Abstract

This report documents the process of extending the GUM run-time system (RTS) for Glasgow parallel Haskell (GpH). First, we obtain the sources, set up the environment and build the Glasgow Haskell Compiler (GHC) and GUM RTS. Next, GpH and Evaluation Strategies are briefly introduced, followed by a short discussion of parallelisation, profiling and optimisation. Subsequently, we illustrate how to make small incremental changes to the RTS, touch on debugging and conclude by presenting some open issues and future work directions. Basic knowledge of Linux tools, C and Haskell is assumed.

1 Setting Up the Development Environment

To benefit from running GpH [38] programs using the distributed GUM [37] RTS [5] instead of the standard GHC RTS, we need a distributed target platform such as a cluster of multi-cores or a heterogeneous system such as a node comprising at least one host CPU (e.g. Intel Xeon) and at least one co-processor (e.g. Xeon Phi) which communicate using message passing (for instance using Virtual TCP/IP over PCIe). Adding and exploiting GPUs and accelerator cores to this configuration remains for future work.

We assume a standard Linux environment (in our case CentOS 6.7; use $\texttt{uname -a}$ and $\texttt{cat /etc/redhat-release}$ to check). Below we discuss some common command line utilities that should be available by default and several libraries that might need installation, notably PVM [32] or MPI [10] on which the messaging sub-system is built.

1.1 Software

We benefit from using Linux and its comprehensive set of tools for productive software development. Here we briefly mention just a few – refer to manuals and online documentation for details, as well as to StackOverflow and Google for quick non-obvious solutions for various issues including correct use, configuration and bug work-arounds.

The autotools suite (including configure, autoscanner, automake, etc., depends on libtool) facilitates code distribution, pkg-config (although non-POSIX) helps to find correct compiler and linker flags, recent gcc is the standard C compiler and works well together with gdb, the invaluable debugger. We also need an editor with syntax highlighting for C and Haskell and with ‘jump-to-definition’ functionality across files (e.g. Emacs or Vim). Version control is a must so we use git [20]. The valgrind memory profiler (including cachegrind), and gprof help profile C programs, whereas in addition to GHC profiling capabilities (e.g. hp2ps for heap profiles) ThreadScope [18] proved particularly useful in visualising the execution of parallel Haskell programs. The htop utility is recommended for monitoring the load on the machines, whereas egrep proved useful for searching the sources. Shell scripts, perl or python help automate repetitive tasks such as output processing, whilst R facilitates statistical analysis and plotting.

---

*HW-MACS-TR-0112, Heriot-Watt University, School of Mathematical and Computer Sciences, 15.12.2015*
1.1.1 Parallel Virtual Machine (PVM)

PVM is a message passing library mainly defined by a single implementation\(^1\) that creates a Parallel Virtual Machine and supports portable parallel programming on potentially heterogeneous networks of workstations. PVM uses a set of daemons distributed across PEs and the fork/exec spawn model for program execution, i.e. the same program is copied to and spawned at every PE and ids are used for conditional execution of different parts of the code in Single-Program-Multiple-Data (SPMD) fashion [31]. Unix-based operating systems are supported alongside Windows. Linux is assumed below.

After downloading the source code, the library can be installed using the common 
\$ configure; make; make install. Don’t forget to set the relevant environment variables, in particular PVM\_ROOT and PVM\_ARCH. PVM is started by using \$ pvm command and providing an optional hostfile that contains hostnames of nodes to be used (one per line)\(^2\). Table 1 summarises the basic PVM shell commands which can also be used in scripts (e.g. echo "halt" \| pvm). Note that we should halt the PVM after a crash during the measurements to clean up the temporary files and restart it before attempting to run the next set of measurements. Further commands are available and information on them can be obtained by using the help command or from the manual [32].

<table>
<thead>
<tr>
<th>command</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>help</td>
<td>displays overview of the commands</td>
</tr>
<tr>
<td>help &lt;cmd&gt;</td>
<td>shows usage info for the cmd command</td>
</tr>
<tr>
<td>conf</td>
<td>provides a table of hostnames in the system</td>
</tr>
<tr>
<td>add</td>
<td>adds a hostname to the configuration</td>
</tr>
<tr>
<td>del</td>
<td>deletes a node from the configuration</td>
</tr>
<tr>
<td>quit</td>
<td>exits the shell without stopping PVM</td>
</tr>
<tr>
<td>halt</td>
<td>stops the PVM system</td>
</tr>
</tbody>
</table>

The PVM library defines a message-passing API which you should be familiar with if you intend to change the packing format or implement new messaging primitives. The basic communication functions are pvm\_send() and pvm\_recv(), whereas several functions are available to obtain some information about the current configuration (such as pvm\_mytid(), pvm\_parent(), pvm\_config(), pvm\_tasks()). The pvm\_spawn() function is used to spawn tasks. For more information on PVM refer to the manual [32].

Consult MPSys\_tem.h, Parallel\_Rts\_h, PVMComm\_c, Datacomms\_c, HLComms\_c, among other modules in the rts/parallel/ folder, for implementation of multiple layers of communication within the RTS including sending and processing GUM’s protocol messages such as FISH and SCHEDULE. Note that communication and synchronisation in GpH are implicitly managed by the RTS, i.e. completely hidden from the application programmer.

When the application linked with the RTS that supports distributed reduction using PVM (-parpvm) is executed, the executable is copied to $PVM\_ROOT/bin/$PVM\_ARCH where also the .gr and .gs profiling files are placed, whilst output files are placed in the $HOME directory. The only exception is the output for the main PE which is placed in the directory where the executable was invoked. PVM is mature and relatively stable (last release in 2009 at the time of writing); several limitations are discussed in Section 1.1.3.

\(^1\) refer to http://www.csm.ornl.gov/pvm/pvm\_home.html for source code and documentation
\(^2\) we assume that passwordless login using ssh was already set up for all the nodes
1.1.2 Message Passing Interface (MPI)

MPI [14, 32] is a popular message-passing standard that facilitates distributed parallel programming using the SPMD programming model and defines over 300 functions including non-blocking and one-sided communication primitives and some higher-level collective operations, although only six functions are essential\(^3\).

Once the library is installed and $\$PATH$ is set appropriately, mpd daemons can be started on the available nodes\(^4\) whilst utilities such as mpdtest and mpadltest can be used to test their operation. The application needs to be compiled with mpicc which links the executable with the library. Finally, the application is executed using mpirun which takes as additional arguments the number of processes to create ($-n$) and an optional hostfile that specifies the nodes and the number of processes per node to be used.

1.1.3 MPI vs PVM

PVM is less portable and less broadly used than MPI, as the library functionality is chiefly defined by a single implementation, whilst it has more freedom for heterogeneous networks of workstations due to dynamic single machine view for the user (nodes can be added at run time). Packing functions help handle mixed-mode operation, however, parallel I/O and one-sided communication are supported only through experimental extensions. Moreover, programs using different PVMs cannot interoperate as nodes of different VMs may have identical ids, but static groups were recently added and may help to some extent. There is no support for different logical topologies as provided by MPI. The development is not very active as MPI has gained more widespread use. Debugging is difficult as adding printf() calls is still a state-of-the-art methodology (it is also possible to attach gdb instances to running processes) and tool support is lacking.

MPI was designed as a portable and efficient communications library for writing parallel applications\(^5\). The design does not mandate memory copies so that message passing on the node can be implemented as copying a pointer but does not require an implementation to be thread-safe, although the design allows for thread safety (e.g. there is no notion of 'current' buffer). MPI offers higher modularity since processes don’t have absolute ids but use ranks within opaque communication contexts. Additionally, topologies help organise logical communication links, whereas derived data types help support heterogeneity and transmission of non-contiguous data. As many vendors provide their own implementations and since the standard leaves many details unspecified, different libraries are likely to encounter interoperability issues. Mixed-mode operation (e.g. 32/64bit) is possible but is rarely implemented. MPI supports one-sided communication, which is particularly efficient on some architectures, as well as parallel I/O.

Although portability (i.e. the ability to compile and run code across diverse architectures) is a design goal for both libraries they do not provide performance portability. Additionally, there is no support for fault tolerance apart from handling some communication errors in general (there exist experimental fault-tolerant MPI implementations). Either library can be used as part of a hybrid programming model, for instance in addition to OpenMP for shared-memory programming on a node and OpenCL for using available GPUs. Unfortunately, deadlocks and race conditions can be introduced when using these libraries. Ultimately, MPI and PVM were developed with different goals in mind [12, 13], but both provide sufficient functionality to implement the point-to-point communication layer that would fulfil the requirements of the GUM RTS.

---

\(^3\)prominent implementations of the MPI standard include, for instance, MPICH (http://www.mpich.org/) and OpenMPI (http://www.open-mpi.org/)

\(^4\)more recent versions of MPI don’t require this explicit step

\(^5\)historically these were mainly parallel high-performance scientific applications
1.2 Building GHC and GUM

Having set up the environment, we can now proceed to build GHC and GUM from the obtained sources (available on request; release upcoming). Currently, the sources use an older version of GHC (6.12.3) and the aim is to get in line with the latest GHC release.

1.2.1 Build Sequence

The following sequence builds the GHC compiler and the GUM RTS in-place (i.e. in the source directory; specify `--prefix=<path>` that could be used by `$ make install`).

1. Use `$ git clone <url>` to obtain the source\(^6\). Then issue the `$ ./boot` command to boot the libraries (currently there is an issue with `autom4te`, which needs an extra manual step during the installation, see below).

