GranSim User's Guide

Version 0.03

July 1996

Hans-Wolfgang Loidl
hwloidl@dcs.gla.ac.uk

This user's guide describes how to use GranSim for simulating the parallel execution of (annotated) Haskell programs. In passing we will discuss how to write parallel, lazy functional programs and how to tune their performance. To this end, some visualisation tools for generating activity and granularity profiles of the execution will be discussed. A set of example programs demonstrates the use of GranSim.

GranSim is part of the Glasgow Haskell Compiler (GHC), in fact it is a special setup of GHC, which uses a slightly modified compiler (for instrumenting the code) and an extended runtime-system. For users who are already familiar with the GHC and parallel functional programming in general there is a quick introduction to GranSim available (see Chapter 1 [Quick Intro], page 2).

1 A Quick Introduction to GranSim

If you already know how to compile a Haskell program with GHC and if you have an installed version of GranSim available there are only a few changes necessary to simulate parallel execution. Basically, a compile time flag has to be used to generate instrumented code. Runtime-system flags then control the behaviour of the simulation.

1. Compile all modules with the additional options `-gransim` and `-fvia-C`. Use `-gransim` also when linking object files. For example

   ```
   ghc -gransim -fvia-C -o foo foo.hs
   ```

   creates a GranSim executable file `foo`.

2. When running the program use the runtime-system option `-bP` to generate a full GranSim profile (see Section 8.1 [Types of GranSim Profiles], page 50). See Chapter 5 [Runtime-System Options], page 17 for a description of all options that allow you to control the behaviour of the simulated parallel architecture. For example

   ```
   ./foo +RTS -bP -bp16 -bl400
   ```

   starts a simulation for a machine with 16 processors and a latency of 400 machine cycles. It generates a GranSim profile '`foo.gr`' .

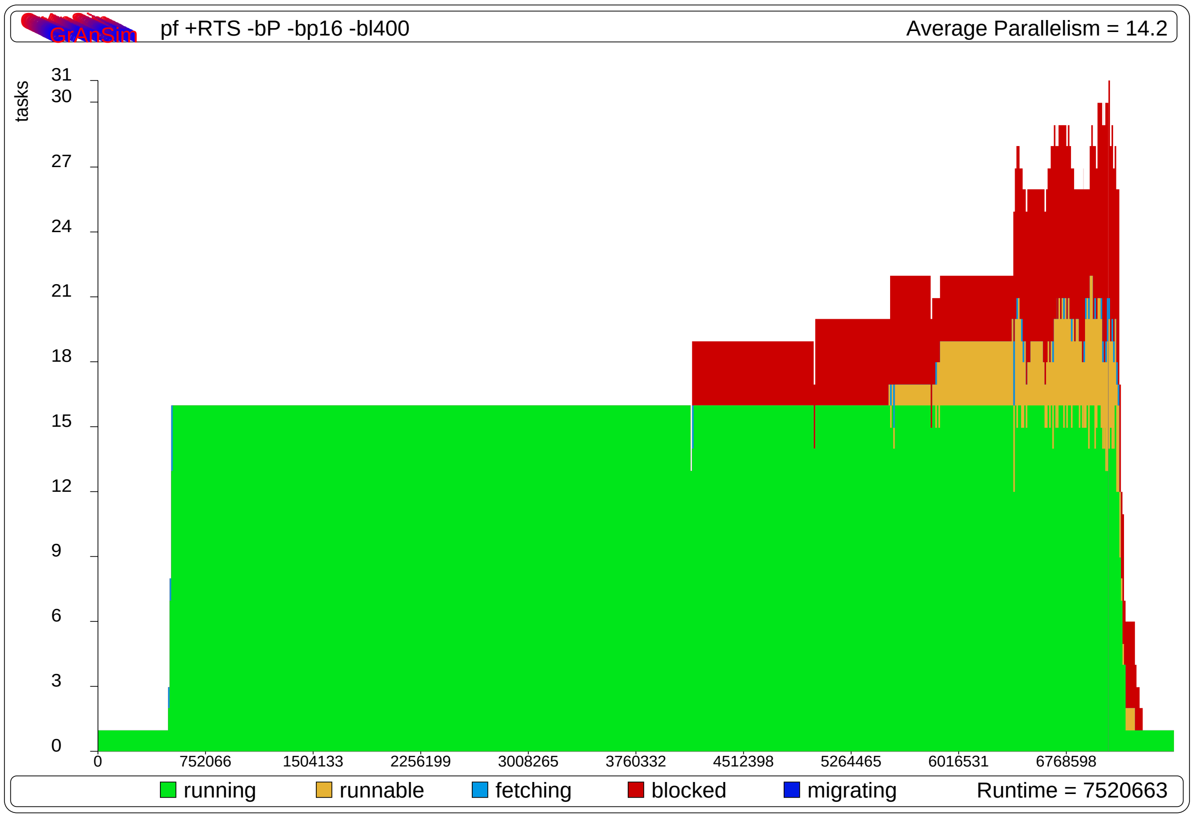
3. Use one of the visualisation tools (see Chapter 7 [Visualisation Tools], page 39) to examine the behaviour of the program. The first bet is to use '`gr2ps`', which generates a graph showing the overall activity of the machine in a global picture. For example

   ```
   gr2ps -O foo.gr
   ```

   generates an activity profile as a colour PostScript file '`foo.ps`'. It shows how many threads in total have been running, runnable (but not running), blocked (on data under evaluation), fetching (remote data) and migrating (to another processor) at each point during the execution. Other tools you might want to try are '`gr2pe`' (giving a per-PE activity profile) and '`gr2ap`' (giving a per-thread activity profile).

   Additionally, another set of visualisation tools allows to focus on the granularity of the generated threads. The most important one is '`gr2gran`', which generates bucket statistics showing the runtime of the individual threads (see Section 7.2 [Granularity Profiles], page 45).

As an example for an overall activity profile the graph below shows the result of running a parfib program (see Section 2.4 [Example], page 7) on 16 processors with a latency of 400 cycles.



Overall, for this simple program the utilisation is almost perfect as the green (medium-gray) area reaches up to 16 almost through the whole computation. Only at the end of the computation there is a significant number of runnable (amber or light-gray) and blocked (red or black) threads.

The header of the picture shows the average parallelism and the options used for this execution (e.g. the `-bp16` part shows that 16 processors have been simulated). The runtime shown in the footer of the picture is measured in machine cycles. In GranSim all times are given in machine cycles.

2 Overview

GranSim is mainly intended to be a research tool for studying the dynamic behaviour of parallel, lazy functional programs. By simulating the parallel execution it is possible to concentrate on the amount and structure of the parallelism exposed by the program without being distracted by 'external' details like load of the machine produced by other users or the basic network traffic. However, for obtaining results that are of relevance for a particular machine it is possible to set parameters in GranSim that very accurately model the behaviour of this machine.

2.1 Components of the GranSim System

The overall GranSim System consists of the following components:

- The *GranSim Simulator* proper.
- Some *Visualisation Tools*.
- The *GranSim Toolbox*.
- The *GranSim T-shirt* (not available, yet, sorry).

The GranSim simulator is built on top of the *runtime-system* (RTS) of the Glasgow Haskell Compiler (*GHC*). Thus, the major part of it is a (rather big) *runtime system for GHC*. This part is responsible for implementing the event driven simulation of parallelism. For example communication is modelled by adding new features to the appropriate routines in the sequential runtime-system.

For simulating parallelism the generated code has to be instrumented, which is achieved by a slightly modified code generator in GHC. To produce instrumented code the compile time option `-gransim` of the Haskell compiler has to be used. This is what we mean by saying 'compiling under GranSim'. Note that it is not possible to use object file compiled for another setup of GHC to generate a GranSim executable file.

Our approach is similar to that of *GUM* (a portable parallel implementation of Haskell), which has also been developed in this department. Both systems are basically extensions of the sequential runtime-system. GranSim is also similar to GUM in the way it implements several features like the communication mechansim. To a large extend both systems actually use the same code. See Chapter 11 [GranSim vs GUM], page 65 for a more detailled comparison.

In order to analyse the result of a simulation it is very important to visualise the abundance of data. To this end the GranSim system contains a set of *visualisation tools* that generate PostScript graphs, focusing on specific aspects of the execution. The most important usage of these tools is the generation of an overall activity profile of the execution.

The system contains a set of small tools, usually scripts processing the output of a simulation. These tools are collected in the so-called *GranSim Toolbox*.

The final component, the *GranSim T-shirt* has not been implemented, yet. If you have suggestions, feel free to email me.

2.2 Semi-explicit Parallelism

The underlying execution model of GranSim is one of semi-explicit parallelism where expressions that should be evaluated in parallel have to be annotated by using the `par` annotation. However, it is up to the runtime-system to decide when and where to create a parallel thread. It may even discard potential parallelism altogether.

The communication between the threads is performed via accessing shared data structures. If a thread tries to access a value that is under evaluation, it is put into a blocking queue attached to that closure. When the closure is updated by the result this blocking queue is released. So, the only extension of the sequential evaluation mechanism is the addition of blocking queues.

Because of these characteristics of hiding as many low-level details as possible in the runtime-system we call this model semi-explicit. It is only necessary to mark potential parallelism in the program. All issues related to communication, load balancing etc are handled by the runtime system and can be ignored by the programmer.

For example in the expression

```
let
  squares = [ i^2 | i <- [1..n] ]
in
squares `par` sum squares
```

the list of squares will be produced by one thread (the *producer*) and the result sum will be computed by another thread (the *consumer*). Note that this producer-consumer structure of the algorithm is solely determined by the data-dependencies in the program. All the programmer has to do is to annotate a named subexpression (in this example `squares`), which should be computed in parallel. Thread placement, communication and whether the thread should be created at all are up to the runtime-system.

In this introduction we don't go into the operational details of this example. See ⟨undefined⟩ [Example: Sum of Squares], page ⟨undefined⟩ for a discussion how to improve the parallelism of the algorithm.

This model of using `par` and `seq` annotations is the same as it is used for the `GRIP` machine and for the `GUM` system. In fact, there is a strong correspondence between the GranSim and GUM. This allows to carry results of a GranSim simulation over to a real parallel machine operating under GUM (this issue will be addressed in more detail in Chapter 11 [GranSim vs GUM], page 65.).

2.3 GranSim Modes

This section outlines a methodology for parallelising lazy functional programs. The development and performance tuning of a parallel algorithm typically proceeds in several stages:

1. *Profiling* of the sequential algorithm if the goal is the parallelisation of an already existing algorithm. This stage should give an idea about the 'big computations' in the program. The programmer can then concentrate on parallelising this part of the algorithm.
2. *Extracting* parallelism out of the algorithm. In our model of computation this means adding `seq` and `par` annotations to the program (See Chapter 4 [Parallelism Annotations], page 14, for a discussion of all available annotations).
3. *Restructuring* parts of the algorithm to increase the amount of parallelism. This can be called performance tuning on an abstract level where the performance of the algorithm is solely modelled by the amount of parallelism in the program.
4. *Optimising* the parallel algorithm for execution on a specific machine. In this stage low-level details have to be addressed.

To facilitate such a process of parallelisation the GHC system provides several tools. In particular GHC and GranSim support different modes reflecting different stages:

1. The sequential *profiling* setup of GHC allows to get accurate information about computation time and heap usage of program expressions. Especially for parallelising big sequential programs the profiler gives invaluable information for the proper parallelisation. A more detailed description of the profiler is part of the overall GHC documentation.

2. The *GranSim-Light* mode simulates the parallel execution of a program with an unbounded number of processors. In this mode no communication is modelled (i.e. access of remote data is as expensive as access of local data). Thus, this mode mainly shows the amount of parallelism in the program and indicates an upper bound of the parallelism when running the program on a real machine.
3. In the usual mode *GranSim* can simulate the execution of up to 32 processors[1] with an exact modelling of communication. Thus, the execution on a particular machine with the chosen characteristics is simulated.
4. A set of *visualisation tools* allows to generate activity and granularity profiles of the execution. These tools allow to generate overall profiles as well as per-PE and per-thread profiles (see Section 10.4 [Visualisation], page 63). This especially facilitates the performance tuning of the program.

