2, 3]. We present a *prototype implementation* of key concepts in such a system in the form of an executable specification, amenable to formal reasoning. We arrive at a system with a clear modular design, separating basic components by their functionality and employing a hierarchy with increasing levels of abstraction. The micro-kernel of this system is accessed via a narrow interface, and most of the coordination of the system is realised in a functional language. We demonstrate the flexibility of the system by refining a simple scheduler and adding sophisticated work distribution policies.

2 Related Work

Work in the 80s on high-level languages for system-level programming mainly focused on how to implement O/S concepts in a functional [8, 18, 14] or logic [17] style. Most of these systems introduce specific primitives to deal with non-determinism, whereas later approaches either insisted on maintaining deterministic behaviour [9] or used special data structures to control interactions between concurrent threads (such as MVars in Concurrent Haskell [15]). Early implementations of functional operating systems are NEBULA [11] and KAOS [20]. More recent functional systems are Famke [21] and Hello [4].

An early system that uses a micro-kernel (or substrate) approach in the RTE, is the Scheme-based Sting [10] system. Sting defines a coordination layer on top of Scheme, which is used as computation language. Genericity is demonstrated by directly controlling concurrency and processor abstractions, via Scheme-level policy managers, responsible for scheduling, migration etc. This general framework supports a wide range of features, such as (first-order) light-weight threads, thread pre-emption, and asynchronous garbage collection. Common paradigms for synchronisation (e.g. master-slave parallelism, barrier communication etc) are implemented at system level and demonstrate the possibility to easily define application-optimised synchronisation patterns. However, since Sting uses Scheme as a system level language, it lacks the clear separation of pure and im-

pure constructs at system level as offered by Haskell. We also consider the static type safety for system level code, provided by Haskell, an advantage.

Most closely related to our high-level language approach to O/S design is [7]. It defines a Haskell interface to low-level operations and uses a hardware monad to express stateful computations. It focuses on safety of system routines, using its own assertion language and Haskell's strong type system. This interface has been used to code entire O/S kernels (House, Osker) directly in Haskell, reporting satisfying performance. In contrast to this proof-of-concept approach, we want to improve maintainability by realising the more complex RTE routines in Haskell, but still keeping a micro-kernel implemented in a low-level language.

Another related project, the Manticore [5] system, targets parallelism at multiple levels, and enables the programmer to combine task and data parallelism. Manticore's computation language is a subset of ML, a strict functional language. The compiler and runtime system add NESL-like support for parallel arrays and tuples, and a number of scheduling primitives. Similar in spirit to our approach, only a small kernel is implemented in low-level C; other features are implemented in external modules, in an intermediate ML-like language of the compiler. A prototype implementation is planned for the end of 2007, and aims to be a testbed for future Manticore implementations and language design. As opposed to ArTCoP's genericity in coordination support, Manticore explicitly restricts itself to shared-memory multi-core architectures, and does not support networked computing, nor location-awareness and monitoring features.

The Famke system [21] is implemented in Clean and explores the suitability of Clean language features such as dynamic types and uniqueness typing for O/S implementation. Using these features type-safe mobile processes and concurrency are implemented. The latter uses a first class continuation approach and implements scheduling at system level.

Most recently Peng Li et al [13] have presented a micro-kernel (substrate) based design for the concurrent RTE of GHC, including support for software transactional memory (STM). This complements our work, which focuses on control of parallelism, and we intend to combine the design of our interface with that currently produced for GHC.

3 Design Aims of a Generic Runtime-environment

3.1 Simplest Kernel

ArTCoP aims to provide support for parallel programming from the conceptual, language designer perspective. A major goal of its design is to explore how many of the coordination tasks can be specified at higher levels of abstraction, and to identify the minimal and most general runtime support for parallel coordination. Therefore, major parts of the RTE are implemented in a high-level language. Following a functional paradigm has the advantage that specifications can more or less be executed directly and that it facilitates theoretical reasoning such as correctness proofs.

3.2 Genericity

Our study concentrates on identifying and structuring the general requirements of parallel coordination, with the only assumption that concurrent threads are executing a functionally specified computation, explicitly or implicitly coordinated by functional-style coordination abstractions.