2. Now call the familiar `$ ./configure` and specify the path to the GHC compiler (version 6.2.13) using the `$ --with-ghc=<path-to-ghc>`\(^7\) argument to the configure tool. Ensure that at least either PVM or MPI has been detected (you might need to set the environment variables as discussed above). There is an issue with the build process that requires to unpack some additional configuration files using `$ tar -xfz confs.tgz` (available with the distribution or upon request).

3. Create and edit `mk/build.mk` to specify library optimisation level and needed RTS functionality such as the used communication library (refer to 1.2.2 for more details).

4. Use `$ nohup make 1>make.log 2>&1 &` to start the build process (add `-j N` to use `N` threads), where `nohup` prevents session termination on logout if you have logged in using `ssh` and both `stdout` and `stderr` are redirected to the log file. We can use `$tail -f make.log` to continuously check the log file to monitor build progress.

5. If you have not specified the install directory explicitly and have no root access to write to default directories like `/usr/local/bin`, you can just use the in-place installation in `<PATH-TO-SRCS>/inplace/bin/ghc-stage1` (or `stage2`). Otherwise use `$ make install` to copy the executables, libraries, and include files into common (but distribution-specific) directories.

6. Currently, there is an issue with the auto-generated `rts/AutoApply.cmm`, which should be manually edited to ensure that the hash symbols for preprocessor macros are at the beginning of the line without any preceding characters, in particular whitespaces. The file `AutoApply-HACKED.cmm` illustrates the changes and should be available within the distribution (or upon request).

Now you should have a working GHC and RTS with capabilities specified in the `mk/build.mk` file in place, which can be used to compile GpH programs. The RTS can be modified to include additional functionality such as profiling, monitoring, or cost-model-based policy control, among other extensions. Some examples are discussed in Section 3.

1.2.2 Specifying Build Options in the `build.mk` File

Build options must be specified in the `mk/build.mk` file by copying, renaming (`build.mk`), and editing the provided `build.mk.sample`\(^8\), which defines the ways the RTS is built with, the communications library used, and the used functionality (e.g. support for threaded RTS and/or for virtual shared memory that enables execution across multiple nodes).

\(^6\)at DSG available from `/home/hwloidl/INPLACE/ghc-6.12-eden-gum-sm-fresh` (last access: 15.12.2015)

\(^7\)used to bootstrap, e.g. `/home/hwloidl/OPT/x86_64-unknown-linux/bin/ghc-6.12.3`

\(^8\)or by re-using a `build.mk` from an already installed GUM instance
The example file contains different build flavours, such as perf, quickest, quick, and prof. The flavours are distinguished based on the degree of applied optimisations. High level of optimisation increases the compilation time so that building using the quick setup is recommended for development, whereas a production system should use the perf build for best performance. Most flags are explained in accompanying comments. In particular, the GhcRTSWays specifies the flavours of the RTS that should be supported: v (vanilla), thr (threaded), thr_e (threaded with event logging), pp (GUM using pvm), pm (GUM using MPI) etc. Multiple ways can be combined, e.g. pp_debug for distributed RTS based on PVM with debugging turned on. Additionally, some way-specific flags can be specified: WAY_pp_HC_OPTS += -optc-DPAR_TICKY to enable per-thread (TSO) profiling or -optc-DDEBUG to enable debugging for debug ways (for an example refer to Appendix A).

2 A Whirlwind Tour of Glasgow Parallel Haskell

This section briefly introduces Glasgow parallel Haskell\(^9\) (GpH), a dialect of Haskell that supports a semi-explicit parallel programming model, in which most of parallelism management is hidden from the programmer, and can be used to create benchmark applications that can be employed to evaluate the effectiveness of the RTS-level policy and mechanism extensions. Alternative parallel programming models available within the Haskell ecosystem, e.g. Eden, Par monad, Accelerate, are reviewed elsewhere \[6, 23\].

2.1 Haskell Extension for Semi-Explicit Parallelism

GpH \[35, 38\] extends Haskell \[17, 26\], a state-of-the-art non-strict purely functional language which has been used in programming language research for over two decades and builds on top of the industrial-strength optimising Glorious Glasgow Haskell Compilation System (GHC), by introducing two combinators to specify evaluation degree and order: 
\[
\text{par} : : a \to b \to b \quad \text{and} \quad \text{pseq} : : a \to b \to b
\]
for parallel and sequential composition, respectively. The par combinator can be used by the programmer to express that the first argument could be beneficially evaluated in parallel, whereas the RTS is free to ignore the hint\(^{10}\). In turn, pseq is used to specify evaluation order by evaluating the first argument to weak head normal form (WHNF, i.e. applying the outermost constructor or function as opposed to fully evaluating data to normal form where no further reduction is possible) before evaluating the second argument\(^{11}\). As thunks represent expressions yet to be evaluated once they are demanded and futures represent computations under evaluation which one may wait for, we can view par as implementing lazy futures. This programming model is deterministic by design and avoids race conditions and deadlocks.

Several conditions must be satisfied for effective use of par \[23\]:

- the computation passed to par has to be unevaluated
- this computation must be computationally expensive to merit parallel execution despite the introduced parallelism management overhead
- its value is not required by the enclosing computation for a while
- the result is eventually demanded and shared with the rest of the program

If the last requirement is not met, the result may get garbage-collected before it can be used, whilst the first three points should be satisfied to achieve any speedup.

\(^9\)for more details see http://www.macs.hw.ac.uk/~dsg/gph
\(^{10}\)this is termed advisory (or optional) as opposed to mandatory parallelism (e.g. in Pthreads)
\(^{11}\)pseq is strict only in its first argument, whereas seq is strict in both; using pseq avoids compiler optimisations that would lead to an undesirable and unintended evaluation order (due to re-ordering)
{-# OPTIONS ~fglasgow~exts #~}~
~
module Main (main) where
import System (getArgs)
import Control.Parallel (par, pseq) -- or GHC.Conc(par,pseq)

main = do args <- getArgs -- checking args is not null omitted for brevity
  let n = read (args !! 0) -- size of the interval
      t = read (args !! 1) -- threshold for granularity control
      res = pfib n t
  putStrLn $ "pfib " ++ (show n) ++ " "
       ++ (show t) ++ " = "
       ++ (show res))

pfib :: Integer -> Integer -> Integer
pfib 0 _ = 1
pfib 1 _ = 1
pfib n t | n <= t = nfib n
          | otherwise = x `par` y `pseq` x + y + 1
          where x = pfib (n-1) t
                 y = pfib (n-2) t

nfib :: Integer -> Integer -- counts the number of function calls
nfib 0 = 1
nfib 1 = 1
nfib n = nfib (n-1) + nfib (n-2) + 1

Listing 1: GhP Code for a Divide-and-Conquer Version of parfib

Listing 1 shows an example implementation of parfib (adopted from [24]), which calculates the number of function calls in a Divide-and-Conquer (D&C) implementation that computes the nth Fibonacci number, with a threshold for granularity control, using par to identify potential parallelism and pseq to specify evaluation order.

We see how the changes to code are minimal, allowing incremental transformation into a parallel program that scales actual parallelism depending on the number of available PEs. The more PEs are available, the less of the advisory parallelism hints will be ignored by the RTS. However, for larger applications the direct use of par and pseq is discouraged as it is somewhat unstructured and mixes coordination and computation code. Note that maximum actual parallelism does not necessarily lead to minimal run time: for instance, if task granularity is too small, associated overhead defeats the benefits of parallelism.

2.2 Evaluation Strategies

Built on top of par and pseq, Evaluation Strategies [36] are non-strict, polymorphic, higher-order functions that increase the level of abstraction and provide means for structured parallel programming by separating computation and coordination concerns, facilitating understanding the algorithm without considering the coordination aspects.

The most recent version of Evaluation Strategies can be obtained from hackage and can be used by importing Control.Parallel and Control.Parallel.Strategies modules. A recent paper re-defined the original Strategies in terms of a strict identity Monad called Eval which also provides rpar and rseq, lifting par and pseq into Eval [24]. Strategies can be passed as parameters or combined using the dot, .| or .|| operators and used to define algorithmic skeletons [9] that are higher-order functions implementing common patterns of parallelism such as map, fold, scan and filter among others.

---

12There has been an API change from version 1.x to 2.x+ described in the package documentation; one issue was a space leak caused by GC failing to collect unused sparks as discussed in [25, Sec. 7]
13Haskell’s library repository; install the parallel package using the Cabal package management tool
data Eval a = Done a

type Strategy a = a 

runEval :: Eval a -> a
runEval (Done x) = x

instance Monad Eval where
  return = Done
  m >>= k = case m of
    Done x -> k x

r0, rseq, rpar :: Strategy a

r0 x = return x  -- no evaluation
rseq x = x `pseq` return x  -- evaluate to wnf
rpar x = x `par` return x  -- create a spark

rdeepseq :: NFData a => Strategy a  -- evaluate to normal form
rdeepseq x = x `deepseq` ( )  -- relies on Control.DeepSeq

using :: a -> Strategy a -> a  -- strategy application
e `using` strat = runEval ( strat e )
dot :: Strategy a -> Strategy a -> Strategy a  -- strategy composition
strat2 `dot` strat1 = strat2 . runEval . strat1

dot applies its strategy argument to all elements of a list
evalList :: Strategy a -> Strategy [a]
evalList strat [] = return []
evalList strat (x:xs) = do x' <- strat x
  xs' <- evalList strat xs
  return (x':xs')

dot applies a strategy to all elements of a list in parallel
parList :: Strategy a -> Strategy [a]
parList strat = evalList ( rpar `dot` strat )

-- a parallel map skeleton
parMap :: Strategy b -> (a -> b) -> [a] -> [b]
parMap strat f xs = map f xs `using` parList strat

Listing 2: Basic Eval Strategies [24]

Listing 2 shows the definition of the Eval monad as well as few basic Strategies. For instance, parList applies a strategy to each element of a list in parallel using the sequential evallist. In turn, parList can be used to implement the parMap skeleton, which applies a function to each element of the list in parallel.