2.4 A Simple Example Program

This section gives an example of how to write a simple parallel program in the parallel, lazy functional execution model. Due to a sad lack of imagination we take our good old friend, `nfib`, as an example program. This is the sequential function from which we start:

```
nfib :: Int -> Int
nfib 0 = 1
nfib 1 = 1
nfib n = nf1+nf2+1
         where nf1  = nfib (n-1)
               nf2  = nfib (n-2)
```

The straightforward idea for parallelising `nfib` is to start parallel processes for computing both recursive calls. This is expressed with the `par` annotation, which ‘sparks’ a parallel process for computing its first argument and whose result is its second argument (the exact operational semantics of `par` is discussed in more detail in chapter Chapter 4 [Parallelism Annotations], page 14.). This gives the following parallel program:

```
parfib :: Int -> Int
parfib 0 = 1
parfib 1 = 1
parfib n = nf2 `par` (nf1 `par` (nf1+nf2+1))
           where nf1 = parfib (n-1)
                 nf2 = parfib (n-2)
```

[1] In general the word size of the machine is the upper bound for the number of processors.

The drawback of this program is the blocking of the main task on the two created child-tasks. Only when both child tasks have returned a result, the main task can continue. It is more efficient to have one of the recursive calls computed by the main task and to spark only one parallel process for the other recursive call. In order to guarantee that the main expression is evaluated in the right order (i.e. without blocking the main task on the child task) the `seq` annotation is used:

```
parfib :: Int -> Int
parfib 0 = 1
parfib 1 = 1
parfib n = nf2 `par` (nf1 `seq` (nf1+nf2+1))
           where nf1 = parfib (n-1)
                 nf2 = parfib (n-2)
```

The operational reading of this function is as follows: First spark a parallel process for computing the expression `parfib (n-2)`. Then evaluate the expression `parfib (n-1)`. Finally, evaluate the main expression. If the parallel process has not finished by this time the main task will be blocked on `nf2`. Otherwise it will just pick up the result.

This simple example already shows several basic aspects of parallel, lazy functional programming:

- For adding parallelism to the program `par` annotations on local variables (that appear in the main expression) are used.
- To specify evaluation order of the program `seq` annotations are used. It reduces the first argument (usually a variable occurring in the definition of the second argument) to weak head normal form (WHNF).
- For the efficiency of the parallel program it is important to avoid unnecessary blocking during the execution.

Another aspect not shown in this example is the importance of evaluation degree. If the parallel expressions create a compound type (e.g. a list) then it is important that they are evaluated to a certain degree. Otherwise there won't be a lot of parallelism in the program. We will revisit this aspect again in Chapter 6 [Strategies], page 29.

Finally, here is the total example as a stand-alone program:

```
module Main where

import Parallel
```

```
main = getArgs abort ( \ a ->
       let
         n = head ( map (\ a1 -> fst ((readDec a1) !! 0)) a )
       in
       print ("parfib " ++ (show n) ++ " = " ++ (show (parfib n)) ++ "\n")  )

nfib :: Int -> Int
nfib 0 = 1
nfib 1 = 1
nfib n = nf1+nf2+1
         where nf1 = nfib (n-1)
               nf2 = nfib (n-2)

parfib :: Int -> Int
parfib 0 = 1
parfib 1 = 1
parfib n = nf2 `par` (nf1 `seq` (nf1+nf2+1) )
           where nf1 = parfib (n-1)
                 nf2 = parfib (n-2)
```

2.5 Running the Example Program

In a standard installation of GHC with GranSim enabled the example program can be compiled as usually but with adding the compiler option `-gransim`. So, if the above program is in file '`parfib.hs`' compile a GranSim version of the program with

```
ghc -gransim -fvia-C parfib.hs
```

GranSim requires a compilation using C code as an intermediate stage. Compiling with the native code generator is not supported.

To simulate the execution on a machine with 4 processors type

```
./a.out +RTS -bP -bp4
```

This will generate a granularity profile ('`.gr`' extension). For getting an overall activity profile of the execution use

```
gr2ps -O a.out.gr
```

Use your favourite PostScript viewer to examine the activity profile in '`a.out.ps`'. You'll get the best results with GNU ghostview and ghostscript version 2.6.1 or later.

3 Setting-up GranSim

[Warning: This is very drafty for now. If you have severe installation problems please let me know.]

This chapter describes how to get the latest version of GranSim, and how to install it.

The system requirements are basically the same as for GHC itself. So far, GranSim has been tested on SUNs under SunOS 4.1 and Solaris 2, and on DEC Alphas under OSF 1 and OSF 3.2. The visualisation tools require Perl, Bash and Gnuplot (only for granularity profiles).

3.1 Retrieving

The important addresses to check for the latest information about GHC and GranSim:

- GranSim Home Page – http://www.dcs.glasgow.ac.uk/fp/software/gransim.html
- Anonymous FTP Server – ftp://ftp.dcs.glasgow.ac.uk/pub/haskell/glasgow/
- GHC home page – http://www.dcs.glasgow.ac.uk/fp/software/ghc.html
- Glasgow FP group page – http://www.dcs.glasgow.ac.uk/fp/

To get the current version of GranSim go to the anonymous FTP Server at Glasgow and retrieve version 0.29 of GHC. The sources of the compiler with GranSim support are in

ghc-0.29-src.tar.gz

Only if you don't already have an installed version of GHC for bootstrapping the new version you'll have to download the HC files in that directory, too.

Check the README file on the FTP server to get more information about the different versions you can download. There should be at least one binary installation of GranSim available for Suns. If you have the right machine and operating system you can just download and unpack this version.

3.2 Installing

Follow the instructions in the GHC Installation Guide (in the subdirectory '`ghc/docs/install_guide`'.

Note that so far GranSim has only been tested with Haskell 1.2. I recommend using that version until I have had a closer look at the interaction of GranSim with the new, shiny GHC for 1.3.

The only things different from a normal installation are:

- Call the `configure` script with the additional option
  ```
  --enable-gransim
  ```
- Typing
  ```
  make all
  ```
  (after '`configure`' and '`STARTUP`') should create all necessary libraries. All GranSim specific libraries (and hi files) have the extension '`_mg.a`' ('`_mg.hi`').

You can find a copy of the GranSim User's Guide in the subdirectory '`ghc/docs/gransim`'.

3.3 Trouble Shooting

If an installation with `make all` doesn't run through smoothly, try to build the components by hand. Often it's just a problem with dependencies or old interface files. Try to recompile the modules by hand on which the failing module depends.

The top level files for the installation are

- '`hsc`' in the directory '`ghc/compiler/`': the compiler proper.
- The libraries '`libHS_mg.a`' and '`libHS13_mg.a`' in the directory '`ghc/lib/`': these are the necessary built-in libraries (preludes). The libraries '`libHSghc_mg.a`' and '`libHShbc_mg.a`' are optional.
- '`libHSrts_mg.a`' in the directory '`ghc/runtime/`': the runtime-system.

You can create a version of the RTS with debugging information (and all GranSim debugging options) by typing

```
make EXTRA_HC_OPTS="-optcO-DGRAN -optcO-DGRAN_CHECK -optcO-DDEBUG -optcO-g"
     libHSrts_mg.a
```

For those Bravehearts who want to actually hack on the RTS I recommend first having a closer look at the hackers sections of the general GHC User's Guide (see ⟨undefined⟩ [(ghc-user-guide.info)Top], page ⟨undefined⟩). Eventually, I'll add GranSim specific stuff to the chapter about the internals of GranSim (see ⟨undefined⟩ [GranSim Internals], page ⟨undefined⟩) but for now there is not much in this chapter.

4 Parallelism Annotations

This chapter discusses the constructs for adding parallelism to a Haskell program. All constructs are annotations that do not change the semantics of the program (one exception to this rule is the annotation for forcing sequential evaluation since it may change the strictness properties of the program).

Normally the basic annotations that are discussed first are sufficient to write and tune a parallel program. The advanced annotations allow to name static spark sites and to provide granularity information.

NB: To use these annotations the module `Parallel` must be imported as shown in the introductory example (see Section 2.4 [Example], page 7).

4.1 Basic Annotations

The two basic parallelism annotations in GranSim (as well as in GUM and GRIP) are `par` and `seq`. Both take two arguments and return the value of the second argument as result. However, they differ in their operational behaviour:

par `:: a -> b -> b` Annotation

`par x y` creates a spark for evaluating *x* and returns the value of *y* as a result.

seq `:: a -> b -> b` Annotation

`seq x y` reduces *x* to weak head normal form and then returns the value of *y* as a result.

The process of sparking a parallel thread creates potential parallelism (a *spark*). Such a spark may be turned into a thread or may be discarded by the runtime system. Several important facts should be noted about these annotations:

- Semantically the value of `seq x y` equals that of *y* only if the evaluation of *x* terminates and is error free. Thus, `seq` is strict in its first argument.
- Typically *x* is a local variable and *y* is an expression with *x* as a free variable. A typical idiom for using `par` is

```
let x = ... in x `par` f x
```

If x does not occur in y speculative parallelism is created, which might trigger the evaluation of objects that are not needed for computing the result. This may also change the termination properties of the program.

- The evaluation degree of y is unaffected by these annotations. It is solely determined by the expression in which the result of this annotated expression is used.

4.2 Advanced Annotations

The annotations in the previous section are actually specialisations of the following set of built-in functions which can be called directly:

parGlobal `:: Int -> Int -> Int -> Int -> a -> b -> b` Annotation

`parGlobal n g s p x y` creates a spark for evaluating x and returns the value of y as a result. The spark gets the name n with the granularity information g, the size information s and a degree of parallelism of p. The g, s and p fields just forward information to the runtime system.

parLocal `:: Int -> Int -> Int -> Int -> a -> b -> b` Annotation

`parLocal n g s p x y` behaves like `parGlobal` except that the thread (if it is generated at all) must be executed on the current processor.

parAt `:: Int -> Int -> Int -> Int -> a -> b -> c -> c` Annotation

`parAt n g s p v x y` behaves like `parGlobal` except that the thread (if it is generated at all) must be executed on the processor that contains the expression v.

parAtForNow `:: Int -> Int -> Int -> Int -> a -> b -> c -> c` Annotation

`parAtForNow n g s p v x y` behaves like `parAt` except that the generated thread may be migrated to another processor during the execution of the program.

4.3 Experimental Annotations

Some experimental annotations currently available in GranSim are:

parAtAbs :: `Int -> Int -> Int -> Int -> Int -> a -> b -> b` Annotation

`parAtAbs n g s p m x y` behaves like `parAt` except that the thread must be executed on processor number *m* (the processors are numbered from 0 to p-1 where p is the runtime system argument supplied via the `-bp` option).

parAtRel :: `Int -> Int -> Int -> Int -> Int -> a -> b -> b` Annotation

`parAtRel n g s p m x y` behaves like `parAtAbs` except that the value of *m* is added to the number of the current processor to determine the target processor.

copyable :: `a -> a` Annotation

`copyable x` marks the expression *x* to be copyable. This means that the expression will be transferred in its unevaluated form if it is needed on another processor. This may duplicate work.

This annotation is not yet implemented.

noFollow :: `a -> a` Annotation

`noFollow x`

This annotation is not yet implemented.

5 Runtime-System Options

GranSim provides a large number of runtime-system options for controlling the simulation. Most of these options allow to specify a particular parallel architecture in very much detail.

As general convention all GranSim related options start with `-b` to separate them from other GHC RTS options. To separate the RTS options from usual options given to the Haskell program the meta-option `+RTS` has to be used.

If you are not interested in the details of the available options and just want to specify a somewhat generic setup for one class of parallel machines go to the last section of this chapter (see Section 5.10 [Specific Setups], page 26).

5.1 Basic Options

The options in this section are probably the most important GranSim options the programmer has to be aware of. They define the basic behaviour of GranSim rather than going down to a low level of specifying machine characteristics.

`-bP` RTS Option

This option controls the generation of a GranSim profile (see Chapter 8 [GranSim Profiles], page 50). By default a reduced profile (only `END` events) is created. With `-bP` a full GranSim profile is generated. Such a profile is necessary to create activity profiles. With `-b-P` no profile is generated at all.

`-bs` RTS Option

Generate a spark profile. The GranSim profile will contain events describing the creation, movement, consumption and pruning of sparks. *Note:* This option will drastically increase the size of the generated GranSim profile.

`-bh` RTS Option

Generate a heap profile. The GranSim profile will contain events describing heap allocations. *Note:* This option will drastically increase the size of the generated GranSim profile.

`-bp`*n* RTS Option

Choose the number of processors to simulate. The value of *n* must be less than or equal to the word size on the machine (i.e. usually 32). If *n* is 0 GranSim-Light mode is enabled.