The genericity we aim at is two-fold: By providing only very simple actions as primitive operations, our system, by design, is not tied to particular languages. We avoid language-specific functionality whenever possible, thus ArTCoP supports a whole spectrum of coordination languages. Secondly, the coordination system can be used in combination with different computation engines, and is not restricted to a particular virtual machine. Furthermore, this coordination makes minimal assumptions on the communication between processing elements (PEs). ArTCoP thus concentrates *key aspects of parallelism* in one place, without being tied to a certain parallelism model.

3.3 Multi-level System Architecture

High-level parallel programming manifests a critical trade-off: providing operational control of the execution while abstracting over error-prone details. In our system, we separate these different concerns into different levels of a multi-level system architecture. As shown in Figure 1, ArTCoP follows the concept of a *micro-kernel*, proven useful in the domain of operating system design.

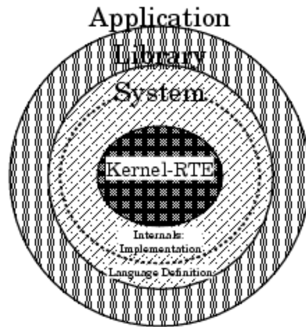

Fig. 1. Layer view of ArTCoP

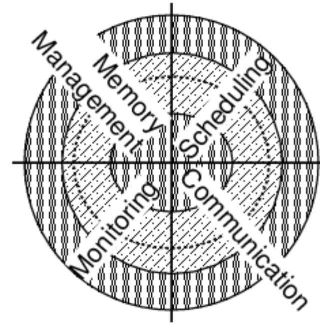

Fig. 2. Component view of ArTCoP

At *Kernel level*, the most generic support for parallelism is implemented. The system offers explicit asynchronous data transfer between nodes, means to start and stop computations, as well as ways to retrieve machine information at runtime. Operations at this level are very simple and general. *System Modules* build on the kernel to restrict and combine the basic actions to higher-level constructs, i.e. the constructs of a proper parallel functional language. The runtime support is necessarily narrowed to a special model at this level. The implemented parallel

coordination language is nothing else but the interface of the system level modules. At *Library level* and *Application level*, concrete algorithms, or higher-order functions for common parallel algorithmic patterns (called skeletons [16]) can be encoded using the implemented language.

Focusing more on functionality and modularity, the kernel can be divided vertically into four interacting components, as shown in Figure 2: Parallel subtasks are created and sent to other processing elements (PEs) for parallel execution by the *scheduling* component, which controls the local executing units. Explicit *communication* between several scheduler instances on different PEs is needed to coordinate and monitor the parallel execution. The *memory management* component is responsible for (de-)allocating dynamic data and distributing it over the available machines, interacting in this task with the communication component. Explicit message passing is possible, but not mandatory for data communication, and it is possible to implement a shared address space instead. In order to decide which PE is idle and suitable for a parallel job, static and dynamic system information is provided by a *monitoring component*.

3.4 High Level Scheduler Control

The key issue in efficiently using a wide-area network infrastructure for parallel computations is to control the parallel subtasks that contribute to the overall program, and to *schedule* the most suitable task for execution, depending on the current machine load and connectivity (whereas efficiently combining them is an algorithmic issue). Likewise, modern multicore CPUs will often expose uneven memory access times and synchronisation overhead. Parallel processes must be placed with minimal data dependencies, optimised for least synchronisation, and dynamically consider system load and connectivity. ArTCoP aims to be a common framework for different coordination concepts. Adaptive scheduling support will thus be specified in the *high-level language* and not in the runtime system.

4 Implementation of ArTCoP

4.1 Configurable Scheduling Support

We propose a parallel RTE which allows system programmers and language designers to define appropriate scheduling control at the system level in Haskell. In our parallel system the scheduler is a monadic Haskell function using an internal scheduler state, and *monitors* all computations on one machine. Subtasks are activated and controlled by a separate manager thread, which can take into account properties of the subtask and static and dynamic machine properties. The scheduler thread runs concurrently to the controlled computations and relies on a low-level round-robin scheduler inside the RTE. To specify it, we use the state monad and features of Concurrent Haskell, combining stateful and I/O-actions by a monad transformer [12]. We briefly summarise main features and notation in Fig. 3.