2.3 Explicit Granularity Control

Maximum parallelism does not necessarily lead to highest performance due to coordination and parallelism management overheads. The GUM RTS implements an implicit thread subsumption mechanism allowing the RTS to prune superfluous parallelism by inlining child sparks into the parent to reduce thread creation overheads. However, often the available parallelism in functional programs is too fine-grained and high so that overheads of sparking and discarding sparks become significant and thread management overhead cancels out the benefits of parallelism and may even lead to overall slowdown. Hence, some explicit granularity control using annotations provided by the programmer proved necessary to achieve higher performance, as attempts at automatic granularity analysis and control have not yet yielded a general and effective solution.\footnote{one issue is estimating the granularity associated with a spark in the context of a non-strict language}
2.3.1 Chunking

Chunking is a granularity control technique commonly used in data-parallel applications and with task farm skeletons. The input is decomposed into coarse-grained chunks and then a strategy or a function is applied in parallel to each of the chunks instead of the single elements. Alternatively, a group of tasks is combined into a larger task and then sent to a worker. This is similar to aggregating a few small packets into a larger one to reduce communication overhead.

Chunking can be static where the decision is made only once at compile time or at execution time and can be based on input size, number of available processors, or on a cost model which can potentially incorporate more detailed architectural and system-level information (e.g. number of hierarchy levels, latency, load). Dynamic chunking happens at run time and allows to adaptively tune the chunk size during the execution, which is more flexible but often incurs some additional overhead.

Moreover, chunking can be advisory, where the annotation can be ignored by the system, or mandatory, where the chunking guarantees the splitting of work as specified. To a certain extent chunking can be viewed as nesting, since the original work units – elements – are turned into larger work units – chunks of elements – that aim to increase the computation-to-overhead ratio.

```
parListChunk :: Int -> Strategy a -> Strategy [a]
parListChunk n strat xs
| n <= 1 = parList strat xs
| otherwise = concat ('fmap' parList (evalList strat) (chunk n xs))
```

Listing 3: Chunking Strategy parListChunk

2.3.2 Thresholding

Thresholding is another explicit granularity control technique that is used with D&C algorithms and restricts the depth of the tree to a certain value, beyond which computation proceeds sequentially. As chunking, thresholding can be static or dynamic, and advisory or mandatory. Listing 4 presents a version of a generic D&C skeleton adopted from the 'Seq no More' paper [24].

The skeleton takes a function that sequentially solves a sub-problem once the threshold is reached, the actual input, a function that determines whether the threshold is reached, a function for combining sub-results, a function for sub-dividing the input, and returns a result. If the threshold has not been reached, the problem is recursively divided into smaller problems until these sub-problems can be solved and then the intermediate results combined to yield the final result. Note how fib is implemented by parameterising DnC.

```
DnC :: (a->b) -> a -> (a->Bool) -> (b->b->b) -> (a->[Maybe (a,a)])-> b
DnC solve arg threshold conquer divide = go arg
where go arg =
  case divide arg of
  Nothing -> solve arg
  Just (l0,r0) -> conquer l1 r1 'using' strat
  where l1 = go l0
  r1 = go r0
  strat x = do r l1; r r1; return x
    -- coordination
  where r | threshold arg = rseq
    | otherwise = rpar

fib n t = DnC n (fib n (<t) (+)) (\x->if x<2 then Nothing else Just (n-1,n-2))
```

Listing 4: Generic Divide-and-Conquer Skeleton
Granularity control techniques have been generalised as a more flexible clustering [21] technique and the Cluster type class with a proof obligation for the user to ensure that cluster . decluster = id [24, Sec. 5.2]. Additionally, more sophisticated library-level thresholding strategies have recently been investigated [34], in particular using a fuel-based approach. A detailed discussion of these topics is beyond the scope of this memo.

2.4 Compiling and Running GpH Programs

Interpreters such as ghci are useful for testing functions and for checking types of expressions, but ultimately the application should be compiled with optimisations turned on for best performance. GpH programs are compiled as Haskell programs using the modified GHC compiler with the $ ghc --make Main.hs -O2 for sequential execution.

Other often used flags include -fforce-recomp to force recompilation of used modules, -fglasmal-exts as a shortcut to include some useful extensions\textsuperscript{15} and -cpp to enable the use of the preprocessor for conditional compilation. Further flags include: -H<X> to set initial heap size, -K<X> to set the stack size, -A<X> to set the size for the allocation area\textsuperscript{16}, and -c to enable compacting GC\textsuperscript{17}. Table 2 provides a summary of several flags. For a complete overview of compiler flags refer to GHC’s manual whilst for GUM’s run time flags it is best to consult the source code directly (rts/RtsFlags.c).

2.4.1 Using the Threaded GHC-SMP RTS

To benefit from the multi-threaded GHC-SMP RTS a -thraeded compilation flag should be used, as well as -eventlog if you wish to generate trace files used by the ThreadScope profiler [18] and -rtsopts to enable extended RTS parameters to be passed following +RTS on the command line. At run time, use -H<X> to suggest the number of worker threads should be X, whereas -S and -s options enable more or less detailed summary output with a focus on garbage collection statistics (stdout is used for output unless a filename is specified). If the program was compiled with -eventlog the -1 RTS option can be used to create the .eventlog trace file that can be examined using ThreadScope.

2.4.2 Using the Distributed GUM RTS

To use the distributed GUM RTS with its implicit virtual shared memory abstraction, -parpvm or -parmpi must be used to select the library to implement low-level communication primitives as well as -rtsopts to enable parsing of options following the +RTS marker flag. Finally, -debug should be used when debugging to gain access to more detailed information and execution statistics.

At startup time the user selects the number of RTS instances using the -qp<X> flag\textsuperscript{18}. Depending on the RTS version many experimental flags are supported to enable and tune RTS parameters such a packet size, fishing policy, watermarks, load balancing policy, scheduling policy and to print out global statistics (-qPg and -s<filename>), event-based profiles (-qP, creates a .gr file per RTS instance), census-based profiles (-qPc, .gs files).

Make sure you restart PVM or MPI if a program run using GUM crashed to remove temporary files and locks. You can check return value of the application with $ echo $? (0 indicating success, non-zero values signifying failure)\textsuperscript{19}.

\textsuperscript{15}although generally its use is discouraged as it is a deprecated feature and extensions should be specified using the -X<extension-name> compiler flag separately for each extension

\textsuperscript{16}uses more cache but leads to less GC activity

\textsuperscript{17}may reduce GC activity if the ratio of live data to heap size is high (> 30%)

\textsuperscript{18}actually, it is a value used in a script and passed to the communication sub-system’s startup script

\textsuperscript{19}it is also useful to increase bash history file size and to search this history either using grep or Ctrl+R
Table 2: Commonly Used Compile and Run-Time Flags