`-bp:` RTS Option

Enable GranSim-Light (same as `-bp0`). In this mode there is no limit on the number of processors but no communication costs are recorded.

5.2 Special Features

The options in this section allow to simulate special features in the runtime-system of the simulated machine. This allows to study how these options influence the behaviour of different kinds of parallel machines. All of these flags can be turned of by inserting a `-` symbol after `b` (as in `-b-P`).

5.2.1 Asynchronous Communication

If a thread fetches remote data by default the processor is blocked until the data arrives. This *synchronous communication* is usually only advantageous on machines with very low latency. On such machines it is better to wait and avoid the overhead of a context switch. Synchronous communication may also increase data locality as a thread is only descheduled when it gets blocked on data that is under evaluation by another thread.

On machines with high latency *asynchronous communication* is usually better as it allows the processor to perform some useful computation while a thread waits for the arrival of remote data. However, the processor might be even more aggressive in trying to get work while another thread is waiting for data. The aggressiveness of the processor to get new work is determined by the *fetching strategy*. Currently five different strategies are supported.

`-by`*n* RTS Option

Choose a Fetching Strategy (i.e. determine what to do while the running thread fetches remote data):

0. Synchronous communication (default).

1. This and all higher fetching strategies implement asynchronous communication. This strategy schedules another runnable thread if one is available. This gives the same behaviour as `-bZ`.
2. If no runnable thread is available a local spark is turned into a thread. This adds task creation overhead to context switch overhead for asynchronous communication.
3. If no local sparks are available the processor tries to acquire a remote spark.
4. If the processor can't get a remote spark it tries to acquire a runnable thread from another busy processor provided that migration is also turned on (`-bM`).

`-bZ` RTS Option

Enable asynchronous communication. This causes a thread to be descheduled when it fetches remote data. Its processor schedules another runnable thread. If none is available it becomes idle. This gives the same behaviour as `-by1`.

Note that fetching strategies 3 and 4 involve the sending of messages to other processors. Therefore, it's likely that by the time a spark (or thread) has been fetched, the original thread has already received its data and is being executed again. Therefore, in most cases strategies 1 and 2 yield better results than 3 and 4.

5.2.2 Bulk Fetching

When fetching remote data there are several possibilities how to transfer the data. The options `-bG` and `-bQ`*n* allow to choose among them. By default GranSim uses *incremental fetching* (also called single closure fetching, or lazy fetching). This means that only the closure that is immediately required is fetched from a remote processor. Again, this strategy is preferable for low latency systems as it minimises the total amount of data that has to be transferred. However, if the overhead for creating a packet and for sending a message are high it is better to perform *bulk fetching*, which transfers a subgraph with the required closure as its root. The size of the subgraph can be bounded by specifying the maximal size of a packet or by specifying the maximal number of *thunks* (unevaluated closures) that should be put into one packet. The way how to determine which closures to put into a packet is called *packing strategy*.

`-bG` RTS Option

Enable bulk fetching. This causes the whole subgraph with the needed closure as its root to be transferred.

`-bQ`*n* RTS Option

Pack at most *n*-1 non-root thunks into a packet. Choosing a value of 1 means that only normal form closures are transfered with the possible exception of the root, of course. The value 0 is interpreted as infinity (i.e. pack the whole subgraph). This is the default setting.

`-Q`*n* RTS Option

Pack at most *n* words in one packet. This limits the size of the packet and does not distinguish between thunks and normal forms. The default packet size is 1024 words.

The option for setting the packet size differs from the usual GranSim naming scheme as it is also available for GUM. In fact, both implementations use almost the same source code for packing a graph.

5.2.3 Migration

When an idle processor looks for work it first checks its local spark pool, then it tries to get a remote spark. It might happen that no sparks are available in the system any more and that some processors have several runnable threads. In such a situation it might be advantageous to transfer a runnable thread from a busy processor to an idle one. However, this *thread migration* is very expensive and should be avoided unless absolutely necessary. Therefore, by default thread migration is turned off.

`-bM` RTS Options

Enable thread migration. When an idle process has no local sparks and can't find global sparks it tries to migrate (steal) a runnable thread from another busy processor.

Note that thread migration often causes a lot of fetching in order to move all required data to the new processor, too. This bears the risk of destroying data locality.

5.3 Communication Parameters

The options in this section allow to specify the overheads for communication. Note that in GranSim-Light mode all of these values are irrelevant (communication is futile).

`-bl`*n* RTS Option

Set the latency in the system to *n* machine cycles. Typical values for shared memory machines are 60 – 100 cycles, for GRIP (a closely-coupled distributed memory machine) around 400 cycles and for standard distributed memory machines between 1000 and 5000 cycles. The default value is 1000 cycles.

`-ba`*n* RTS Option

Set the additional latency in the system to *n* machine cycles. The additional latency is the latency of follow-up packets within the same message. Usually this is much smaller than the latency of the first packet (default: 100 cycles).

`-bm`*n* RTS Option

Set the overhead for message packing to *n* machine cycles. This is the overhead for constructing a packet independent of its size.

`-bx`*n* RTS Option

Set the overhead for tidying up the packet after sending it to *n* machine cycles. On some systems significant work is needed after having sent a packet.

`-br`*n* RTS Option

Set the overhead for unpacking a message to *n* machine cycles. Again, this overhead is independent of the message size.

`-bg`*n* RTS Option

Set the overhead for fetching remote data to *n* machine cycles. By default this value is two times latency plus message unpack time.

5.4 Runtime-System Parameters

The options in this section model overhead that is related to the runtime-system of the simulated parallel machine.

`-bt`*n* RTS Option

Set the overhead for thread creation to *n* machine cycles. This overhead includes costs for initialising a control structure describing the thread and allocating stack space for the execution.

`-bq`*n* RTS Option

Set the overhead for putting a thread into the blocking queue of a closure to *n* machine cycles.

`-bc`*n* RTS Option

Set the overhead for scheduling a thread to *n* machine cycles.

`-bd`*n* RTS Option

Set the overhead for descheduling a thread to *n* machine cycles.

`-bn`*n* RTS Option

Set the overhead for global unblocking (i.e. blocking on a remote closure) to *n* machine cycles. This value does not contain the overhead caused by the communication between the processors.

`-bu`*n* RTS Option

Set the overhead for local unblocking (i.e. putting a thread out of a blocking queue and into a runnable queue) to *n* machine cycles. This value does not contain the overhead caused by the communication between the processors.

5.5 Processor Characteristics

The options in this section specify the characteristics of the microprocessor of the simulated parallel machine. To this end the instructions of the processor are divided into six groups. These groups have different weights reflecting their different relative costs.

The groups of processor instructions are:

- Arithmetic instructions
- Load instructions
- Store instructions
- Branch instructions
- Floating point instruction
- Heap allocations

The options for assigning weights to these groups are:

`-bA`*n* RTS Option

Set weight for arithmetic operations to *n* machine cycles.

`-bL`*n* RTS Option

Set weight for load operations to *n* machine cycles.

`-bS`*n* RTS Option

Set weight for store operations to *n* machine cycles.

`-bB`*n* RTS Option

Set weight for branch operations to *n* machine cycles.

`-bF`*n* RTS Option

Set weight for floating point operations to *n* machine cycles.

`-bH`*n* RTS Option

Set weight for heap allocations to *n* machine cycles.

Strictly speaking, the heap allocation costs is a parameter of the runtime-system. However, in our underlying machine model allocating heap is such a basic operation that one can think of it as a special instruction.

5.6 Granularity Control Mechanisms

There are three granularity control mechanisms:

1. *Explicit threshold*

   No spark whose priority is smaller than a given threshold will be turned into a thread.
2. *Priority sparking*

   The spark queue is sorted by priority. This guarantees that the highest priority spark is turned into a thread. Priorities are not maintained for threads.
3. *Priority scheduling*

   The thread queue is sorted by priority, too. This guarantees that the biggest available thread is scheduled next. This imposes a higher runtime overhead.

`-bY`*n* RTS Option

Use the value *n* as a threshold when turning sparks into threads. No spark with a priority smaller than *n* will be turned into a thread.

`-bX`*x* RTS Option

Enable priority sparking. The letter *x* indicates how to use the granularity information attached to a spark site in the source code:

- Use the granularity information field as a priority.
- `I` Use the granularity information field as an inverse priority.
- `R` Ignore the granularity information field and use a random priority.
- `N` Ignore the granularity information field and don't use priorities at all.

`-bI` RTS Option

Enable priority scheduling.

`-bK`*n* RTS Option

Set the overhead for inserting a spark into a sorted spark queue to *n* machine cycles.

`-bO`*n* RTS Option

Set the overhead for inserting a thread into a sorted thread queue to *n* machine cycles.

5.7 Miscellaneous Options

`-bC` RTS Option

Force the system to eagerly turn a spark into a thread. This basically disables the lazy thread creation mechanism of GranSim and ensures that no sparks are discarded (except for sparks whose closures are already in normal form).

`-be` RTS Option

Enable deterministic mode. This means that no random number generator is used for deciding where to get work from. With this option two runs of the same program with the same input will yield exactly the same result.

`-bT` RTS Option

Prefer stealing threads over stealing sparks when looking for remote work. This is mainly an experimental option.

`-bN` RTS Option

When creating a new thread prefer sparks generated by local closures over sparks that have been stolen from other processors. This is mainly an experimental option, which might improve data locality.

`-bf`*n* RTS Option

Specify the maximal number of outstanding requests to acquire sparks from remote processors. High values of *n* may cause a more even distribution of sparks avoiding bottlenecks caused by a 'spark explosion' on one processor. However, this might harm the data locality. The default value is 1.

`-bw`*n* RTS Option

Set the time slice of the simulator to *n* machine cycles. This is an internal variable that changes the behaviour of the simulation rather than that of the simulated machine. A longer time slice means faster but less accurate simulation. The default time slice is 1000 cycles.

5.8 Debugging Options

These options are mainly intended for debugging GranSim. Only those options that might be of interest to the friendly (i.e. non-hacking) user of GranSim are listed here for now.

Note: These options are only available if the RTS has been compiled with the cpp flag `GRAN_CHECK` (see Section 3.2 [Installing], page 11).

`-bDE` RTS Option

Print an event statistics at the end of the computation. This also includes a statistics about the packages sent (if bulk fetching) has been enabled.

If you are *really* interested in all the hidden options in GranSim look into the file '`ghc/runtime/main/RtsFlags.lc`'.

5.9 General GHC RTS Options

Some options of the GHC runtime-system that are not specific for GranSim are of special interest, too. They are discussed in this section.

`-o`*n* RTS Option

This option sets the initial size of the stack of a thread to *n* words. This might be of importance for GranSim-Light, which can create an abundance of parallel threads filling up the heap. The default stack size in GranSim-Light is already reduced to 200 words (usually the default is 1024 words). If you run into problems with heap size in a GranSim-Light setup you might want to reduce it further.

`-S`*f* RTS Option

Print messages about garbage collections to the file *f* (or to *stderr*).

`-F2s` RTS Option

Use a two-space garbage collector instead of a generational garbage collector. Previous versions of GranSim had problems with the latter. If you experience problems try this option and send me a bug report (see Chapter 13 [Bug Reports], page 67).

5.10 Specific Setups

When using GranSim a programmer often just wants to specify the general architecture of the machine rather than going down to the details of a specific machine. To facilitate this approach this section presents examples of standard set-ups for GranSim.

Note that these setups specify the characteristics of a machine, but not of the runtime-system. Thus characteristics like thread creation costs are left open. However, the default setting fairly closely reflect the real costs for example under GUM. So, unless you have a different implementation of runtime-system details in mind the default settings should be sufficiently accurate.

5.10.1 The Ideal GranSim Setup

This setup reflects the ideal case, where communication is for free and where there is no limit on the number of processors. This is used to show the maximal amount of parallelism in the program.

Using such a *GranSim-Light* setup is usually the first step in tuning the performance of a parallel, lazy functional program (see Section 2.3 [GranSim Modes], page 6).

The typical GranSim-Light setup is:

```
+RTS -bP -b:
```

5.10.2 GranSim Setup for Shared-Memory Machines

In a shared memory machine the latency is roughly reduced to the costs of a load operation. Potentially, some additional overhead for managing the shared memory has to be added. Also the caching mechanism might be more expensive than in a sequential machine. In general, the latency should be between 5 and 20 machine cycles.