Monads and Concurrency in Haskell Monads, in one sentence, are Haskell's way to hide side-effects of a computation. If a computation is not referentially transparent, e.g. depends on externals (`IO`) or a system `State`, it can be mathematically described as a *monadic* evaluation. Likewise for side-effecting constructs, those which modify an *external* "state".
The `IO` monad in Haskell implements user interaction, and also encapsulates the nondeterministism of Concurrent Haskell: `fork`ing and `kill`ing threads, `yield`ing (to the scheduler), and synchronised communication via `MVar`s. The monad `State` encapsulates and provides controlled and ordered access to an arbitrary state As a (contrived) example, we define some functions which modify a simple counter, or run stateful counting actions.

```
data Counter = Counter Int Int    -- data type Int x Int (and constructor)

-- modifiers, stateful action on Counter
inc,dec,reset :: State Counter ()
-- modify the state by a given function (lambda-notation)
inc = modify (\(Counter n accesses) -> Counter (n+1)(accesses+1))
dec = modify (\(Counter n accesses) -> Counter (n-1)(accesses+1))
reset = do (Counter _ accesses) <- get  -- read the state
           put (Counter 0 (accesses+1)) -- set sth. as the new state
```

Do-notation, as shown in `reset`, is an intuitive notation for composing monadic actions, and for binding new names to returned values for subsequent use.

Modern Haskell implementations come with a rich set of hierarchically organised libraries, which provide these general monad operations, and specifics to certain monads, e.g. for the `State` monad, to elegantly program and run complex stateful computations. Exemplified here: `evalState` runs a stateful computation, `sequence_` sequences several monadic actions (all return the void type `()`).

```
countTo :: Int -> Counter -- run stateful computation on start state,
countTo m = evalState     --   and return final state
              (sequence_ (replicate m inc ++ [reset])) -- actions
              (Counter 0 0)                            -- start state
```

Monad transformers [12] can be used to combine two monads, in our case the `IO` and the `State` monad. `IO` actions are embedded into the combined monad by `liftIO`.

Fig. 3. Summary: Monads and Concurrency in Haskell

Parallel tasks in a coordination language implemented by ArTCoP will appear as a new type of job at library level. Haskell's type system allows to specify the respective scheduler for a certain kind of parallelism by overloading instances of a *type class* named `ScheduleJob`. The internal scheduler state type depends on the concrete job type and forms another type class which provides a start state and a termination check. A third type class `ScheduleMsg` relates Jobs and State to messages between the active units and provides a message processing function. Table 1 summarises the overloaded functions in the scheduler classes. A trivial default scheduler `schedule` is provided (shown in Fig. 4), which only starts the main computation, repeatedly checks for termination, and returns the final scheduler state upon termination.

Thus, language designers do not deal with runtime system code, but simply define the scheduling for such jobs at the system level. As a simple example, every

Table 1. Overview of class funtions (implemented at system level)

`type StateIO s a = StateT s IO a`	type alias combining State and IO monad
`class ScheduleState st where`	
`startSt :: st`	the initial state of the scheduler
`killAllThreads :: StateIO st ()`	shutdown function
`checkTermination :: StateIO st Bool`	check state, return whether to stop
`checkHaveWork :: StateIO st Bool`	check state, return whether any local work available
`class ScheduleJob job st | job -> st where`	
`runJobs :: [job] -> IO st`	run jobs with default start state
`schedule :: [job] -> StateIO st st`	schedule jobs, return final state
`forkJob :: job -> StateIO st ()`	fork one job, modify state accordingly
`class ScheduleMsg st msg | st -> msg where`	
`processMsgs:: [msg] -> StateIO st Bool`	process a set of message for the scheduler, modify state accordingly. Return `True` immediately if a global stop is requested.

```
runJobs jobs = evalStateT (schedule jobs) startSt
schedule (job:jobs) = do forkJob job
                         schedule jobs
schedule [] = do liftIO kYield               -- pass control
                 term <- checkTermination     -- check state
                 if term then get      -- return final state
                         else schedule ([]::[job]) -- repeat
```

Fig. 4. Default scheduler

machine could control a subset of the jobs, running one instance of the scheduler. To model this behaviour, only a few simple operations need to be hard-wired into the kernel. The basic kernel support can be grouped into scheduler control, communication, and system information. All primitive operations provided by the kernel (indicated by the leading `k`), and their types, are shown in Table 2. For the example, the Kernel has to provide the number of available PEs (`kNoPe`), and must support spawning asynchronous jobs on other PEs (`kRFork`), namely a scheduler instance which runs the jobs assigned to the local PE.