<table>
<thead>
<tr>
<th>compile-time flag</th>
<th>effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>-O&lt;N&gt;</td>
<td>turn on optimisation of level N (0 = off; typically N = 2)</td>
</tr>
<tr>
<td>--make</td>
<td>automatically find the dependencies and build the executable</td>
</tr>
<tr>
<td>-fforce-recomp</td>
<td>forces recompilation of used modules</td>
</tr>
<tr>
<td>-gclay-exts</td>
<td>enables (many) language extensions (deprecated: use -XextName instead)</td>
</tr>
<tr>
<td>-cpp</td>
<td>enables the use of the C preprocessor for conditional compilation</td>
</tr>
<tr>
<td>-rtsopts</td>
<td>enables extended (but unsafe) RTS options</td>
</tr>
<tr>
<td>-threaded</td>
<td>links with the multi-threaded RTS (GHC-SMP)</td>
</tr>
<tr>
<td>-eventlog</td>
<td>enables event logging for ThreadScope (GHC-SMP)</td>
</tr>
<tr>
<td>-parvm or -parmpi</td>
<td>links with the distributed RTS (GUM) and PVM or MPI</td>
</tr>
<tr>
<td>-debug</td>
<td>enables verbosity flags and extra sanity checks for debugging</td>
</tr>
<tr>
<td>-prof -auto-all</td>
<td>enables sequential profiling (may require profiling versions of libraries)</td>
</tr>
<tr>
<td>-i&lt;path&gt;</td>
<td>specifies a non-standard path to a library</td>
</tr>
<tr>
<td>-v</td>
<td>verbose compiler output (e.g. to check locations searched for libraries)</td>
</tr>
<tr>
<td>-with-rtsopts&lt;optstr&gt;</td>
<td>sets specified RTS options to different default values</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>run-time flag</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-H&lt;size&gt;</td>
<td>suggested heap size (add e.g. M for Mbyte; not a limit)</td>
</tr>
<tr>
<td>-A&lt;size&gt;</td>
<td>suggested stack size</td>
</tr>
<tr>
<td>-K&lt;size&gt;</td>
<td>maximum stack size (limit)</td>
</tr>
<tr>
<td>-M&lt;size&gt;</td>
<td>maximum heap size</td>
</tr>
<tr>
<td>-p</td>
<td>generates a time profile</td>
</tr>
<tr>
<td>-H1 or -HC</td>
<td>creates a heap profile (.hp, convert with hp2ps and ps2pdf)</td>
</tr>
<tr>
<td>-G&lt;N&gt;</td>
<td>number of generations for GC (typically 2)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>threaded run-time flag</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-N&lt;X&gt;</td>
<td>use X Capabilities (worker threads)</td>
</tr>
<tr>
<td>-l&lt;flag&gt;</td>
<td>creates .eventlog trace file for ThreadScope</td>
</tr>
<tr>
<td>-S[&lt;filename&gt;]</td>
<td>detailed GC statistics (to stdout or to a file)</td>
</tr>
<tr>
<td>-s[&lt;filename&gt;]</td>
<td>GC summary statistics</td>
</tr>
<tr>
<td>-t</td>
<td>one-line GC summary statistics</td>
</tr>
<tr>
<td>-I&lt;N&gt;</td>
<td>idle GC delay in seconds (experimental)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>distributed run-time flag</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-qp&lt;N&gt;</td>
<td>specifies number of GUM instances (usually one per core)</td>
</tr>
<tr>
<td>-qPg</td>
<td>prints summary statistics to par_log.&lt;PeID&gt;</td>
</tr>
<tr>
<td>-qP</td>
<td>generates event-based profile (.gr files)</td>
</tr>
<tr>
<td>-qPc</td>
<td>generates census-based profile (.gs files)</td>
</tr>
<tr>
<td>-qz&lt;i&gt;</td>
<td>enables history table with update interval i (experimental)</td>
</tr>
<tr>
<td>-qy&lt;i&gt;</td>
<td>enables spark co-location with lookahead bound L (experimental)</td>
</tr>
<tr>
<td>-qw&lt;s&gt;</td>
<td>wait S seconds at the start of execution (useful for attaching gdb)</td>
</tr>
<tr>
<td>-qD&lt;level&gt;</td>
<td>debugging output verbosity level (power of two, see Table 4)</td>
</tr>
<tr>
<td>-qf&lt;N&gt; and -qqf&lt;N&gt;</td>
<td>to set FISH delay and FISH delay factor</td>
</tr>
<tr>
<td>-qL&lt;N&gt;</td>
<td>low watermark (start stealing if the number of local sparks is lower)</td>
</tr>
<tr>
<td>-qT&lt;N&gt;</td>
<td>maximum number of thunks in a packet</td>
</tr>
</tbody>
</table>

Another experimental version of the RTS (GUM-SMP) is currently being developed that combines GUM's distribution mechanism and SMP's multi-threaded execution and accepts most flags from all sets. For detailed description of the system refer to [1].
2.5 Profiling and Optimising Strategic GpH Programs

In this section we briefly discuss profiling and optimisation of GpH programs. We use a program that approximates π by using the Monte Carlo method [28]. We simulate random dart throws at a unit square and count how many darts hit the unit circle. We estimate \( \pi = 4 \times \frac{\text{#hits}}{\text{#samples}} \) and report program run time and an error term (the difference between the estimated and the true value). The run time increases with the number of samples as does accuracy (up to a certain limit), hence parallelising will potentially help to reduce run time whilst increasing accuracy.

First we discuss a naive sequential GpH implementation and compare it to a version written in C. Then we briefly discuss sequential profiling and optimisation. Next we parallelise the code using parallel map and compare the scalability and performance on a multi-core server and on a cluster of multi-cores using up to 64PEs for a multi-threaded (SMP) and a distributed RTS (GUM). Finally, keep in mind that premature optimisation is evil, so one should profile the program to find the bottlenecks, as often optimisation of a small fraction of the code results in largest improvement [19].

2.5.1 Sequential Profiling and Optimisation

Listing 5 shows the source code of the sequential version of the application (MCPI), whilst Table 3 summarises the run times and error characteristics compared to a version of the same program written in C. Both programs represent a naive initial implementation. The median run out of five is presented for the Haskell version run on a 4-core processor (some variation is due to few background processes), whereas the C version was run 100 times for smaller sample sizes (up to \( 10^{10} \)). The program takes as arguments the number of samples (problem size, run time increases with the number of samples, but we expect a more accurate answer) and a seed value for the pseudo-random number generator.

```
module Main (main) where
  -- source from https://github.com/jevelnikov/mcpi
import System.Environment (getArgs)
import System.Random (StdGen, mkStdGen, randomR)
import Text.Printf (printf)

main = do
  args <- getArgs
  if null args || (length args) < 2
    then error "usage: ./mcpi <nsamples> <seed>\n"
    else do
      let
        n = read (args!!0)
        seed = read (args!!1)
        res = mcPi n seed
      putStrLn (" seq MCpi : N = " ++ (show n) ++ " and SEED " ++ (show seed) ++ " = " ++ (show res))
      printf " error : %.6f\n" (res - pi)

mcPi :: Integer -> Int -> Double
mcPi n seed = go prng (0, n)
  where
    prng = mkStdGen seed
    go (h, 0) = 4.0 * (fromIntegral h) / (fromIntegral n)
    go (h, m) = go newGen (h, m-1) -- tail recursive
      where
        range = ((-1.0), 1.0) -- unit square / circle
        (x, newGen) = randomR range gen
        (y, newGen') = randomR range newGen
        h | (x*x + y*y) <= 1.0 = 1
        | otherwise = 0
```

Listing 5: Monte Carlo Estimation of π in GpH
Table 3: Sequential Run Times (sec) for Initial Haskell vs C MCPI Program

<table>
<thead>
<tr>
<th>number of samples</th>
<th>run times (Haskell)</th>
<th>error (Haskell)</th>
<th>run times (C)</th>
<th>error (C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^5$</td>
<td>0.232</td>
<td>-0.003153</td>
<td>0.001</td>
<td>0.002367</td>
</tr>
<tr>
<td>$10^6$</td>
<td>2.292</td>
<td>0.000435</td>
<td>0.022</td>
<td>-0.001353</td>
</tr>
<tr>
<td>$10^7$</td>
<td>22.858</td>
<td>-0.000113</td>
<td>0.191</td>
<td>-0.000673</td>
</tr>
<tr>
<td>$10^8$</td>
<td>236.249</td>
<td>-0.000018</td>
<td>1.864</td>
<td>0.000257</td>
</tr>
<tr>
<td>$10^9$</td>
<td>2341.559</td>
<td>0.000041</td>
<td>18.650</td>
<td>0.000334</td>
</tr>
<tr>
<td>$10^{10}$</td>
<td>23084.706</td>
<td>0.000023</td>
<td>186.000</td>
<td>-0.00020</td>
</tr>
<tr>
<td>$10^{11}$</td>
<td>timeout</td>
<td>NA</td>
<td>1860.000</td>
<td>0.000007</td>
</tr>
<tr>
<td>$10^{12}$</td>
<td>timeout</td>
<td>NA</td>
<td>18744.995</td>
<td>0.000000</td>
</tr>
</tbody>
</table>

We observe a strong linear correlation between the problem size and execution time: increasing sample number by an order of magnitude results in run time increasing by an order of magnitude. Hence, the benchmark is deemed computationally intensive and due to lack of dependencies between samples appears to merit parallelisation. Moreover, Haskell performance consistently lags behind C by two orders of magnitude (both are compiled with optimisations, -O2). However, this is rather an argument for naive C programs being close to optimised versions, whereas naive Haskell programs often result in less efficient programs. Note that the comparison is unfair as the C version (cf Appendix B) uses simple variables to store the results, whilst Haskell uses lists and boxed arbitrary length Integers that use the GNU gmp library resulting in additional overhead.

Another observation is that for the Haskell program the error term decreases faster but then at some point reaches an oscillating plateau, whereas C’s error term decreases slower but consistently every time the sample size is increased by the factor of 10.

Often naive Haskell programs result in creation of some unnecessary intermediate data structures and in redundant data-structure traversals. Next we examine time and memory profiles of the sequential Haskell program which helps identify performance bottlenecks and largest memory consumers. For a more detailed discussion refer to Real World Haskell book [29, Ch. 25] (available online). One could also introduce strictness annotations to avoid delayed computation and investigate the potential for using more efficient data structures than lists (e.g. ByteString, if lists are used) as well as unboxing the values.

Listing 6: Summary Statistics Output Using the -stderr Flag
First, we can collect run-time statistics such as heap residency, allocation rate, garbage collection time and overall run time, by passing the `+RTS -sstderr` arguments when running the program. Listing 6 shows high productivity, low heap residency (line 3), and low GC percentage (line 16), which suggests that the naive version (using an accumulator parameter) is relatively efficient and contains no memory leaks, but this doesn’t explain the large difference in performance compared to the C version. The allocation rate is high but within the expected range for functional programs that are often memory intensive due to graph reduction execution model.

Next, to gain a more detailed picture, we need to recompile the program for profiling using `-prof -auto-all -caf-all -fforce-recomp` compiler flags and assuming that a profiling versions of the used libraries are avaiable. Then we run the program with `+RTS -p` to obtain profiling report in `<program-name>.prof` file. Another run-time option can be used to increase stack size: e.g. `-K512M`. Profiling increases the run time, memory residency, and GC overhead (in our case: it doubles the run time, increases residency by ca. 30%, and GC% by 0.2%).