For machines where the latency is of the same order of magnitude as loading and storing data, it is reasonable to assume incremental, synchronous communication. Migration should also be possible.

This gives the following setup (for 32 processors):

```
+RTS -bP -bp32 -bl10 -b-G -by0 -bM
```

5.10.3 GranSim Setup for Strongly Connected Distributed Memory Machines

Strongly connected DMMs put a specific emphasis on keeping the latency in the system as low as possible. One example of such a machine is GRIP, which has been built specifically for performing parallel graph reduction. Therefore, this setup is of special interest for us.

Most importantly the latency in such machines is typically between 100 and 500 cycles (400 for GRIP). Furthermore, the GRIP runtime-system, as an example for such kind of machines, uses incremental, synchronous communication. Migration is also possible.

This gives the following setup (for 32 processors):

```
+RTS -bP -bp32 -bl400 -b-G -by0 -bM
```

5.10.4 GranSim Setup for Distributed Memory Machines

General distributed memory machines usually have latencies that are a order of magnitude higher than that of strongly connected DMMs. However, especially in this class of machines the differences between specific machines are quite significant. So, I strongly recommend to use the exact machine characteristics if GranSim should be used to predict the behaviour on such a machine.

The high latency requires a fundamentally different runtime-system to avoid long delays for fetching remote data. Therefore, usually synchronous bulk fetching is used. I'd recommend choosing a fetching strategy of 1 or 2 (it's hard to say which one is better in general). Thread migration is such expensive on DMMs that it is often not supported at all.

This gives the following setup (for 32 processors):

```
+RTS -bP -bp32 -bl2000 -bG -by2 -b-M
```

6 Parallel Functional Programming in the Large: Strategies

[See the strategy paper: "Strategies for writing parallel non-strict programs", Trinder P.W., Loidl H.W., Hammond K., Peyton Jones S.L. (in preparation).]

This chapter deals with parallel, lazy functional programming in the large. It introduces the notion of strategies and shows how they make it easier to develop large parallel programs.

6.1 Motivation for Strategies

The following naive version `parmap` often does not give good parallelism:

```
parmap :: (a -> b) -> [a] -> [b]
parmap  f []     = []
parmap  f (x:xs) = fx `par` pmxs `par` (fx:pmxs)
                   where fx = f x
                         pmxs = parmap f xs
```

If in this version the spark for `pmxs` is discarded almost all parallelism in the program is lost. On the other hand, if all `par`'s are executed more parallel threads than necessary are created (two for each list element).

An alternative version of `parmap` reduces the total number of generated threads by replacing the second `par` with a `seq`:

```
parmap :: (a -> b) -> [a] -> [b]
parmap  f []     = []
parmap  f (x:xs) = fx `par` pmxs `seq` (fx:pmxs)
                   where fx = f x
                         pmxs = parmap f xs
```

The problem of this version is that in a context that requires the result of parmap only in weak head normal form, the recursive call to parmap will be deferred until its value is needed. This drastically reduces the total amount of parallelism in the program. Especially, if such a function is used as the producer in a program with producer-consumer parallelism, this gives very poor parallelism (see Section 6.4 [Sum of Squares (Example)], page 33).

To improve this behaviour one can write a forcing function, which guarantees that all of the result is demanded immediately:

```
forcelist []     = ()
forcelist (x:xs) = x `seq` (forcelist xs)
```

Using this function in `parmap` gives the following function:

```
parmap :: (a -> b) -> [a] -> [b]
parmap  f []     = []
parmap  f (x:xs) = fx `par`  (forcelist pmxs) `seq` (fx:pmxs)
                   where fx = f x
                         pmxs = parmap f xs
```

However, following this approach yields a big set of forcing functions and a mixture of defining the result and driving the computation. We want to avoid both.

6.2 The Main Idea

[For now mainly the abstract of the Strategy paper]

Separate the components of a parallel, lazy functional program:

1. Parallelism
2. Evaluation degree
3. Definition of the result

In most current parallel non-strict functional languages, a function must both describe the value to be computed, and the *dynamic behaviour*. The dynamic behaviour of a function has two aspects: *parallelism*, i.e. what values could be computed in parallel, and *evaluation-degree*, i.e. how much of each value should be constructed. Our experience is that specifying the dynamic behaviour of a function obscures its semantics. Strategies have been developed to address this problem; a strategy being an explicit description of a function's potential dynamic behaviour. The philosophy is that *it should be possible to understand the semantics of a function without considering it's dynamic behaviour.*

In essence a strategy is a runtime function that traverses a data structure specifying how much of it should be evaluated, and possibly sparking threads to perform the construction in parallel.

Because strategies are functions they can be defined on any type and can be combined to specify sophisticated dynamic behaviour. For example a strategy can control thread granularity or specify evaluation over an infinite structure. A disadvantage is that a strategy requires an additional runtime pass over parts of the data structure. Example programs are given that use strategies on divide-and-conquer, pipeline and data-parallel applications.

6.3 Example Program: Quicksort

This section compares two version of parallel quicksort: one using a forcing function and another one using strategies.

6.3.1 Parallel Quicksort using Forcing Functions

The following naive attempt to introduce parallelism in quicksort fails because the threads generating `loSort` and `hiSort` only create a single cons cell.

```
quicksortN []     = []
quicksortN [x]    = [x]
quicksortN (x:xs) =
 losort `par`
 hisort `par`
 result
  where
   losort = quicksortN [y|y <- xs, y < x]
   hisort = quicksortN [y|y <- xs, y >= x]
   result = losort ++ (x:hisort)
```

The current practice of parallel Haskell programmers is to introduce a *forcing function* that forces the evaluation of the required elements of a data structure. For example

```
forceList :: [a] -> ()
forceList [] = ()
forceList (x:xs) = x `seq` forceList xs
```

Quicksort can be rewritten to have the desired behaviour using `forceList` as follows.

```
quicksortF []      = []
quicksortF [x]     = [x]
quicksortF (x:xs)  =
```

```
(forceList losort) `par`
(forceList hisort) `par`
losort ++ (x:hisort)
 where
  losort = quicksortF [y|y <- xs, y < x]
  hisort = quicksortF [y|y <- xs, y >= x]
```

To obtain the required dynamic behaviour for the `parMap` example we might use `forceList` within the definition of `f`:

```
f x = forceList result `seq` result
      where
        result = [fib x]
```

When programs are written in this style, a number of forcing functions are required: at least one for each type, e.g. `forcePair`. To obtain good dynamic behaviour, `par`, `seq` and forcing functions are inserted throughout the program. In consequence the dynamic behaviour and static semantics of the computation are intertwined. This intertwining obscures the semantics of the program. Small-scale programs remain manageable, but in larger programs, particularly those with complex data structures and parallel behaviour, discerning the meaning of a program becomes very hard.

6.3.2 Parallel Quicksort using Strategies

In the strategy version we can specify the evaluation degree by applying one of the predefined strategies to the components of the result. These strategies are then combined by using either `par` or `seq` to specify the parallelism.

In `quicksort`, each parallel thread should construct all of its result list, and `rnf` expresses this neatly. The interesting equation becomes

```
quicksortS (x:xs)  = losort ++ (x:hisort) `using` strategy
    where
                     losort = quicksortS [y|y <- xs, y < x]
                     hisort = quicksortS [y|y <- xs, y >= x]
                     strategy result =
                       rnf losort `par`
                       rnf hisort `par`
                       rnf result `par`
                       ()
```

6.4 Example Program: Sum of Squares

Although the sum-of-squares program is a very simple producer-consumer program, it already shows some basic problems in this model. Unfortunately, the simple version of the program, which was presented in Section 2.2 [Semi-explicit Parallelism], page 5, produces hardly any parallelism at all. The reason for this behaviour is that because of the lazy evaluation approach the producer will only create the top cons cell of the intermediate list. When the consumer then demands the rest of the list it computes `squares` as well as the result sum. In the example below version 1 (with `res_naive`) shows this behaviour.

To improve the parallelism one can use a *forcing function* on the intermediate list. The parallel thread is then the application of this forcing function on the `squares` list. As expected, this creates two threads that are computing most of the time and occasionally have to exchange data. Version 2 (with `res_force`) shows this behaviour.

A better way to achieve the same operational behaviour is to define a strategy, how to compute the result. The strategy describes the parallelism and the evaluation degree. In doing so, it uses the reduce to normal form (`rnf`) strategy, which is defined in the class `NFData`. Compared to version 2 the strategy version 3 (with `res_strategy`) achieves a clean separation of defining the result and specifying operational details like parallelism and evaluation degree.

```
module Main where

import StrategiesVII

main = getArgs abort ( \ a ->
       let
         args :: [Int]
         args = map (\ a1 -> fst ((readDec a1) !! 0)) a
         version =  args !! 0
         n = args !! 1
         -- l = [1..]
         squares :: [Int]
         squares = [ i^2 | i <- [1..n] ]
         -- Naive version sparks a producer for list squares but doesn't force it
         res_naive = squares `par` sum squares
         -- This version sparks a forcing function on the list (producer)
         res_force = (foldl1 seq squares) `par` sum squares
         -- The strategy version
         res_strat = sum squares `using` (rnf squares `par` rnf)

         res = case version of
                1 -> res_naive
```

```
              2 -> res_force
              3 -> res_strat
              _ -> res_naive
       str = case version of
              1 -> "Naive version"
              2 -> "Forcing version"
              3 -> "Strategy version"
              _ -> "Naive version"
     in
     print ("Sum of squares (" ++ str ++ ") of length " ++ (show n) ++
            " = " ++ (show res) ++ "\n")  )
```

When running the naive version of this program the spark for the consumer process is pruned and all the computation of the consumer is subsumed by the producer. This gives a totally sequential program. The relevant parts of the GranSim profile are:

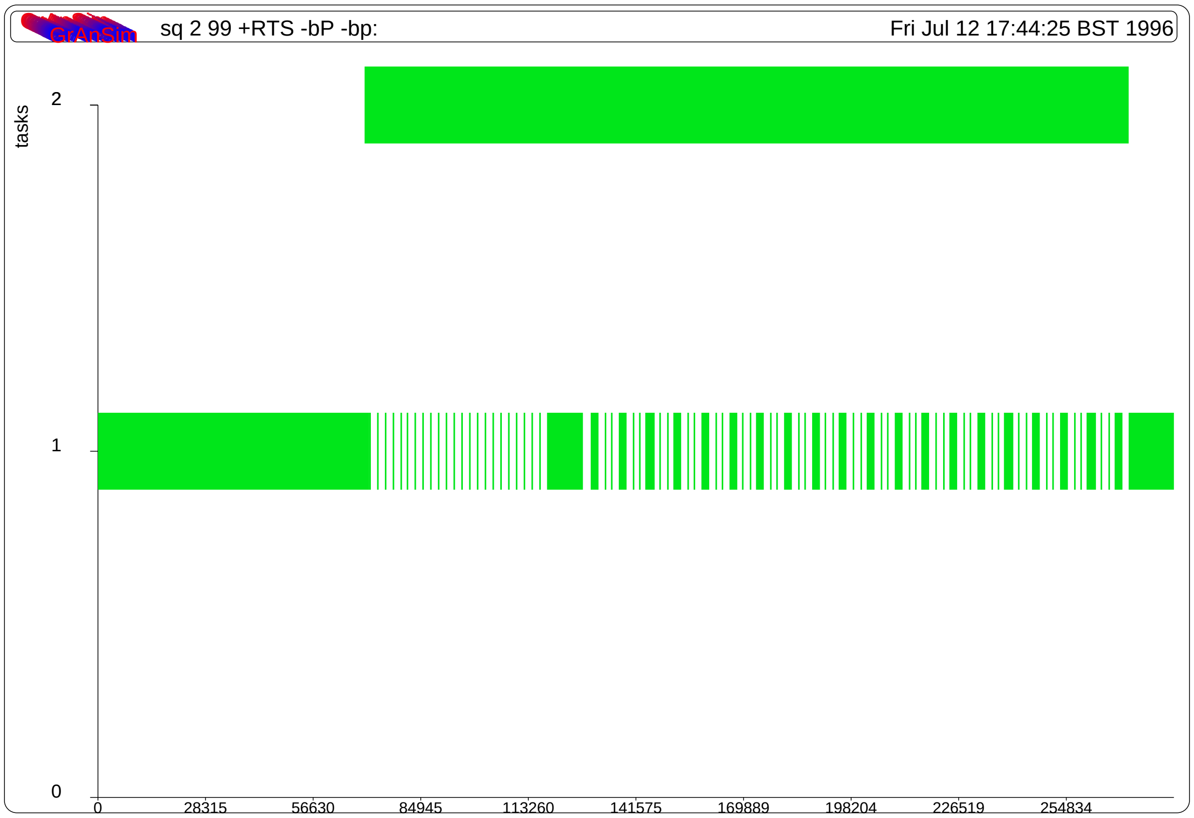
```
Granularity Simulation for sq 1 99 +RTS -bP -bp: -bs

++++++++++++++++++++

PE  0 [0]: START     0 0x100620 [SN 0]
PE  0 [67445]: SPARK     20004 0x347dbc [sparks 0]
PE  0 [68241]: PRUNED    20004 0x347dbc [sparks 1]
PE  0 [327088]: END 0, SN 0, ST 0, EXP F, BB 847, HA 5426, RT 327088, BT 0 (0), FT 0 (0), LS 0, GS 1, MY T
```

The other two versions of the program create one producer and one consumer thread as expected.