4.2 Explicit Communication

If additional jobs are created dynamically, they may be transmitted to a suitable PE, and received and activated by its scheduling loop. The scheduler instances may also exchange *requests* for additional work and receive jobs as their answers. This model requires communication between the scheduler instances. The kernel supplies an infrastructure for explicit message passing between any two running threads. It relies on typed *stream channels*, created from Haskell by `kOpenPort`, and managed by the kernel internally. A `kOpenPort` returns a placeholder for the stream, and a Haskell port representation to be used by senders for `kSend`.

Table 2. Overview of primitive operations (provided by the kernel)

Functionality at Kernel Level (primitive operations)	

`kRFork :: PE -> IO() -> IO()`
`kFork  :: IO() -> IO ThreadId`
`kYield :: IO()`
	start a remote computation
	start a local thread (Conc. Haskell)
	pass control to other threads (Conc. Haskell)

`kOpenPort:: IO( ChanName' [a],[a])`
 	
`kSend:: ChanName' [a] -> a -> IO()`
	open a stream inport at receiver side, return port handle and placeholder
	basic communication primitive, send an element of type a to a receiver (a port handle)

`kThisPe,kNoPe :: IO Int`
`kThreadInfo :: ThreadId -> IO ThreadState`
`kPEInfo :: Int -> IO InfoVector`
	get own node's ID / no. of nodes
	get current thread state (`Runnable, Blocked, Terminated`)
	info about a node in the system (cpu speed, latency, load, location etc)

Sending data by `kSend` does not imply any evaluation; data has to be explicitly evaluated to the desired degree prior to sending.

Stream communication between all scheduler instances, and startup synchronisation, are easy to build on this infrastructure. The scheduler may also receive messages from the locally running threads (e.g. to generate new jobs), which can be sent via the same stream. Language designers define suitable message types, accompanied by an instance declaration which provides the message processing function in the class `ScheduleMsg`.

```
instance ScheduleJob MyJob MySchedulerState where
   schedule (j:js) = do forkJob j
                        mapM_ addToPool js
                        schedule ([]::[MyJob])
   schedule empty = do stop <- do { ms <- receiveMsgs ; processMsgs ms }
                       term <- checkTermination
                       if (term || stop)
                         then do { killAllThreads; get }
                         else do work <- checkHaveWork
                                 if (not work)
                                   then sendRequest
                                   else liftIO kYield
                                 schedule empty
```

Fig. 5. Scheduler for a parallel job-pool

Figure 5 sketches a scheduler for such a language, assuming the existence of a globally managed job pool. If an instance runs out of work, it will send a request. It will eventually receive an answer, and the next call to `processMsgs` will activate the contained job. This example enables reasoning about appropriate workload

```
data PEInfo = PE { runQ_length :: Int, noOfSparks  :: Int ,   -- system
                   clusterId  :: Int , clusterPower:: Double,
                   cpuSpeed :: Int   , cpuLoad :: Double,     -- kernel
                   latency  :: Double, pe_ip   :: Int32,
                   timestamp:: ClockTime }
startup :: StateIO s ()
startup = do infos <- buildInfos --  startup, returns initial [PEInfo]
             let ratios = zipWith (\lat str -> fromIntegral str / lat)
                                 (map latency infos) (map cpuSpeed infos)
                 myVote = fromJust (findIndex (== maximum ratios) ratios)
             votes <- allGather myVote
             setMainPE (1 + hasMostVotes votes)
```

Fig. 6. System level code related to load information

distribution and the consequences and side conditions, while the scheduling loop itself remains small and concise. All essential functionality is moved from the scheduling loop into separate functions, e.g. we leave completely unspecified how jobs are generated and managed in the job pool, and how a scheduler instance decides that it needs work (in `checkHaveWork`). All these aspects can be defined in helper functions, allowing a clear, structured view on the scheduling implemented.