Listing 7: Time and Allocation Profile (Truncated)

The output is unsurprising: the program spends the whole time in a single function which generates two random numbers and depending on the numbers recursively calls itself with potentially increased hits counter until the specified number of samples has been reached. Although memory consumption doesn’t look suspicious we also look at the heap profile obtained by using the `-hc` run time option which generates the `<program-name>.hp` file. This file is not really human-readable, so we use $ hp2ps -e8in -c <program-name>.hp $ to create a graphical summary of heap usage over time in a .ps file (use ps2pdf for further conversion or ghostview -orientation=seascape -scale=2 <program-name>.ps to view). More flags are available such as `-hy` (break down by data type), `-hd` (by constructor allocation), `-hr` (retainers, i.e. system stack or thunks), `-i<N>` (sampling interval in seconds), among others (refer to GHC manual for more details).

The heap profile also looks rather unsuspicious as memory use is constant throughout.

---

20Use `cabal reinstall <lib-name> --with-library-profiling` or manually call `runghc Setup configure --user --prefix=<where to install> -p; runghc Setup build; runghc Setup install.`
the execution and requires a small amount of memory. A typical indication of a space leak would be a gradually increasing and large memory view with a drop in the end of the execution. A space leak is usually triggered by a build-up of intermediate data structures or thunks that are finally reduced in the end (a common example is the use of `foldl` to implement the `sum` function).

The following list contains few pointers to common optimisations:

- use tail recursion (to help compiler optimise)
- use more suitable and optimised data structures (e.g. ByteString, Vector, Text)
- reduce the amount of created intermediate data structures
- eliminate unnecessary data-structure traversals, use accumulating parameters
- use BangPatterns language extention to increase strictness
- use unboxed values\(^{21}\), inlining, and strict data fields (UNPACK pragma)
- try different compiler flag settings for optimisation
- try different RTS flag settings (e.g. increase heap to reduce GC pressure)

Care must be taken as some optimisations may reduce the available parallelism. Moreover, it is often useful to keep the inefficient but obviously correct program for comparison to ensure correctness of the optimised versions. As the focus of this memo is on extending the RTS, an in-depth discussion of more advanced sequential and parallel optimisations, including fusion, compile and run-time parameter auto-tuning, and inspection or the intermediate compiler outputs (\texttt{core}, \texttt{stg}, \texttt{cmm}, \texttt{asm}) is not covered in this memo.

\(^{21}\)ensure the values are not passed to functions that take boxed values or are non-strict as it requires packing
2.5.2 Parallelisation Methodology

To develop a parallel version of a given sequential code, which can serve for checking correctness and as the baseline for performance comparisons, we need to find opportunities for parallelism by recognising computations that can be beneficially performed in parallel. Common parallelism patterns include map, fold, filter, scan, compose, zip, among others. If there are no interdependencies, lists lend themselves for chunked parallelism, as do trees and graph data structures for clustering and divide-and-conquer parallelism.

There are several distinct advantages of the semi-explicit programming model used by GpH [15]. First, it allows for sequential debugging as the value semantics of the sequential and the parallel program are equivalent due to referential transparency (i.e. values can be substituted for equivalent expressions). Moreover, independent sub-expressions can be evaluated in any order, in particular in parallel. Second, the programming model is deterministic by design, thus avoiding deadlocks and race conditions. Third, the level of abstraction is high, as parallelism control is mostly hidden from the programmer, who is required only to identify potential parallelism, but the RTS ultimately decides whether an expression will be evaluated in parallel. This has an added benefit that the RTS can adapt the amount of actual parallelism to the number of the available cores. Finally, iterative parallelisation is easy as Evaluation Strategies were designed for separation of concerns, which means coordination can be changed without interfering with the algorithm [22, 24].

We build on these strengths when parallelising given sequential code by incrementally introducing parallelism using Strategies whilst ensuring correctness after every transformation. Profiling results help identify the most computationally intensive functions on which to focus the optimisation effort. Often after parallelism has been identified, granularity needs to be tuned to avoid excessive overhead (too many small tasks) and load imbalance (too few tasks overall). To further improve parallel performance, sequential fraction should be as small as possible (cf Amdahl’s Law [2, 16]). Then profiling can be used to compare different parallel versions and compiler and RTS flag settings to tune them for best performance. Usually, the obtained performance will not be optimal but in many cases it is possible to achieve near-linear speedups (at least as long as enough work is available). Tuning continues iteratively until satisfactory performance is obtained. Note that sometimes the given code was optimised too much for sequential execution to allow parallelisation and developing parallel code from scratch may prove more beneficial.

There are some problems with effective use of GpH. The key issue is the non-obvious operational cost model in the non-strict setting which requires the use of idioms mentioned in Section 2 and of profiling to confirm that parallelism is in fact being created and utilised as intended without creating memory leaks. High level of abstraction leads to a convenient architecture-independent programming model but makes obtaining performance across different architectures more difficult: it involves manual tuning of many parameters and making choices between different parallel implementations. In principle, this task could be automated (at least to some extent), which is a subject of ongoing research.

2.5.3 Parallel Profiling and Optimisation

Once a parallel version of the code is available it should be profiled using an increasing number of PEs to assess performance and scalability. As an overview it is useful to plot the run time and the speedup over the number of PEs. The expected speedup is sub-linear but should be as close to linear as possible. Profiling reveals opportunities for further optimisation. In particular, detailed event-based and census-based profiles will provide information about the amount of memory in use, the number of messages and on the actual and potential degree of parallelism. Finally, tools like ThreadScope can be used to visualise event logs and inspect program behaviour over the course of the execution.
We parallelise the MCPI program by generating multiple seeds (one per spark) so that each thread can use a separate random number generator. Additionally, a further command line argument specifies the number of sparks to generate (i.e. the degree of parallelism), where each spark is responsible for computing a fraction of the total number of samples. We slightly modify the core function to return the number of hits instead of $\pi$ itself to avoid the need to perform arithmetic operations on floating point numbers which may increase the overall error term [11]. Instead we first add the intermediate numbers of hits produced by different threads and then calculate the approximate value of $\pi$. Listing 8 summarises the required and fairly small code modifications.

```
 1−− . . .−− randoms function helps generate a list of seeds
 2import System.Random(StdGen,mkStdGen,randomR,randoms)
 3−− . . .
 4−− in main get an extra command line argument, chunk size and seeds
 5  nchunks = read (args!!1) :: Int−− inside the let clause
 6−− chunk size should divide nsamples without remainder for simplicity
 7  chunk_sz = nsamples ‘div’ (toInt nchunks)
 8−− . . .
 9  seeds = take nchunks $ randoms gen−− create a seed for each prng
 10−− use parMap at top level to create parallelism
 11  intermediates = parMap rdeepseq (mcPiI chunk_sz) seeds
 12−− calculate the final result at top level instead of inside mcPi
 13  res = 4.0 * (fromIntegral $ sum intermediates)
 14    / (fromIntegral nsamples) :: Double
 15−− . . .
 16−− new type of mcPiI (we add I to the name which stands for Integer)
 17mcPiI :: Integer -> Integer
 18−− . . .−− last change: return the number of hits instead of pi
 19  go _ (hits,0) = hits
 20−− . . .
```

Listing 8: Parallel Monte Carlo Estimation of $\pi$ using parMap (changes only)

**GHC-SMP** After compiling the program with the additional `-threaded`, `-rtsopts` and `-eventlog` flags, as described in Section 2.4.1, we can take advantage of using the multi-threaded SMP RTS by running the program with the following extra runtime flags: `+RTS -N4 -l -sstderr -S`, placed after the program’s command line arguments. Using Threadscope we can view the execution profile that was stored in the .eventlog file.

*Figure 2: A Threadscope Activity Profile (10^7 samples, seed = 23)*
In the main window we can observe the overall activity profile and profiles for each worker (HEC, Haskell Execution Context). Figure 2 shows good activity on 4 HECs, as green color marks mutation (computation) activity and orange stands for garbage collection (idle HECs would have white gaps). It is possible to zoom in and examine the traces in more detail as illustrated in Figure 3. For instance, we can see lower activity at the end of the execution where the main HEC is collecting the results. Another example of Threadscope’s functionalities and usage can be found in the literature [18].

Figure 3: Zoomed Onto the End of the Execution Trace

In the Traces tab on the left we can turn on additional details on sparks and see that all sparks are created at the beginning of the execution and are gradually turned into threads one by one by the worker Capabilities (HECs) and that all 100 sparks have similar sizes (use -lf RTS flag). Threadscope can be installed by using the $ cabal update; cabal install threadscope command and may require a set of other libraries. By default, the executable is placed into the $HOME/.cabal/bin directory.

Figure 4 summarises the execution times of running the parallel application using a threaded RTS on a multi-core server using up to 64 cores and the resulting relative speedups, comparing two different inputs (10e8 vs 10e9 samples). Notice how the run times on one core differ by an order of magnitude and drop significantly once more cores are added in both cases. The largest improvement occurs for smaller core counts as the parallelism overhead is lower relative to computation. The speedup curve even starts to drop for the highest core counts signifying that parallel overhead may be negating the parallelism benefits due to insufficient amount of work or load imbalance.