The per thread activity profile shows the behaviour of the strategy version of the program. Thread 1 is the main thread which computes `sum squares` and therefore is the consumer process. It creates thread 2 as the producer process, which evaluates `squares`. The producer is evaluating the list to normal form and therefore is one continuous thread. The consumer has to wait for the results from the producer from time to time and is blocked during these periods.



6.5 Using a Class of Strategies

[For now this section is just a short discussion of the strategies module]

This section discusses the contents of the strategies module. It shows how strategies are implemented on top of the basic `par` and `seq` annotations. Haskell's overloading mechanism is used to define a strategy for reducing to normal form. The current implementation is based on Haskell 1.2 but in the near future we want to use constructor classes as provided in Haskell 1.3.

6.5.1 Type Definition of Strategies

A strategy does not return a meaningful result it only specifies parallelism and evaluation degree. Therefore, we use the following type for strategies

```
type Strategy a = a -> ()
```

We can now define a *strategy application*, which adds information about the operational behaviour of the execution to an expression

```
using :: a -> Strategy a -> a
using x s = s x `seq` x
```

Note that `` x `using` s `` is a projection on `x`, i.e. both

- a retraction: `` x `using` s [ x ``
- \- and idempotent: `` (x `using` s) `using` s = x `using` s ``

6.5.2 Primitive Strategies

The primitive strategies are basically just syntactic sugar to fit `par` and `seq` into the strategy scheme.

`sPar` is a strategy corresponding to `par`, i.e. `` x `par` e <=> e `using` sPar x ``

```
sPar :: a -> Strategy b
sPar x y = x `par` ()
```

`sSeq` is a strategy corresponding to `seq`, i.e. `` x `seq` e <=> e `using` sSeq x ``

```
sSeq :: a -> Strategy b
sSeq x y = x `seq` ()
```

6.5.3 Basic Strategies

Three basic strategies describe the evaluation degree of a data structure:

- `r0` is a strategy which performs no evaluation at all

  ```
  r0 :: Strategy a
  r0 x = ()
  ```
- `rwhnf` reduces a data structure to weak head normal form

  ```
  rwhnf :: Strategy a
  rwhnf x = x `seq` ()
  ```

- `rnf` reduces a data structure to normal form. This strategy can't be defined independent of the type of the data structure. Therefore, we define a class `NFData` and overload the `rnf` strategy

```
class NFData a where
  -- rnf reduces it's argument to (head) normal form
  rnf :: Strategy a
  -- Default method. Useful for base types. A specific method is necessary for
  -- constructed types
  rnf = rwhnf
```

For primitive types like `Int`s and `Char`s the default methods can be used

```
instance NFData Int
instance NFData Char
...
```

6.5.4 Strategy Combinators

When defining a strategy on a compound data type we can use these three basic strategies and compose them with the strategy combinators below. These combinators describe the parallelism in the evaluation.

```
-- Pairs
instance (NFData a, NFData b) => NFData (a,b) where
  rnf (x,y) = rnf x `seq` rnf y

seqPair :: Strategy a -> Strategy b -> Strategy (a,b)
seqPair strata stratb (x,y) = strata x `seq` stratb y

parPair :: Strategy a -> Strategy b -> Strategy (a,b)
parPair strata stratb (x,y) = strata x `par` stratb y

-- Lists
instance NFData a => NFData [a] where
  rnf [] = ()
  rnf (x:xs) = rnf x `seq` rnf xs

-- Applies a strategy to every element of a list in parallel
parList :: Strategy a -> Strategy [a]
parList strat []     = ()
parList strat (x:xs) = strat x `par` (parList strat xs)

-- Applies a strategy to the first  n elements of a list  in parallel
parListN :: (Integral b) => b -> Strategy a -> Strategy [a]
```

```
parListN n strat []     = ()
parListN 0 strat xs     = ()
parListN n strat (x:xs) = strat x `par` (parListN (n-1) strat xs)

-- Sequentially applies a strategy to each element of a list
seqList :: Strategy a -> Strategy [a]
seqList strat []     = ()
seqList strat (x:xs) = strat x `seq` (seqList strat xs)


-- Sequentially applies a strategy to the first  n elements of a list
seqListN :: (Integral a) => a -> Strategy b -> Strategy [b]
seqListN n strat []     = ()
seqListN 0 strat xs     = ()
seqListN n strat (x:xs) = strat x `seq` (seqListN (n-1) strat xs)
```

Now we can use these predefined strategies to define a parallel map function

```
-- parMap applies a function to each element of the argument list in
-- parallel.  The result of the function is evaluated using 'strat'
parMap :: Strategy b -> (a -> b) -> [a] -> [b]
parMap strat f xs  = map f xs `using` parList strat
```

For bigger programs the programmer only has to define strategies on the most important data structures in order to introduce parallelism in to his program. With this approach the parallel program will not be cluttered with `par` annotations and it's easier to determine semantics and dynamic behaviour of the individual functions.

6.6 Experiences with Strategies

[Strategies are cool]

7 Visualisation Tools

The visualisation tools that come together with GranSim take a GranSim profile as input and create level 2 PostScript files showing either activity or the granularity of the execution. A collection of scripts in the GranSim Toolbox allows to focus on specific aspects of the execution like the 'node flow' i.e. the movement of nodes (closures) between the PEs.

Most of these tools are implemented as Perl scripts. This means that they are very versatile and it should be easy to modify them. However, processing a large GranSim profile can involve a quite big amount of computation (and of memory). Speeding up crucial parts of the scripts is on my ToDo-list but better don't hold your breath.

7.1 Activity Profiles

Three tools allow to show the activity during the execution in three levels of detail:

The *overall activity profile* (created with '`gr2ps`') shows the activity of the whole machine (see Section 7.1.1 [Overall Activity Profile], page 40).

The *per-processor activity profile* (created with '`gr2pe`') shows the activity of all simulated processors (see Section 7.1.2 [Per-Processor Activity Profile], page 42).

The *per-thread activity profile* (created with '`gr2ap`') shows the activity of all generated threads (see Section 7.1.3 [Per-Thread Activity Profile], page 43).

All tools discussed in this section print a help message when called with the option `-h`. This message shows the available options. In general, all tools understand the option `-o <file>` for specifying the output file and `-m` for generating a monochrome profile (by default the tools generate colour PostScript).

The '`gr2ps`' and '`gr2ap`' scripts work in two stages:

1. First the '`.gr`' file is translated into a '`.qp`' file, which is basically a stripped down version of a GranSim profile.
2. Then the '`.qp`' file is translated into a '`.ps`' file.

The '`gr2pe`' script works directly on the '`.gr`' file.

7.1.1 Overall Activity Profile

The *overall activity profile* (created via `gr2ps`) shows the activity of the whole machine by separating the threads into up to five different groups. These groups describe the number of

- running threads (i.e. threads that are currently performing a reduction),
- runnable threads (i.e. threads that could be executed but that have not found an idle PE),
- blocked threads (i.e. threads that wait for a result that is being computed by another thread),
- fetching threads (i.e. threads that are currently fetching data from a remote PE),
- migrating threads (i.e. threads that are currently being transferred from a busy PE to an idle PE).

If the GranSim profile includes information about sparks (`-bs` option) it is also possible to show the number of sparks. However, this number is usually much bigger than the number of all threads. Therefore, it doesn't make much sense mixing the groups for sparks and threads in one profile.

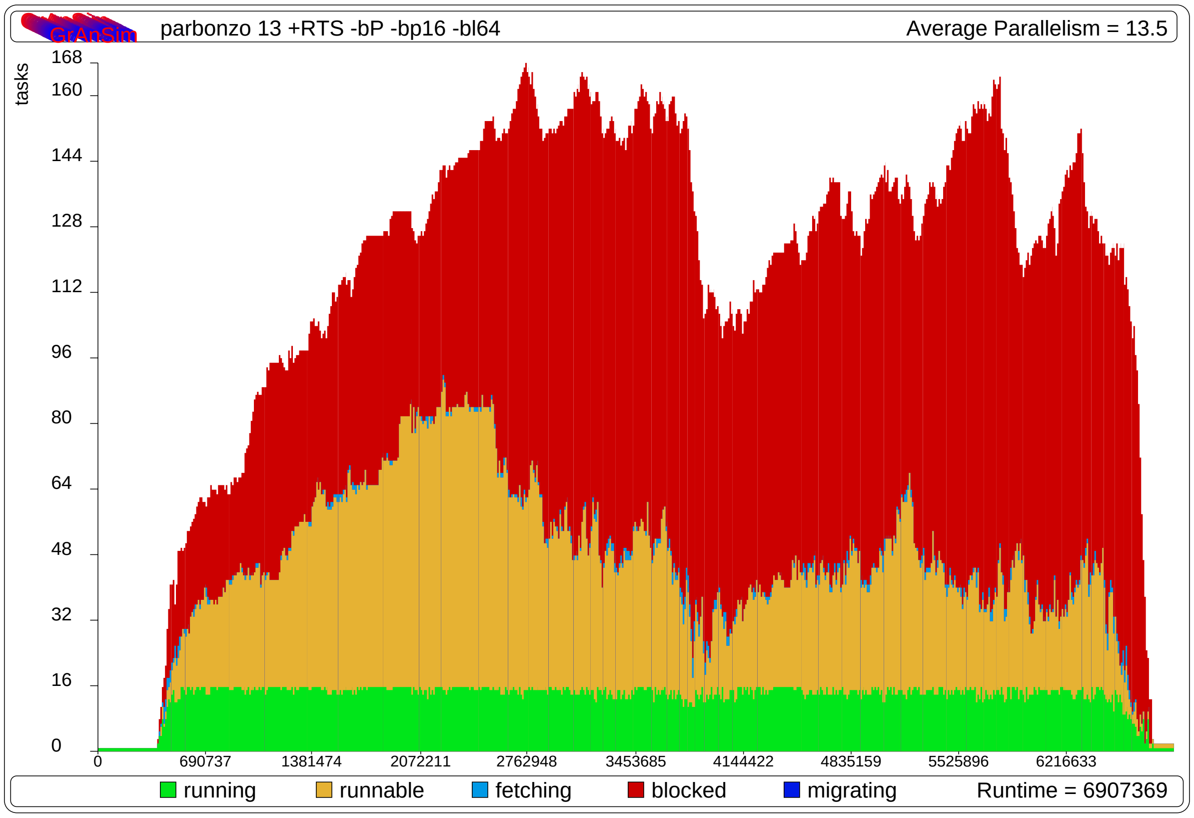
The option `-O` reduces the size of the generated PostScript file. The option `-I <string>` is used to specify which groups of threads should be shown in which order. For example

```
gr2ps -O -I "arb" parfib.gr
```

generates a profile '`parfib.ps`' showing only active ('a'), runnable ('r') and blocked ('b') threads. The letter code for the other groups are 'f' for fetching, 'm' for migrating and 's' for sparks.

In the current version the marks on the y-axis of the generated profile may be stretched or compressed. This might happen if many events occur at exactly the same time. If this is the case, the initial count of the maximal number of y-values may be wrong causing a rescaling at the very end. In practice that happens rarely (more often for GranSim-Light profiles, though).

The picture below shows an overall activity profile for a simple parallel divide-and-conquer program. The header of the graph shows the runtime-system options for the execution.