4.3 Monitoring Information

Programmable scheduling support at system level requires knowledge about static and dynamic system properties at runtime. Our system kernel is geared towards adaptive techniques developed for GridGum 2, GpH on computational Grids [1], and provides the necessary information. For location awareness, we have `kNoPe` for the total number of PEs in the parallel system, and `kThisPe` for the own PE. Another primitive, `peInfo :: PE -> IO InfoVector` returns a vector of data about the current system state of one PE. This information is continuously collected by the kernel and held in local tables *PEStatic* and *PEDynamic*.

Load information at system level: A list of load information represented in a Haskell data structure `PEInfo` is a self-suggesting component of the scheduler state in many cases. The concrete selection, postprocessing and representation of system information (provided by the kernel) depends on how the scheduler at system level wants to use the information. An example of a Haskell type `PEInfo` is shown in Fig. 6. It includes selected components of the scheduler state: the number of threads controlled by the local scheduler, and how many sparks (potential parallel computations) it holds.

As exemplified in the figure, the scheduler can do arbitrary computations on `PEInfo` structures. For instance, to start the computation on a "strong" machine with good connectivity, all PEs could *elect* the main PE by a strength/latency ratio. Each PE votes for a relatively strong neighbour, where neighbourhood is

a function of latency, varying for different electing PEs. A collective (synchronising) message-passing operation `allGather` is easily expressed using explicit communication. Referential transparency guarantees that all PEs will then compute the same value without further synchronisation.

5 Hierarchical Task Management and Adaptive Load Distribution

5.1 Hierarchical Task Management

We now embed the scheduler of the GUM RTE [19], which implements the GpH parallel extension of Haskell, into the generic framework presented in the previous section. In short, GUM provides two concepts going beyond the design of the simple scheduler in the previous section:

- *hierarchical task management*, distinguishing between potential parallelism ("sparks") and realised parallelism ("threads"); the former can be handled cheaply and is the main representation for distributing load; the latter, representing computation, is more heavy-weight and fixed to a processor;
- *adaptive load distribution*, which uses information on latency and load of remote machines when deciding how to distribute work;

We will see that, in this high-level formulation of the scheduler, the code modifications necessary to realise these two features are fairly simple. Hereafter, we first describe how to model the hierarchical task management in GUM. These changes only affect the scheduling component. In tuning load distribution, we then interact with the monitoring and communication components.

First we specify the machine state in the GUM RTE, consisting of: *a)* a *thread pool* of all threads; these are active threads controlled by the scheduler, each with its own stack, registers etc; *b)* a *spark pool* of all potential parallel tasks; these are modeled as pointers into the heap; *c) monitoring information* about load on other PEs; this information is kept, as a partial picture, in tables on each processor;
We model this data structure as a triple:

```
data GumState = GSt Threadpool Sparkpool [PEInfo]
type Sparkpool  = [GumJob]
type Threadpool  = [ThreadId]
```

and we make `GumState` an instance of `ScheduleState`.

The code for the GUM scheduler is summarised in Figure 7. The arguments to `schedule` are jobs to be executed. These jobs are forked using a kernel routine, and added to the thread pool (`forkJob`). The case of an empty argument list describes how the scheduler controls the machine's workload. First the scheduler checks for termination (1). Then the scheduler checks the thread pool for runnable tasks, otherwise it tries to activate a local spark (2). If local work has

```
instance ScheduleJob GumJob GumState where
 runJobs jobs = evalStateT (initLoad >> (schedule jobs)) startSt
 forkJob (GJ job) =   do tid <- liftIO (kFork job)
                         modify (addThread tid)
 schedule (j:js) = do { forkJob j ; schedule js }
 schedule empty  = do
   (runThrs, blThrs) <- updateThreadPool      -- update and
   term <- checkTermination                  -- (1) check local state
   if term
      then do { bcast GSTOP ; get } -- finished
      else do localWork <- if runThrs > 0    -- (2) local work available?
                              then return True   -- yes: runnable thread
                              else activateSpark -- no: look for spark
              stop <- if localWork
                          then do reqs <- readMs
                                  processMsgs reqs
                          else do sendFish   -- (3) get remote work
                                  waitWorkAsync
              if stop then do { killAllThreads; get } -- finished
                      else do liftIO kYield  -- (4) run some threads
                              schedule empty
-- essential helper functions:
activateSpark :: StateIO GumState Bool -- tries to find local work
sendFish :: StateIO GumState ()        -- sends request for remote work
waitWorkAsync :: StateIO GumState Bool -- blocks on receiving messages

updateThreadPool :: StateIO GumState (Int,Int)
updateThreadPool =  do
    (GSt threads sps lds) <- get
    tStates <- liftIO (mapM kThreadInfo threads)
    let list = filter (not . isFinished . snd) (zip threads tStates )
        blocked = length (filter (isBlocked . snd) list)
        runnable = length (filter (isRunnable . snd) list)
    put (GSt (map fst list) sps lds)
    return (runnable, blocked)
```