Figure 4: Threaded Version Run Times and Speedups (100 sparks)

We can observe a respectable speedup of close to 10 on 16 cores and of around 13 on 48 cores, however it is still far from linear. This leaves potential for optimisation and exhibits a scalability issue of either the application or the RTS (or signals lack of work).
At this point we could continue by examining the profiles in more detail and investigate the effect of increasing the number of sparks on performance. Some details on useful metrics can be found in a study of characteristics of parallel functional applications [7].

Moreover, we could try using the Divide-and-Conquer skeleton from Listing 4 or parBuffer instead of parMap, to ensure that sparks are gradually created and used throughout the execution. Alternatively, nested parallelism can prove useful, in particular on a target platform that has a hierarchical architecture. This is left as an exercise for the keen reader as we move on to briefly discuss additional profiling of the distributed RTS and then look at several ways to extend the RTS.

GHC-GUM Alternatively, we can compile with `-parpvm -rtsopts` for distributed execution using GUM with appropriate RTS flags as described in Section 2.4.2. As communication costs are higher in the distributed setup, data locality becomes more of a concern whilst load balancing remains important. One potential benefit of distributed execution is the reduced contention for caches and memory controllers. Moreover, less aggressive thread creation pays off if RTS keeps larger threads and prunes smaller ones.

Currently, GUM’s traces cannot be displayed using Threadscope and a set of custom scripts can be used to analyse the profiles\(^\text{22}\). Using the `-qP` flag to create .gr files for each PE requires RTS built with the `-DPAR_TICKY` compiler flag. After extending the profiling capabilities, these files also contain information about thread sizes (granularity: the RT value in the END event, emitted when a lightweight thread terminates) in textual form, similar to the example in Listing 9. The information can be plotted as histograms to gain insight into the granularity profile of the application as illustrated by Figure 5.

```
1 PE 2 [34]: FETCH 0x7f4b4b97e1c0 (from 1) [Capability 0]
2 PE 2 [37]: REPLY 0x7f4b4b97e1c0 (from 2) [Capability 0]
3 PE 2 [37]: SCHEDULE 0x0 (from 2) [Capability 0]
4 PE 2 [65]: END 1, SN 0, ST 1450052146484, EXP F, BB 0, HA 0, RT 25,
             BT 0 (0), FT 3 (1), LS N/A, GS 0, MY F
Listing 9: Per-PE Event-Based Profile (Excerpt)
```

Figure 5: Application Granularity Profile (on 8PEs of a cluster)

From the granularity profile we see that the 100 sparks are of roughly the same size and are relatively small compared to the total application run time. This hints at some improvement potential involving increasing thread granularity. For larger PE numbers more sparks may be needed. As this memo is focused on RTS extension, we leave out more detailed discussion of optimisations to be addressed in a separate memo.

\(^{22}\)e.g. the gr2pe and gs2pe tools are available upon request
3 Extending the GUM Run-Time System

This section provides several examples of extending the RTS. First, we demonstrate how to add a new RTS flag or option (command line argument). Second, we explore extending profiling capabilities of the RTS by adding some per-PE global state to gather summary messaging statistics and per-lightweight-thread granularity profiles based on events and how to write them to files. We also discuss the census-based approach to profiling and monitoring, which allows to use the profile information within the same run of the program to adapt to detected changes. Next we discuss how to change the default load balancing mechanism. Finally, we learn how to introduce a new primitive operation and thus to modestly extend the GpH language, which requires minimal changes to the compiler itself.

GUM’s policies and mechanisms, including virtual shared memory using global addresses, sparking to efficiently represent potential parallelism, and fishing for passive load distribution (random work stealing), are described in some detail in the original paper [37].

3.1 Adding a New RTS Flag or Option

RTS flags and options\textsuperscript{23} are the main way to configure the RTS by passing flags and parameter values to the program at startup time. Remember to compile the program with \texttt{-rtsopts} to enable extended options to be accepted between the \texttt{+RTS} and \texttt{-RTS} markers on the command line. Adding a new command line flag or option includes fairly straightforward and localised changes to the RTS making it a good first example:

- Declare your flag in \texttt{includes/rts/Flags.h}; presumably as part of \texttt{PAR\_FLAGS}.
- Initialise the flag to a default value in \texttt{rts/RtsFlags.c}.
- Add the code to recognise and parse the flag along with potential arguments in \texttt{rts/RtsFlags.c}; use \texttt{decodeSize()} to parse a numerical value; make sure the flag is not already used for another purpose\textsuperscript{24}; use the existing code as an example and add a new \texttt{case} for the new flag; note that there is no space between an option and its argument (also important when running the program).
- Add a comment to document the purpose and the usage of the flag or option
- Re-compile the RTS (\$ make clean \&\& make in the \texttt{rts} directory)
- Test that the implementation works as expected

Done! After checking that the flag is being properly initialised, recognised, and set, you can use the flag throughout the RTS (e.g. in \texttt{rts/Schedule.c}). For instance, assuming a boolean flag you could check whether it is set and perform some action only in this case as show in Listing 10.

```c
1  if (RtsFlags.ParFlags.myFlag == rtsTrue) {
2      // ... code block executed only is myFlag is set
3  }
```

Listing 10: Using a RTS Flag

If you find that using the extra conditional has undesirable effect on performance it is possible to use conditional compilation instead, downside of which is cluttering the code with \texttt{ifdefs} and requiring re-compilation of the RTS every time a parameter change is required\textsuperscript{25}. Avoid such conditionals inside tight innermost loops if possible.

\textsuperscript{23}in Unix parlance a flag (e.g. \texttt{-qPg}) takes no arguments but an option takes one (e.g. \texttt{-qz100})
\textsuperscript{24}the parsing code is quite long and messy due to GHC-specific requirements and depending on whether the RTS is sequential, threaded, or parallel, different flags are already in use (and there are quite a few of them)
\textsuperscript{25}another solution would be to add support for configuration files and use callback functions
3.2 Extending Profiling Capabilities

Currently, profiling is essential for manual optimisation of parallel applications since it helps determine bottlenecks and execution hot spots by recording statistical data about events that happen during the execution. For similar reasons automated approaches to performance optimisation can benefit from profiling or monitoring.

Here we will look into how profiling capabilities in GUM can be extended to provide summary statistics on message counts and event-based per-thread granularity information. The discussion mostly applies to census-based profiling and monitoring as well.

First, we familiarise ourselves with the part of the parallel RTS in the rts/parallel sub-directory that is responsible for parallel profiling. Using the $ find <dir> -name "<filename-pattern>" command we can search for different files. Additionally, we can apply slicing [33] to the code base to find file names and line numbers on which a specified search-string occurs by using the following command:

$ egrep -n -R -I --exclude-dir="." --exclude-dir=".." "<search-string>" *. To find a particular command you recently used in bash history (see $HOME/.history), press Ctrl+R and specify a command (press Ctrl+R again to look for a less recent match). Once the suitable code sections are located, we can proceed to modify the profiling sub-system and test the extended implementation. Version control [20] enables recovery if something goes wrong.

3.2.1 Enriching Cumulative Statistics with Detailed Message Counts

The profiling module is called ParTicky (see rts/parallel/ParTicky.c and .h), but the necessary changes are spread across multiple files. First we need to figure out which files require change by examining existing profiling code. A good starting point is to look for RTS flags that enable profiling. For instance, -qPg enables summary statistics, which we wish to enrich by adding information on the number of sent and received messages for different message types (e.g. FISH (work requests), ACK, RESUME and FETCH, SCHEDULE).

Due to lack of up-to-date documentation, the first steps may feel somewhat like reverse engineering but are made easier through the availability of the source code.

- Starting from the already familiar rts/RtsFlags.c, we find that the variable associated with the -qPg flag is RtsFlags.ParFlags.ParStats.Global, so we can search for its occurrences to find definition and usage sites in the code. We also notice that RtsFlags.ParFlags.ParStats.Suppress flag needs to be turned off and should be included in our conditional test.

- At usage sites we find the test whether the flag is set wrapped into a preprocessor constant PAR_TICKY for conditional compilation, as shown in Listing 11. It allows to re-compile the RTS to exclude this type of profiling from the code so that the overhead is not incurred if summary statistics are not required. Additionally, in the multi-threaded RTS a mutex has to be used to protect counter update. We can add similar code in other places. For instance, if we want to add some messaging-related code we would declare and initialise new counters and then use them to count messages by extending relevant parts of the rts/HLComms.c module. For example, an obvious place for counting sent FISH messages is inside the sendFish() function.

- Once we have added new counters (e.g. to the globalParStats structure) and appropriately update them, we can print out the formatted results, similar to other results printed in rts/parallel/ParTicky.c.

---

26 $n displays line numbers, -R enables recursive search in sub-directories, -I excludes binary files
27 $ history shows recent commands and their numbers, $ !<N> executes command with number N

20
Listing 11: Conditional Compilation Using C Pre-Processor

Once the set of changes is complete, re-compile the RTS ($make clean && make inside the rts sub-directory) and re-link the application with the modified RTS. Test the implementation by running some test programs with summary profiling turned on (-qPg) and examine the generated output.

Listing 12: Additional Messaging Statistics (Part of the Profiling Output)

Listing 12 presents the extended summary statistics found in the par_log.* files which also contain further cumulative statistics such as the number of sparks created and pruned as well as global address table residency for each PE. We can use a scripting language such as perl to parse the output files and extract the values we wish to analyse.