The overall runtime is measured in machine cycles. The average parallelism is the area covered by the running (green or medium-gray) threads, normalised with respect to the total runtime. In this graph only three groups show up: Most of the time 16 threads are running, utilising all available processors with occasional dips in the green area. The big amber (or light-gray) area of runnable threads indicates that this program can easily use all available processors. The large amount of blocking indicated by the large red (or black) area in the graph is caused by nodes near the root in the computation tree. They have to wait for the results of their children to combine them into the overall result. The sequential part at the beginning of the computation is due to I/O overhead including the initialisation of the basic I/O monad. Towards the end of the computation, when combining results near the root of the computation tree, the overall utilisation drops significantly. In this setup the latency is so small that no large areas of fetching (blue) threads appear. Also migration is turned of in this case (it can be turned on with the runtime system option `-bM`).

7.1.2 Per-processor Activity Profile

The idea of the *per-processor activity profile* is to show the most important pieces of information about each processor in one graph. Therefore, it is easy to compare the behaviour of the different processors and to spot imbalances in the computation.

This profile shows one strip for each of the simulated processors. Each of these strips encodes three kinds of information:

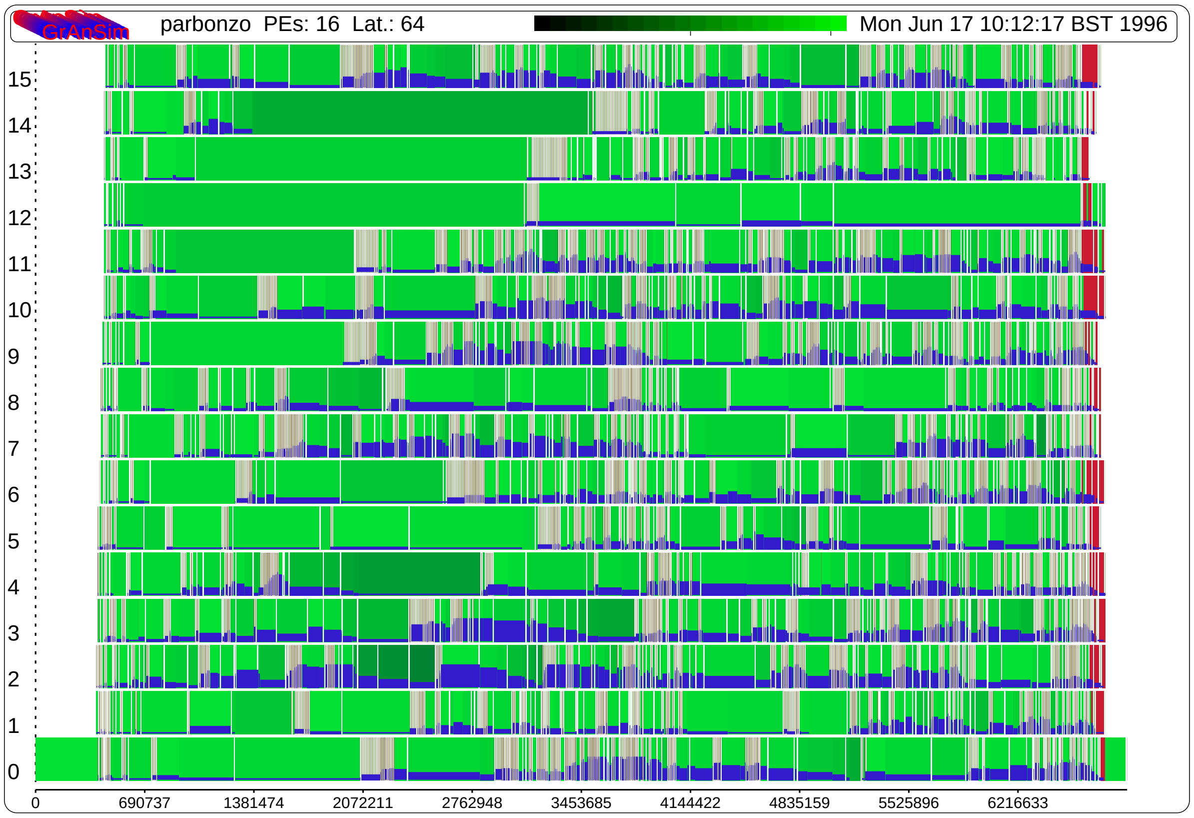
- Is the processor *active* at a certain point? If it is active the strip appears in some shade of green (gray in the monochrome version). If it is idle it appears in red (white in the monochrome version). The area before starting the first thread and after terminating the last thread is left blank in both versions.
- How high is the *load* of the processor? The load is measured by the number of runnable threads on this processor. A high load is shown by a dark shading of green (or grey). A palette at the top of the graph shows the available shadings (two ticks indicate the range that is used in the graph). It is possible to distinguish between 20 different values. Therefore, all processors with more than 20 runnable threads are shown in the same (dark) colour.
- How many *blocked threads* are on the processor? This information is shown by the thickness of blue (black) bar at to bottom of each strip. Without any blocked threads no bar is shown. If 20 or more threads are blocked the bar covers 80% of strip. Thus, the load information is always visible 'in the background'.

This script also allows to produce variants of the same kind of graph that focus on different features of GranSim:

- *Migration:* With the option `-M` this script produces a graph that draws arrows between processors indicating the migration of a thread from one processor to another. No load or blocking information is shown in this graph.
- *Sparking:* With the option `-S` a spark graph is generated. It shows information about the number of sparks on a processor in the same way as the the number of runnable threads (i.e. by shading). This graph is useful to highlight processors that create a lot of work.

No more than about 32 processors should be shown in one graph otherwise the strips are getting too small. This profile can not be generated for a GranSim-Light profile.

The graph below shows a per-processor activity profile for the same parallel divide-and-conquer program as in the previous section.



The graph shows the activity of each of the 16 processors in this simulation. The dark green areas in the first third of the computation show that processors 2, 4 and 14 have a significantly higher load of runnable threads than the rest. Also note the pattern that a decreasing number of blocked threads (thinner blue bar) is accompanied by an increasing number of runnable threads (darker green area).

7.1.3 Per-Thread Activity Profile

The *per-thread activity profile* shows the activity of all generated threads. For each thread a horizontal line is shown. The line starts when the thread is created and ends when it is terminated. The thickness of the line indicates the state of the thread. The possible states correspond to the groups shown in the overall activity profile (see Section 7.1.1 [Overall Activity Profile], page 40).

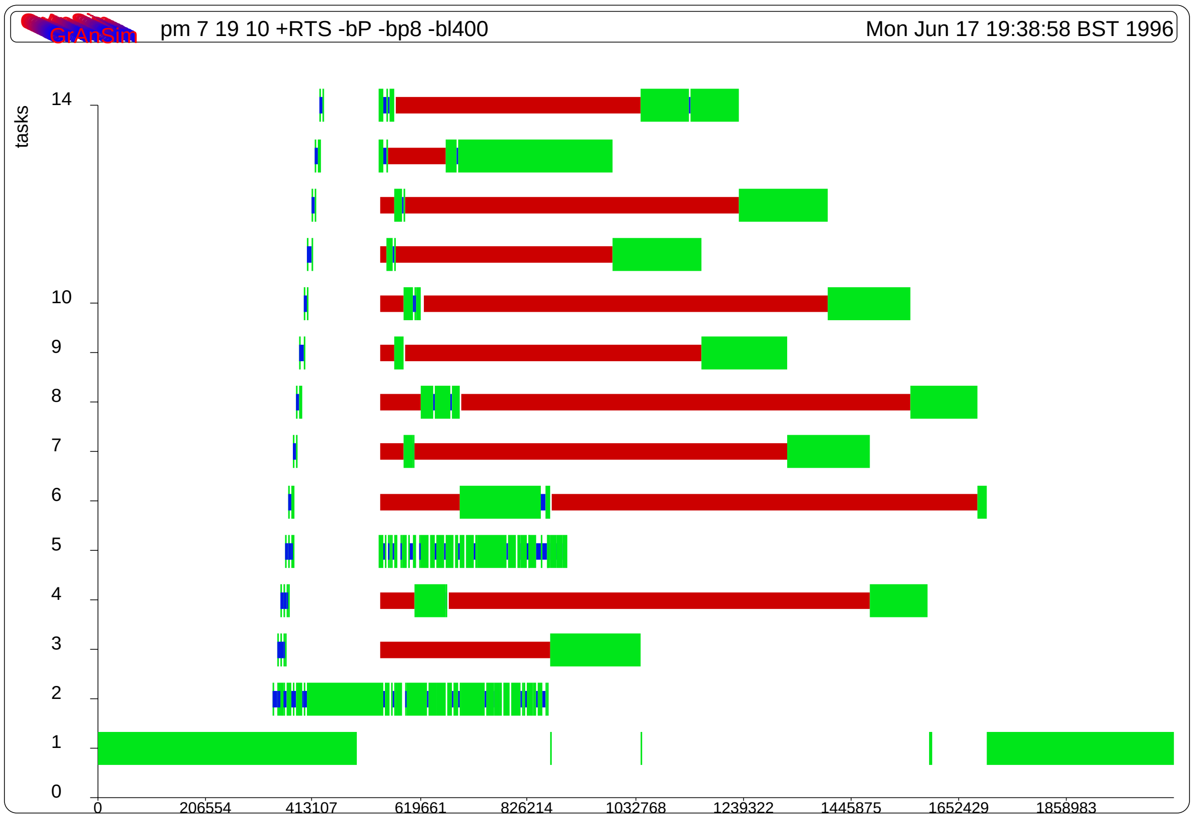
The states are encoded in the following way:

- A *running* thread is shown as a thick green (gray) line.
- A *runnable* thread is shown as a medium red (black) line.

- A *fetching* (or *migrating*) thread is shown as a thin blue (black) line.
- A *blocked* thread is shown as a gap in the line.

This profile gives the most accurate kind of information and it often allows to 'step through' the computation by relating events on different processors with each other. For example the typical pattern at the beginning of the computation is some short computation for starting the thread followed by fetching remote data. After that the thread may become runnable if another thread has been started in the meantime.

The picture below shows an example of a per-thread activity profile. Note the short period of fetching immediately after starting a thread in order to get the data for the spark that has just been turned into a thread. The high degree of suspension is mainly due to the fact that migration is turned off in this example.



However, such a detailed analysis is only possible for programs with a rather small number of threads. Usually, GranSim profiles of bigger executions have to be pre-processed to reduce the number of threads that are shown on one graph (see Section 7.3 [Scripts], page 49). As the level of detail provided by this graph is rarely needed for bigger executions no automatic splitting of a profile into several graphs has been implemented.

7.2 Granularity Profiles

The tools for generating *granularity profiles* aim at showing the relative sizes of the generated threads. Especially the number of tiny threads, for which the overhead of thread creation is relatively high is of interest.

All tools discussed in this section require Gnuplot to generate the granularity profiles. I am using version 3.5 but it should work with older versions, too.

For showing granularity basically two kinds of graphs can be generated:

- A *bucket graph* (see Section 7.2.1 [Bucket Graphs], page 46), which collects threads with similar runtime in the same bucket and shows the number of threads in each bucket.
- A *cumulative graph* (see Section 7.2.2 [Cumulative Graphs], page 47), which shows how many threads have a runtime less than or equal a given number..

The main tools for generating such graphs are:

- `gr2gran` creates one bucket graph and one cumulative graph from a given GranSim profile. The information about the partitioning for the bucket statistics and other set-up information is usually provided in a *template file* (see Section 7.2.3 [Template Files], page 48), which is specified via the `-t` option (`-t ,` uses the global template file in `$GRANDIR/bin`).

  This script works in three stages:

  1. First a '`RTS`' file is generated, which is only a sorted list of runtimes extracted out of the `END` events of a GranSim profile (see Chapter 8 [GranSim Profiles], page 50).
  2. The main stage generates a '`gnuplot`' file by grouping the threads into buckets and computing cumulative values.
  3. Finally, Gnuplot is used to generate '`PostScript`' files showing the graphs.
- `gran-extr` is based on the same idea as `gr2gran`, but it produces even more graphs, showing the communication percentage, determining a correlations coefficient between heap allocations and runtime etc.