Fig. 7. GUM scheduler

been found, it will only read and process messages. The handlers for these messages are called from `processMsgs`, which belongs to the communication module. If no local work has been found, a special FISH message is sent to search for remote work (3). Finally, it yields execution to the micro-kernel, which will execute the next thread (4) unless a stop message has been received, in which case the system will be shut down. The thread pool is modeled as a list of jobs, and `updateThreadPool` retrieves the numbers of runnable and blocked jobs.

The above mechanism will work well on closely connected systems but, as measurements show, it does not scale well on Grid architectures. To address

shortcomings of the above mechanism on wide-area networks, we make modifications to the thread management component for better load balancing, following concepts of the adaptive scheduling mechanism for computational Grids [1]. The key concept in these changes is *adaptive load distribution*: the behaviour of the system should adjust to both the static configuration of the system (taking into account CPU speed etc.) and to dynamic aspects of the execution, such as the load of the individual processors. One of the main advantages of our high-level language approach to system-level programming is the ease with which such changes can be made. The functions of looking for remote work (`sendFish` and its counterpart in `processMsgs`) and picking the next spark (`activateSpark`) are the main functions we want to manipulate in tuning scheduling and load balancing for wide-area networks. Note that by using index-free iterators (such as `filter`) we avoid dangers of buffer-overflow. Furthermore, the clear separation of stateful and purely functional code makes it easier to apply equational reasoning.

5.2 Adaptive Load Distribution Mechanisms

The adaptive load distribution deals with: *startup*, *work locating*, and *work request handling*, and the key new policies for adaptive load distribution are that work is only sought from relatively heavily loaded PEs, and preferably from local cluster resources. Additionally, when a request for work is received from *another cluster*, the receiver may add more than one job if the sending PE is in a "stronger" cluster. The necessary static and dynamic information is either provided by the kernel, or added and computed at system level, and propagated by attaching load information to every message between PEs (as explained in Section 4.3).

Placement of the main computation During startup synchronisation, a suitable PE for the main computation is selected, as already exemplified in Section 4.3. GridGum 2 starts the computation in the 'biggest' cluster, i.e. the cluster with the largest sum of CPU speeds over all PEs in the cluster, a policy which is equally easy to implement.

Work Location Mechanism The Haskell code in Figure 8 shows how the target PE for a FISH message is chosen adaptively by `choosePE`. A ratio between CPU speed and load (defined as `mkR`) is computed for all PEs in the system. Ratios are checked against the local ratio `myRatio`, preferring nearby PEs (with low latency, sorted first), to finally target a nearby PE which recently exposed higher load than the sender. This policy avoids single hot spots in the system, and decreases the amount of communication through high-latency communication, which improves overall performance.