3.2.2 Per-Thread Granularity Profiles

As seen above, granularity profiles may provide additional insight into the computational structure of the application. Here we will look at how per-lightweight-thread granularity profiling can be added to event-based profiling (-qP). It is instructive to first browse the existing profiling code to understand how the related data structures are defined, initialised and used. Additionally, examining the output .gr files gives an idea of the output format and of necessary post-processing for data analysis and visualisation. Notably, we find the globalParStats structure, of type GlobalParStats defined in rts/parallel/ParallelRts.h, which could include additional information.

In GUM, each light-weight thread is implemented by a Thread State Object (TSO) that contains a pointer to the current Capability, a stack and some other book-keeping information such as the unique thread id and state (e.g. runnable or blocked). A TSO is allocated on the heap so it can be automatically garbage-collected once no longer needed after the thread terminates. We opt for avoiding to change the TSO itself as this would require non-trivial modifications to the compiler, the code generator as well as to the garbage collector, because TSO layout is crucial for efficient execution and garbage collection. Instead, we define a separate data structure to hold the information regarding execution, fetching and blocking times for each TSO (let’s call it TSOParInfo; we also add functions for allocating and disposing the structure).

We can use a hash table to map from a thread id to the corresponding info structure. The hash table implementation is provided in rts/Hash.c and the API is defined in rts/Hash.h, which needs to be included in the files that use the hash table. The hash table can be created at RTS startup time if profiling is on and remains empty until new threads are created and info records are added to the table for each thread (e.g. in createThread() in Thread.c). When threads run or block (see Schedule.c, e.g. case ThreadRunGHC), the cumulative timers and counters inside the info structure are updated accordingly to reflect the events (msTime() function is used to obtain current time).
Listing 13: Updating Granularity Info for a TSO

Example code in Listing 13 illustrates how the the granularity counter is updated for a given thread. When a thread terminates (see Schedule.c, e.g. case ThreadFinished), an event description is written to the event file (we can model our printing function on DumpRawGranEvent()), before the info structure is removed from the hash table and deallocated explicitly, whilst the TSO is automatically garbage-collected.

A similar approach applies to extending the census-based profiling (-qPc), which is performed at particularly disruptive points in the execution such as garbage collection to amortise the overhead and tends to sample the RTS state less frequently which may result in lower accuracy. On the other hand, the generated .gs files are usually smaller compared to the files generated using event-based profiling.

Monitoring refers to online profiling where the results are used at run time. This allows the RTS to react more flexibly to detected events. For instance, RTS can distinguish parallelism generators from workers based on the running average of the spark pool size and switch to active load distribution if deemed beneficial for some periods of time. Next we introduce an extension that uses historical information on the location of past stealing successes to tune the selection of spark donors, which will receive the work requests.

3.3 Extending Work Stealing

Work stealing is a passive load distribution mechanism that assumes no knowledge about the system as idle PEs (thieves) initiate the process and select their victims at random. This has the potential to scale and has been shown to perform well on tightly-coupled shared-memory multiprocessors for well-formed workloads [8].

However, the execution starts with a single PE generating the initial sparks and in some cases (e.g. when using parMap) most of the parallelism will be generated early during the execution by a small set of PEs. Since idle PEs would attempt to randomly steal work, they will generate many unsuccessful stealing requests (FISH messages). This situation can potentially be improved by sharing and using information about the locations of past stealing successes to choose victims less randomly, increasing the likelihood of choosing PEs that have useful work to donate.
First, we inspect the implementation of work stealing beginning with the main scheduler loop that runs on each PE (see Schedule.c). At some point, if no threads are runnable, a local spark will be picked up and turned into a thread if available. Failing that, a FISH message will be sent to a PE chosen using the choosePE() function defined in HLComms.c. We will return to this function later to bias the choice based on the available history about the location of past stealing successes. Below we briefly summarise the necessary changes to add this selection mechanism to the RTS.

- Introduce a new data structure to hold the ids for several PEs that recently have donated sparks (it could also hold further information such as spark granularity).
- Add a new RTS flag that turns on the new mechanism (even better: make it an option that takes as argument an interval that determines when the stored information can be considered out of date so it can be discarded from the history). Add the necessary initialisation and cleanup code to RtsStartup.c and ParInit.c.
- Update the code processing the incoming work request, i.e. FISH, messages (in the processFish() function in HLComms.c); if sparks are available, one is sent to the thief; if no work is available, the message is forwarded to another PE, unless it has expired, in which case it is returned to the original sender. Moreover, if an own expired FISH message arrives, a new work request is sent to some other PE.
- Similarly, modify the code that processes the successful response to the stealing attempts, i.e. a SCHEDULE message with at least one spark that can be converted into a new thread (see processSchedule()), to update the history information.
- In general, it will be necessary to extend the packet format to include the additional history information (see sendSchedule() in HLComms.c and sendOpNV() in LLComms.c). Make sure the offsets used for packing and unpacking are properly updated. The easiest way is to extend the header but conceptually the data may rather belong into the payload. Note that endianness is important as ultimately a packet in a buffer is represented by a sequence of bytes.
- Update the choosePE() function to use the available history information for a less random victim selection (if the RTS flag is set).
- Add code to periodically remove stale information from the history to avoid poor choices (e.g. when processing new incoming messages).
- Enhance census-based profiling to emit history coverage (how many of the stored values are not stale for how many PE ids out of total number of PEs). This can help evaluate the appropriate choice of the invalidation interval for a given application on a specific target platform.

This is a rather high-level overview of the changes and the difficulty is often in correctly implementing a mechanism in detail. History-based stealing appears beneficial in cases where parallelism generators are spatially and temporally stable [4]. Further studies are needed to evaluate invalidation interval selection on a broad range of architectures and using larger benchmark applications.

### 3.4 Adding a New par Primitive

Parallelism in GpH is exposed using par that is based on a primitive operation (PrimOp) par# from the GHC.Conc module\(^{28}\), whereas pseq specifies evaluation order. It is possible to add a different version of par to the language by adding support for suitable primitive

\(^{28}\)which can be found in libraries/base/GHC/Conc.lhs
operations to the RTS. One example is `parDist` that acts like `par` but allows the programmer to pass two additional arguments that specify how far in the network topology sparks are allowed to travel, which was shown to improve performance on multi-core clusters [3]. To add a new primitive, say `parInformed#`, which takes an extra argument that carries some information to be used by the RTS such as granularity of the expression as annotated by the programmer or by the compiler, we need to:

1. Add the PrimOp definition to `compiler/prelude/primops.txt.pp` including the PrimOp signature `primop ParInformedOp "ParInformed#" GenPrimOp` followed by the type, e.g. `a → b → InfoT → b`, followed with and few property specifiers like `has_side_effects = True` and `out_of_line = True`. As documentation is scarce, one can use the already implemented PrimOps as an example.

2. Create a RTS function that will be called by the PrimOp and perform the actual work (e.g. in `Spark.c`). In our example it could spark an argument based on the information provided. Let’s call the function `sparkInfoBased()`.

3. Update `includes/stg/MiscClosures.h` with `RTS_FUN(stg_parInformedzh);` where `zh` stands for `#` in the required `z-encoding` (see GHC Wiki for more information).

4. Add a new symbol for `ghci` to `rts/Linker.c` by extending the list of symbols with `SymI_hasProto(stg_parInformedzh)` at the end.

5. Implement the function in `rts/PrimOps.cmm` using the conventions and features of the Cmm language. For instance, first eight arguments are passed in registers named R1-R8 and can be accessed as illustrated in Listing 14.

6. Re-compile the compiler and the RTS (`$ make clean && make` in the `compiler` and `rts` sub-directories) along with the benchmark program.

7. Test the implementation and ideally add some test cases to your test suite.

```
1 stg_parInformedzh {
2   W_ x;
3   W_ y;
4   W_ info;
5   MAYBE_GC(R1_PTR, stg_parInformedzh);
6   MAYBE_GC(R2_PTR, stg_parInformedzh);
7   x = R1;
8   y = R2;
9   info = R3;
10  // calling the RTS function that
11  // will add a spark to the spark pool
12  // based on the info
13  foreign "C" sparkInfoBased(x "ptr", info "ptr");
14  RET_P(y);
15 }
```

Listing 14: Cmm Implementation of the New Primitive

Keep in mind that this discussion is based on using GHC 6.12.3 and related Cmm (a variant of the C-- portable assembly language [30]) which have meanwhile evolved further, so that particular syntax, conventions, and source file references may be different from what you will find with the most recent GHC version.

Here we only discuss the `out-of-line` PrimOps, which require minimal changes to the compiler, whilst potentially more efficient `inline` PrimOps require some changes to the code generator and are (partly) discussed on the GHC Wiki.
3.5 How I Learned to Stop Worrying and Love the Debugger

The extensions discussed in Sections 3.3 and 3.4 are more substantial, so that with high probability some implementation errors will be made and things will not work as expected after the first re-compilation. Debugging the RTS might turn out to be the most time-consuming part of extending the RTS and there are no shortcuts.

Do not panic, \texttt{gdb} (GNU debugger) to the rescue! Despite somewhat cumbersome usage, this invaluable tool will allow you to re-run the execution and stop at specified breakpoints and examine memory contents to verify your assumptions about the state of the RTS. Printf-debugging by printing values at several points during execution may be helpful in closing in onto a suspect code region but can not compete with proper debugging tools. Detailed information on how to use \texttt{gdb} can be found in a good tutorial-style book \cite{27}, which also covers \texttt{ddd}, a GUI for \texttt{gdb}, and debugging using Eclipse.