7.2.1 Bucket Graphs

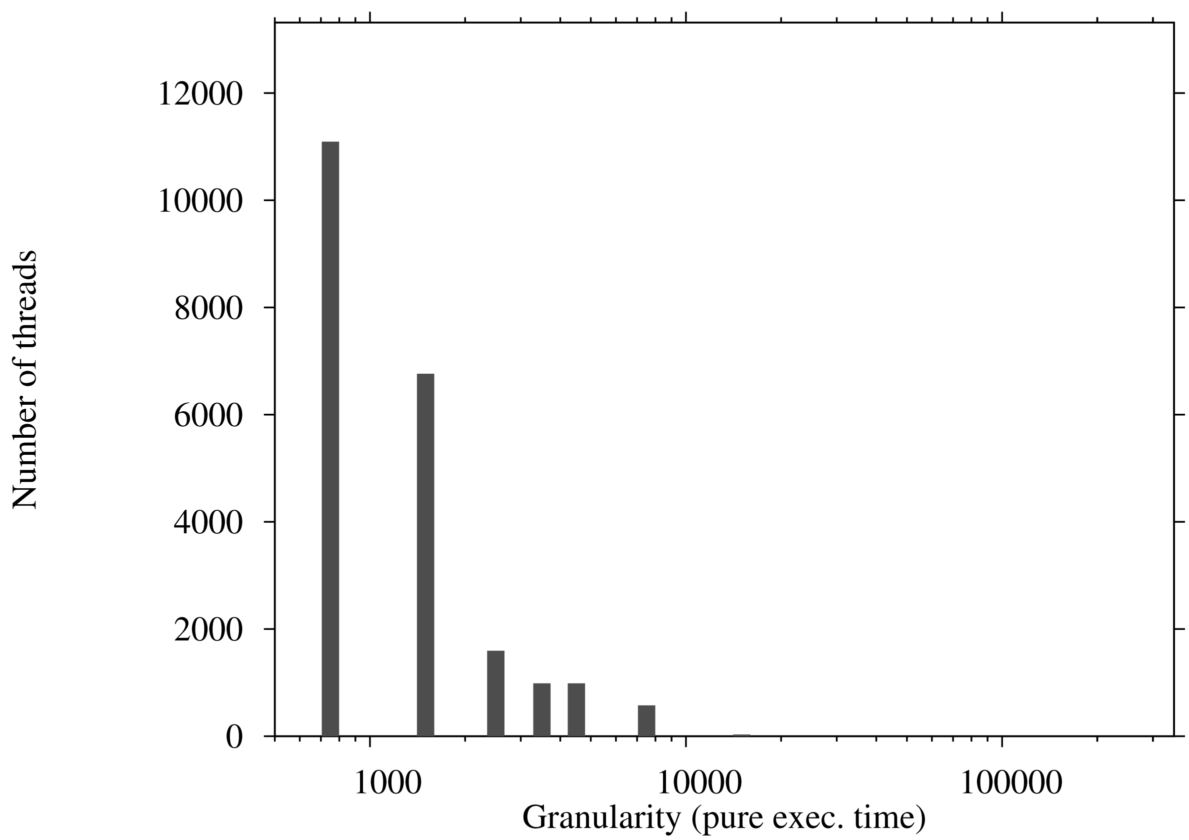
In a bucket graph the x-axis indicating execution times of the threads is partitioned into intervals (dfnbuckets). The graph shows a histogram of the number of threads in each bucket (i.e. whose execution time falls into this interval). For generating this kind of graph only a restricted GranSim profile (containing only `END` events) is required.

For example one of the files generated by running

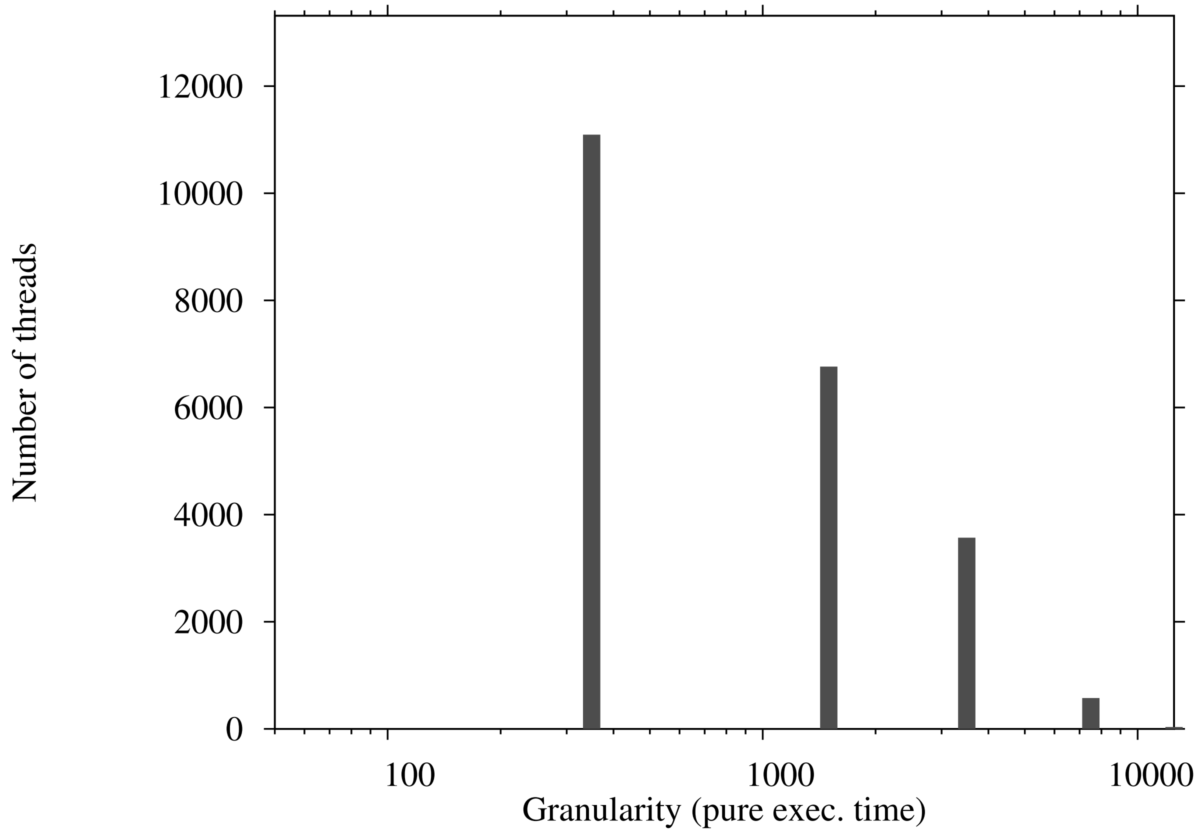
```
gr2gran -t , pf.gr
```

is `g.ps`, which contains such a bucket statistics. The `-t` option of this tool selects the right template file (`,` is a shorthand for the global template in `$GRANDIR/bin`).

Here is the bucket statistics of executing `parfib 22`:



It shows that this program creates a huge number of tiny threads (note the log scale in the graph). Refining the intervals for these tiny threads further gives the following bucket statistics



The necessary change in the template file for this bucket statistics is

```
-- Intervals for pure exec. times
G: (100, 200, 500, 1000, 2000, 5000, 10000)
```

7.2.2 Cumulative Graphs

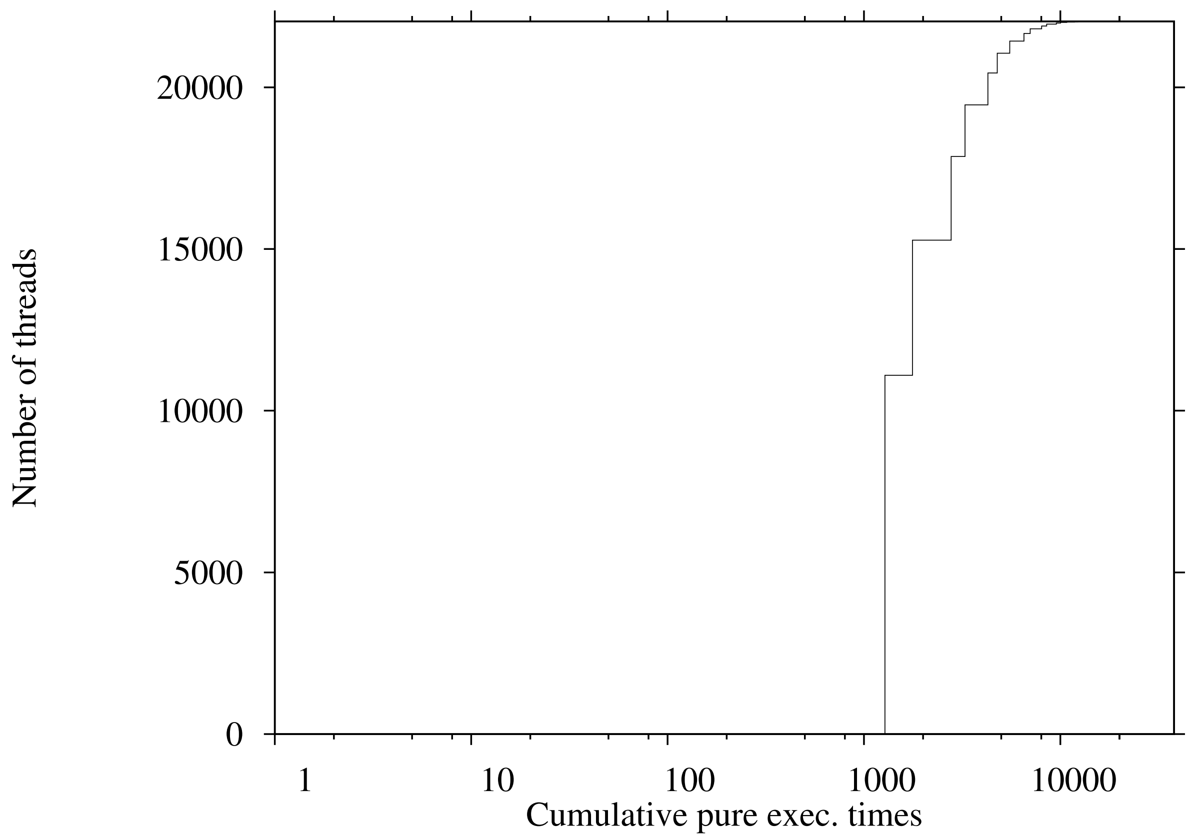
In a cumulative graph the x-axis again represents execution times of the individual threads. The value in the graph at the time `t` represents the number of threads whose execution time is smaller than `t`. Therefore, the values in the graph are monotonically increasing until the right end shows the total number of threads in the execution.

Again, running

```
gr2gran -t , pf.gr
```

generates cumulative graphs for the runtime in the files `cumu-rts.ps` and `cumu-rts0.ps` (one file shows absolute numbers of threads, the other the percentage of the threads on the y-axis).

Here is the cumulative runtime statistics of executing `parfib 22`:



7.2.3 Template Files

The functions for reading template files can be found in '`template.pl`'. This file also contains documentation about the available fields.

7.2.4 Statistics Packages

A set of statistics functions for computing mean value, standard deviation, correlation coefficient etc can be found in 'stats.pl'.

7.3 Scripts

The GranSim Toolbox contains not only visualisation tools but also a set of scripts that work on GranSim profiles and provide specific information.

- The `tf` script aims at showing the task flow (as well as node flow) in the execution of a program. It is used in the GranSim Emacs mode to narrow a GranSim profile (see Section 8.3 [The Emacs GranSim Profile Mode], page 55).
- The `SN` script creates a summary of spark names that occur in a GranSim profile. This summary is shown as a impulses graph via Gnuplot. It allows to compare the relative number of threads generated by each static spark site.
- The `AVG` and `avg-RTS` scripts compute the average runtime from an `RTS` file, which is generated by 'gr2RTS'.

8 GranSim Profiles

This chapter describes the contents of a *GranSim profile* (a '.gr' file). In most cases the profiles generated by the visualisation tools should provide sufficient information for tuning the performance of the program. However, it is possible to extract more information out of the generated GranSim profile. This chapter provides information how to do that.

8.1 Types of GranSim Profiles

Depending on some runtime-system options different kind of profiles are generated:

- A *reduced* GranSim profile contains in the body component only `END` events. This is sufficient to extract granularity profiles, but it is not sufficient to generate activity profiles. This is the default setting for GranSim profiles.
- A *full* GranSim profile contains one line for every major event in the system (see ⟨undefined⟩ [Contents of a Granularity Profile], page ⟨undefined⟩). A generation of such a profile is enabled by the RTS option `-bP`.
- A *spark* profile additionally contains events related to sparks (for creating, using, pruning, exporting, acquiring sparks). Such a profile is generated when using the RTS option `-bs`.
- A *heap* profile additionally contains events for allocating heap. Such a profile is generated when using the RTS option `-bh`.