Work Request Handling Mechanism To minimise high-latency communications between different clusters, the work request handling mechanism tries to send multiple sparks in a SCHEDULE message, if the work request has originated from a cluster with higher relative power (see Figure 9). The relative power of

```
data GumMsg = FISH [PEInfo] Int -- steal work, share PEInfo on the way
            | SCHEDULE [PEInfo] GumJob -- give away work (+ share PEInfo)
            | GSTOP
            | ... other (system) messages...
sendFish:: StateIO GumState ()
sendFish = do infos <- currentPEs -- refresh PE information
              me <- liftIO kThisPe
              pe <- choosePe me
              liftIO (kSend pe ( FISH infos me ))

 -- good neighbours for work stealing: low latency, highly loaded
choosePe :: Int -> StateIO GumState (ChanName' [GumMsg])
choosePe me = do
   (GSt _ _ lds ) <- get
   let mkR pe = (fromIntegral (cpuSpeed pe)) / (cpuLoad pe)
       rList  = [ ((i,mkR pe), latency pe) -- compute 'ratio'
                  | (i,pe) <- zip [1..] lds ]   -- keep latency and PE
       cands  = filter ((< myRatio) . snd) -- check for high load
                 (map fst                  -- low latencies first
                  (sortBy (\a b -> compare (snd a) (snd b)) rList))
       myRatio    = (snd . fst) (rList!!(me-1))
   if null cands  then return (port 1)     -- default: main PE
                  else return (port ((fst . head) cands))
```

Fig. 8. GridGum 2 Work location algorithm

a cluster is the sum of the speed-load ratios over all cluster elements. If the originating cluster is weaker or equally strong, the FISH message is served as usual. In Figure 9, after updating the dynamic information (1), the sender cluster is compared to the receiver cluster (2), and a bigger `amount` of sparks is retrieved and sent if appropriate (3). In this case the RTE temporarily switches from passive to active load distribution.

6 Conclusions

We have presented the scheduling component in ArTCoP, a hierarchical runtime-environment (RTE) for parallel extensions of Haskell, which has been implemented on top of GHC Version 6.6. Using a *micro-kernel* approach, most features of the RTE, such as scheduling, are implemented in Haskell, which enables rapid prototyping of easily replaceable modules. Thus we can support both *deep memory and deep process hierarchies*. The latter is discussed in detail by presenting a scheduler for light-weight tasks. The former is ongoing work in the form of defining a virtual shared memory abstraction. Considering the daunting complexity of global networks with intelligent, automatic resource management, modular support for such deep hierarchies will gain increasing importance. In particular, we are interested in covering the whole range of parallel architectures,

```
instance ScheduleMsg GumState GumMsg where
   processMsgs ((FISH infos origin):rest) = do processFish infos origin
                                               processMsgs rest
   processMsgs ((SCHEDULE ...)     :rest) = ...

processFish :: [PEInfo] -> Int -> StateIO GumState ()
processFish infos orig = do
    updatePEInfo infos           -- update local dynamic information (1)
    me <- liftIO kThisPe
    if (orig == me) then return () --  my own fish: scheduler will retry
      else do
        new_infos <- currentPEs    -- compare own and sender cluster (2)
        let info   = new_infos!!(orig-1)
            myInfo = new_infos!!(me-1)
            amount = if (clusterPower info > clusterPower myInfo)
                       then noOfSparks myInfo `div` 2  -- stronger: many
                       else 1                   -- weak or the same: one
        sparks <- getSparks amount True       -- get a set of sparks (3)
        case sparks of
         []   -> do target <- choosePe me     -- no sparks: forward FISH
                    liftIO (kSend target (FISH new_infos orig))
         some -> liftIO (sequence_ -- send sequence of SCHEDULE messages
                   (map ((kSend (port orig)).(SCHEDULE new_infos)) some))
```

Fig. 9. GridGum 2 work request handling algorithm

from multi-core, shared-memory systems to heterogeneous, wide-area networks such as Grid architectures.

As one general result, we can positively assess the suitability of this class of languages for system level programming. Realising computation patterns as index-free iterator functions avoids the danger of buffer-overflows, and the absence of pointers eliminates a frequent source of errors. In summary, the language features that have proven to be most useful are: higher-order functions, type classes and stateful computation free of side effects (using monads).

Our prototype implementation realises all code segments shown in the paper, using the GHC RTE as micro-kernel, and Concurrent Haskell as a system-level programming language. This prototype demonstrates the feasibility of our micro-kernel approach. The different variants of the scheduler, specialised to several parallel Haskell implementations, show the flexibility of our approach.