Embrace debugging distributed code as one of the most humbling and intellectually stimulating activities. Pathos aside, once you get some experience with debugging you might even find the puzzle-solving aspect somewhat entertaining. First, the RTS and the program need to be compiled with \texttt{-debug} (modify \texttt{build.mk}), which will enable the use of some additional flags that enable sanity checks (\texttt{-DS}) and debug output (\texttt{-dD<mask>}; see \texttt{RtsFlags.c} and Table 4 for a summary). Then attempt debugging execution on a single PE first (\texttt{-qp1}) as this avoids operational non-determinism. You can set breakpoints at particular places in your code and then \texttt{single-step} through a sequence of statements and then \texttt{continue} the execution. Moreover, you can use a \texttt{watchpoint} to stop execution as soon as some variable changes its value and print a \texttt{stack trace} of the calltree.

Table 4: RTS Debugging Output Options

<table>
<thead>
<tr>
<th>option</th>
<th>effect (print debugging output for a sub-component)</th>
</tr>
</thead>
<tbody>
<tr>
<td>-qD1 or -qDv</td>
<td>verbose output related to parallel RTS in general</td>
</tr>
<tr>
<td>-qD2 or -qDc</td>
<td>mpcomm; low level message handling</td>
</tr>
<tr>
<td>-qD4 or -qDp</td>
<td>pack; packing code</td>
</tr>
<tr>
<td>-qD8 or -qDq</td>
<td>packet; verbose packing</td>
</tr>
<tr>
<td>-qD16 or -qDP</td>
<td>processes; process management</td>
</tr>
<tr>
<td>-qD32 or -qDo</td>
<td>ports; port management code</td>
</tr>
<tr>
<td>-qD64 or -qDw</td>
<td>weight; weights and distributed GC</td>
</tr>
<tr>
<td>-qD128 or -qDF</td>
<td>fetching-related</td>
</tr>
<tr>
<td>-qD256 or -qDf</td>
<td>fishing-related</td>
</tr>
<tr>
<td>-qD512 or -qD1</td>
<td>tables; print internal address tables</td>
</tr>
<tr>
<td>-qD1024 or -qDd</td>
<td>unused (reserved for GdH)</td>
</tr>
<tr>
<td>-qD2048 or -qDz</td>
<td>paranoia; (creates huge output files)</td>
</tr>
</tbody>
</table>

Another useful RTS option for debugging on multiple PEs is \texttt{-qW<N>} that pauses the startup of the RTS for the specified number of seconds to allow \texttt{gdb} instances to be attached to each instance of the RTS given its process id. It is also possible to define macros for common sequences of commands.

If changing the compiler or the code generator or when performing low-level optimisation, it is useful to inspect intermediate outputs using the \texttt{-ddump-to-file}, \texttt{-ddump-stg}, \texttt{-ddump-simpl} compiler flags, among others (see GHC Wiki for more information). A good strategy is to make small changes and immediately test them instead of implementing a large set of modifications before re-compiling. Let the force be with you!

\footnote{For GHC-specific info see \url{https://ghc.haskell.org/trac/ghc/wiki/Debugging/CompiledCode}}
4 Discussion

This memo presents a brief account of and a tentative set of guidelines for extending the GUM RTS. Its aim is to contribute to documenting the process as well as the software artifact and to help new developers get up to speed more quickly. The material covers the build process, compilation and running of the applications, sequential and parallel profiling. The focus is on discussing several gradually more substantial examples of RTS extensions. Optimisation and debugging are briefly mentioned. There remain many open issues that should be addressed in the future.

4.1 Open Issues

The RTS code has been developed by many people over more than two decades and as part of different projects which makes the code-base difficult to navigate and maintain. Unfortunately, currently there is no logically central repository, which complicates sharing of new extensions and some effort is at times wasted by re-implementing features already available on a separate branch in a different repository. Additionally, detailed and up-to-date documentation is lacking and there are significant differences between GUM which uses an older version of GHC and most recent GHC-SMP. Additionally, GUM is still in a rather experimental and insufficiently stable state to warrant re-inclusion as part of GHC. Moreover, GUM’s trace files are incompatible with Threadscope which at present requires the use of custom visualisation tools.

One practical concern is attracting more users by releasing a publicly available and stable version, along with documentation to which this report can contribute. An update of the relevant web pages should follow. This goal could be helped, to some extent, by a release of an accessible in-depth tutorial that would use GpH and illustrate the use of Evaluation Strategies in more detail and on larger applications.

4.2 Future Work Directions

One of the key non-research future work directions is the merge of GUM (and GUM-SMP) to align it with most recent GHC. This will increase its visibility, the potential user base, make performance comparisons more meaningful and also further enrich the Haskell ecosystem for parallel and distributed programming. As GUM shares a large part of the RTS with Eden, which is in line with GHC HEAD, the work could be integrated and Eden used as an example when merging the different versions together. Ideally, GUM would reach a state of sufficient stability to merit re-inclusion into the GHC, which would also require at least one person responsible for maintaining the central repository.

From the research point of view there are many interesting avenues for future work, such as integration with other parallel programming models in Haskell for instance supporting GPUs as potential compute nodes, using architectural and system information within a cost model to adaptively control parallelism, and comparing RTS-level solution with a library-level one, among others.

Overall, as shared-memory architectures and associated programming models, such as threads, are bound to hit scalability limits, distributed-memory architectures and programming models appear promising in exploiting the full potential of future heterogeneous and hierarchical parallel architectures, whilst balancing programmer productivity and portable performance across a wide range of target platforms.
References


A  Example build.mk File

```makefile
BuildFlavour = quick
GhcLibWays = v
ifeq "$(BuildFlavour)" "quick"
SRC_HC_OPTS = -H64m -O0 -fasm
GhcStage1HcOpts = -O -fasm
GhcStage2HcOpts = -O0 -fasm
GhcLibHcOpts = -O -fasm
SplitObjs = NO
HADDOCK_DOCS = NO
BUILD_DOCBOOK_HTML = NO
BUILD_DOCBOOK_PS = NO
BUILD_DOCBOOK_PDF = NO
endif
GhcHcOpts = -v5
NoFibWays =
STRIP=: 
# RTS versions (NB: pp is for PVM, use pm for MPI)
# vanilla | threaded | threaded+eventlog | parallel | parallel+debug
GhcRTSWays = v thr thr_1 pp debug_pp
# way-specific flags (e.g. -DPAR_TICKY enables parallel profiling)
WAY_thr_HC_OPTS += -optc-DTRACING -optc-DPAR_TICKY
WAY_thr_1_HC_OPTS += -optc-DTRACING -optc-DPAR_TICKY
WAY_pp_HC_OPTS += -optc-DDUMMY_HL_COMMS -optc-DDUMMY_BLOCKED_FETCH \ 
-optc-DDUMMY_FREE -optc-DPAR_TICKY
WAY_debug_pp_HC_OPTS += -optc-DDUMMY_HL_COMMS -optc-DDUMMY_BLOCKED_FETCH \ 
-optc-DDUMMY_FREE -optc-DDUMMY_FREE -optc-DPAR_TICKY
```

Listing 15: A minimal build.mk file
B Naive Sequential C Implementation of MCPI

```c
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <sys/time.h> /* timeval; gettimeofday() */

double dartboard(long long ndarts);
double gtdiff_ms(struct timeval *tStart, struct timeval *tEnd);

int main(int argc, char *argv[]) {
  double ms;
  struct timeval tStart, tEnd;
  const short SEED = 23;
  long long MAX_TOSSES = 1000000000000;

  if (argc < 2) {
    fprintf(stderr, " usage: ./mc-pi-seq <NUM_SAMPLES>
");
    exit(EXIT_FAILURE);
  }

  MAX_TOSSES = atoll(argv[1]); // get number of samples
  srand48(SEED); // seed the PRNG
  gettimeofday(&tStart, NULL); // start timer
  pi = dartboard(MAX_TOSSES); // estimate pi using monte carlo
  gettimeofday(&tEnd, NULL); // stop timer
  ms = gtdiff_ms(&tStart, &tEnd); // execution time in ms

  printf("estimated pi (n= %lld) = %f
", MAX_TOSSES, pi);
  printf("error (vs math.h) = %f\n", pi - M_PI);
  printf("execution time:\n" , (long)(ms/1000), ((long)ms%1000));
  return EXIT_SUCCESS;
}

double dartboard(long long ndarts) {
  double x, y, pi;
  long long i, hits;
  hits = 0;

  for (i = 0; i < ndarts; ++i) { // throw ndarts darts
    x = (2.0 * ((double)random()) / RAND_MAX) - 1.0; // (x, y) are random
    y = (2.0 * ((double)random()) / RAND_MAX) - 1.0; // in range [-1,1]
    if ((x*x + y*y) <= 1.0) {
      ++hits; // count random darts that land inside the unit circle
    }
  }

  return 4.0 * (double)hits / (double)ndarts;
}

double gtdiff_ms(struct timeval *tStart, struct timeval *tEnd) {
  double elapsedTime; // assumes sane input

  elapsedTime = (tEnd->tv_usec - tStart->tv_usec) / 1000.0; // sec to ms
  elapsedTime += (tEnd->tv_usec - tStart->tv_usec) / 1000.0; // us to ms

  return elapsedTime;
}
```

Listing 16: Monte Carlo Estimation of π in C