While we cannot present realistic performance figures of this implementation yet, we are encouraged by related work reporting satisfying performance for O/S modules purely written in Haskell [7] and by recent performance results from a micro-kernel-structured RTE for Concurrent Haskell [13]. We plan to combine our (parallel) system with this new development by the maintainers of GHC and to further extend the features of the parallel system.

References

1. A.D. Al Zain, P.W. Trinder, H-W. Loidl, and G. Michaelson. Managing Heterogeneity in a Grid Parallel Haskell. *Scalable Computing: Practice and Experience*, 7(3):9–26, 2006.
2. J. Berthold. Towards a Generalised Runtime Environment for Parallel Haskells. In *Computational Science (ICCS'04)*, LNCS 3038, page 297ff, Krakow, 2004. Springer.
3. J. Berthold and R. Loogen. Parallel coordination made explicit in a functional setting. In *Implementation of Functional Languages (IFL 2006)*, LNCS 4449, pages 73–90, Budapest, Hungary, 2007. Springer.
4. E. Biagioni and G. Fu. The Hello Operating System. Information at http://www2.ics.hawaii.edu/~esb/prof/proj/hello/.
5. M. Fluet, N. Ford, M. Rainey, J. Reppy, A. Shaw, and Y. Xiao. Status Report: The Manticore Project. In *Proceedings of the ACM SIGPLAN Workshop on ML*, pages 15–24, Freiburg, Germany, October 2007.
6. I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the Grid: Enabling scalable virtual organizations. *International Journal of High Performance Computing Applications*, 15(3):200–222, 2001.
7. T. Hallgren, M.P. Jones, R. Leslie, and A.P. Tolmach. A Principled Approach to Operating System Construction in Haskell. In *Conf. on Functional Programming (ICFP'05)*, pages 116–128, Tallinn, Estonia, September 26–28, 2005. ACM.
8. P. Henderson. Purely Functional Operating Systems. In *Functional Programming and its Applications*. Cambridge University Press, 1982.
9. I. Holyer, N. Davies, and C. Dornan. The Brisk Project: Concurrent and Distributed Functional Systems. In *Glasgow Workshop on Functional Programming*, Electronic Workshops in Computing, Ullapool, Scotland, July 1995. Springer.
10. S. Jagannathan and J. Philbin. A Customizable Substrate for Concurrent Languages. In *Conf. on Programming Language Design and Implementation (PLDI'92)*, pages 55–67. ACM Press, July 1992. ACM SIGPLAN Notices 27(7).
11. K. Karlsson. Nebula, a Functional Operating System. Tech.Rep., Chalmers, 1981.
12. D.J. King and P. Wadler. Combining Monads. Tech.Report, Glasgow Univ, 1993.
13. Peng Li, A. Tolmach, S. Marlow, and S. Peyton Jones. Lightweight Concurrency Primitives for GHC. In *Haskell Workshop*, Freiburg, Germany, Sept. 2007.
14. N. Perry. Towards a Functional Operating System. Technical report, Dept. of Computing, Imperial College, London, UK, 1988.
15. S.L. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *POPL'96 — Symp on Principles of Programming Languages*. ACM Press, January 1996.
16. F. A. Rabhi and S. Gorlatch, editors. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer, 2002.
17. E. Shapiro. Systems Programming in Concurrent Prolog. In *Symp. on Principles of Programming Languages (POPL'84)*, Salt Lake City, Utah, 1984.
18. W.R. Stoye. A New Scheme for Writing Functional Operating Systems. Technical Report 56, Computer Lab, Cambridge University, 1984.
19. P.W. Trinder, K. Hammond, J.S. Mattson, A.S. Partridge, and S.L. Peyton Jones. GUM: A Portable Parallel Implementation of Haskell. In *Conf. on Programming Language Design and Implementation (PLDI96)*, pages 79–88, Philadelphia, 1996.
20. D. Turner. Functional Programming and Communicating Processes. In *PARLE II*, LNCS 259, pages 54–74, Eindhoven, The Netherlands, 1987. Springer.
21. A. van Weelden and R. Plasmeijer. Towards a Strongly Typed Functional Operating System. In *Implementation of Functional Languages (IFL02)*, LNCS 2670, pages 45–72, September 2002.