8.2 Contents of a GranSim Profile

This section describes the syntactic structure of a GranSim profile.

8.2.1 Header

The header contains general information about the execution. It is split into several sections separated by lines only consisting of `-` symbols. The end of the header is indicated by a line only consisting of `+` symbols.

The sections of the header are:

- Name of the program, arguments and start time.
- General parameters describing the parallel architecture. This covers the number of processors, flags for thread migration, asynchronous communication etc. Finally, this section describes basic costs of the parallel machine like thread creation time, context switch time etc.
- Communication parameters describing basic costs for sending messages like latency, message creation costs etc.
- Instruction costs describing the costs of different classes of machine operations.

8.2.2 Body

The body of the GranSim profile contains events that are generated during the execution of the program. The following subsections first describe the general structure of the events and then go into details of several classes of events.

8.2.2.1 General Structure of an Event

Each line in the body of a GranSim profile represents one event during the execution of the program. The general structure of one such line is:

- The keyword `PE`.
- The *processor number* where the event happened.
- The *time stamp* of the event (in square brackets).
- The *name of the event*.
- The *thread id* of the affected thread (a hex number).
- Optionally a *node* as an additional argument to the event (e.g. the node to be reduced in case of a `START` event). This is either a hex number or the special string `______` indicating a Nil-closure.
- Additional information depending on the event. This can be the processor from which data is fetched or the length of the spark queue after starting a new thread.

The fields are separated by whitespace. A : symbol must follow the time stamp (which must be in sqare brackets).

8.2.2.2 END Events

`END` events are an exception to this general structure. The reason for their special structure is that they summarise the most important information about the thread. Therefore, information about e.g. the granularity of the threads can be extracted out of `END` events alone without having to generate a full GranSim profile.

The structure of an `END` event is:

- The keyword `PE`.
- The *processor number* where the event happened.
- The *time stamp* of the event (in square brackets).
- The *name of the event* (`END` in this case).
- The keyword `SN` followed by the *spark name* of the thread. This information allows to associate a thread with its static spark site in the program (See Chapter 4 [Parallelism Annotations], page 14, on how to give names to spark sites.)
- The keyword `ST` followed by the *start time* of the thread.
- The keyword `EXP` followed by a flag indicating whether this thread has been *exported* to another processor or has been evaluated locally (possible values are `T` and `F`).
- The keyword `BB` followed by the number of *basic blocks* that have been executed by this thread.
- The keyword `HA` followed by the number of *heap allocations*.
- The keyword `RT` followed by the total *runtime*. This is the most important information in an `END` event. It is used by the visualisation tools for generating granularity profiles.
- The keyword `BT` followed by the total *block time* and a block count (i.e. how often the thread has been blocked).
- The keyword `FT` followed by the total *fetch time* and a fetch count (i.e. how often the thread fetched remote data).
- The keyword `LS` followed by the *number of local sparks* (sparks that have to be executed on the local processor) generated by the thread.
- The keyword `GS` followed by the *number of global sparks* generated by the thread.
- The keyword `EXP` followed by a flag indicating whether this thread was *mandatory* or only advisable (in the current version this flag is not used; it would be important in a combination of GranSim with a concurrent set-up).

8.2.2.3 Basic Thread Events

The main events directly related to threads are:

- `START`: Generated when starting a thread (after adding overhead for thread creation to the clock of the current processor). After the thread id it has two additional fields: one specifying the node to be evaluated (as a hex number) and the spark site that generated this thread (format: `[SN` *n*`]` where *n* is a dec number).
- `START(Q)`: Same as `START` but the new thread is put into the runnable queue rather than being executed (only if the current processor is busy at that point).
- `BLOCK`: A thread is blocked on data that is under evaluation by another thread. It is descheduled and put into the blocking queue of that node. Two additional fields contain the node on which the thread is blocked and the processor on which this node is lying (format: `(from` *n*`)` where *n* is a processor (dec) number).
- `RESUME`: Continue to execute the thread after it has been blocked or has been waiting for remote data. This event does not contain additional fields.
- `RESUME(Q)`: Same as `RESUME` but the new thread is put into the runnable queue rather than being executed (only if the current processor is busy at that point).
- `SCHEDULE`: The thread is scheduled on the given processor (no additional fields). This event is usually emitted after terminating a thread on the processor. It may also occur after a `FETCH` (if asynchronous communication is turned on) or after a `BLOCK` event.
- `DESCHEDULE`: The thread is descheduled on the given processor (no additional fields). After this event the thread is in the runnable queue. This event is not used for implicit descheduling that is performed after events like `BLOCK` or `FETCH`. `DESCHEDULE` events should only occur if fair scheduling is turned on.

8.2.2.4 Communication Events

Events that are issued when sending data between processors are:

- `FETCH`: Send a fetch request from the given thread (on the given processor) to another processor. This event has two additional fields: The first field is the node (hex number) that should be fetched. The last field is the processor where this node is lying and from which the data has to be fetched (format: `(from` *n*`)` where *n* is a processor (dec) number).
- `REPLY`: A reply for a fetch request of the given thread arrived at the given processor. The first additional field contains the node and the last field contains the processor from which it arrived (format: `(from` *n*`)` where *n* is a processor (dec) number). Note: This event only marks the arrival of the data. It is usually followed by a `RESUME` or `RESUME(Q)` event for the thread that asked for the data.

8.2.2.5 Thread Migration Events

These events are only produced when thread migration is enabled (`-bM`):

- `STEALING`: Indicates the stealing of a thread on the given processor. The thread which is being stolen appears in the thread field. One additional field (the last field) indicates which processor is stealing that thread (format: `(by` *n*`)` where *n* is a processor (dec) number).
- `STOLEN`: Indicates the arrival of a stolen thread on the given processor. Two additional fields show the node which will be evaluated by this thread next. The last field shows from which processor the thread has been stolen (format: `(from` *n*`)` where *n* is a processor (dec) number). Note: This thread is immediately being executed by the given processor (no `RESUME` event follows).
- `STOLEN(Q)`: Same as `STOLEN` but the new thread is put into the runnable queue rather than being executed (only if the current processor is busy at that point).

8.2.2.6 Spark Events

When enabling spark profiling, events related to sparks will appear in the profile:

- `SPARK`: Indicates the generation of a spark on the given processor for the given node. At that point it is added to this processor's spark pool. Two additional fields show the node to which this spark is pointing and the current size of the spark pool (format: `[sparks` *n*`]` where *n* is a dec number).
- `SPARKAT`: Same as `SPARK` but with explicit placement of the spark on this processor. This is usually achieved in the program by using a `parLocal` or `parAt` rather than a `parGlobal` annotation (see Chapter 4 [Parallelism Annotations], page 14).
- `USED`: Indicates that this spark is turned into a thread on the given processor. A `START` or `START(Q)` event will follow soon afterwards.
- `PRUNED`: A spark is removed from the spark pool of the given processor. This might occur when the spark points to a normal form (there is no work to do for that spark). This is checked when creating a spark and when searching the spark pool for new work.
- `EXPORTED`: A spark is exported from a given processor. Two additional fields show the node to which this spark is pointing and the current size of the spark pool (format: `[sparks` *n*`]` where *n* is a dec number).
- `ACQUIRED`: A spark that has been exported by another proceeessor is acquired by the given processor. Two additional fields have the same meaning as for `EXPORTED`.

8.2.2.7 Debugging Events

Certain debug options generate additional events that allow to monitor the internal behaviour of the simulator. This information shouldn't be of interest for the friendly user but might come in handy for those who dare hacking at the runtime-system:

- `SYSTEM_START`: Indicates that the simulator is executing a "system" routine (a routine in the runtime-system that is not directly related to graph reduction). This allows to show when exactly rescheduling is done in the simulator. It may be useful in GranSim-Light to check that the costs during system operations are attached to the right thread.
- `SYSTEM_END`: See previous event. From this point on normal graph reduction is performed.

8.3 The Emacs GranSim Profile Mode

Looking up information directly in a GranSim profile is very tedious (believe me, I have done it quite often). To make this task easier the GranSim Toolbox contains a GNU Emacs mode for GranSim profiles: the *GranSim Profile Mode*.

The most useful features (IMNSHO) are highlighting of parts of a GranSim profile and narrowing of the profile to specific PEs, threads, events etc.

8.3.1 Installation

To use this mode just put the file '`GrAnSim.el`' somewhere on your Emacs *load-path* and load the file. I don't have autoload support at the moment, but the file is very short anyway, so directly loading it is quite fast. Currently, the mode requires the *hilit19* package for highlighting parts of the profile. It also requires the '`tf`' script in the bin dir of your GranSim installation.

I use Emacs 19.31 with the default '`hilit19.el`' package, but the GranSim profile mode has been successfully tested with Emacs 19.27. However, if you have problems with the mode please report it to the address shown at the end of this document (see Chapter 13 [Bug Reports], page 67).

8.3.2 Customisation

A few Emacs variables control the behaviour of the GranSim Profile mode:

gransim-auto-hilit Variable

This variable indicates whether highlighting is turned on by default. Note that you can customise '`hilit19`' such that it does not automatically highlight buffers that are bigger than a given size. Since GranSim profiles tend to be extremely large you might want to reduce the default value.

grandir Variable

The root of the GranSim installation. The mode searches for scripts of the GranSim Toolbox in the directory grandir/bin. By default this variable is set to the contents of the environment variable `GRANDIR`.

hwl-hi-node-face Variable

Face to be used for specific highlighting of a node.

hwl-hi-thread-face Variable

Face to be used for specific highlighting of a thread.

Here are the hilit19 variables that are of some interest for the GranSim Profile Mode:

hilit-auto-highlight Variable

T if we should highlight all buffers as we find 'em, nil to disable automatic highlighting by the find-file hook. Default value: `t`.

hilit-auto-highlight-maxout Variable

Auto-highlight is disabled in buffers larger than this. Default value: 60000.

8.3.3 Features

The main features of the GranSim profile mode are:

- *Highlighting* of parts of the profile. Colour coding is used to distinguish between events that start a reduction, finish a reduction and block a reduction. Within `END` events the total runtime is specially highlighted.
- *Narrowing* of the profile. This should not be confused with the narrowing mode in Emacs. The narrowing in GranSim profile mode is done by running a script ('`tf`') over the buffer and

displaying the output in another buffer. Hence, narrowing can be further refined be improving the '`tf`' script, which is written in Perl.

It is possible to narrow a GranSim profile to a specific

- *processor* (PE),
- *event*,
- *thread*,
- *node*,
- *spark* (only possible for spark profiles)

This feature is particularly useful to e.g. follow a node, which has been moved between processors or to concentrate on the reductions on one specific processor.

Of course, for those pagans, who don't believe in Emacs it is also possible to run the '`tf`' script directly on a '`.gr`' file.

- A second form of *highlighting*, specialised for nodes and threads is available, too. With the commands `hwl-hi-thread` and `hwl-hi-node` every occurrence of the thread or node in the profile after the current point is highlighted. The function `hwl-hi-clear` undoes all such highlighting.
- There is a menu item for calling most of the functions described here. It automatically appears in any GranSim profile (i.e. any file that has a '`.gr`' extension).

Default key bindings in GranSim profile mode:

`C-c t, M-x hwl-truncate`
: Truncate event lines such that exactly one line is shown for one event in the body of a profile.

`C-c w, M-x hwl-wrap`
: Wrap lines to show them in full length.

`C-c , M-x hwl-toggle-truncate-wrap`
: Toggle between the above two modes.

`C-c h, M-x hilit-rehighlight-buffer`
: Rehighlight the whole buffer.

`C-c p, M-x hwl-narrow-to-pe`
: Narrow the profile to a PE.

`C-c t, M-x hwl-narrow-to-thread`
: Narrow the profile to a thread.

`C-c e, M-x hwl-narrow-to-event`
: Narrow the profile to an event.

`C-c C-e, (lambda () (hwl-narrow-to-event "END"))`
: Narrow the profile to an `END` event.

`C-c , M-x hwl-toggle-truncate-wrap`
: Toggle between the above two modes.

`C-c N, M-x hwl-hi-node`
: Highlight a node in the profile.

`C-c T, M-x hwl-hi-thread`
: Highlight a thread in the profile.

`C-c C-c, M-x hwl-hi-clear`
: Remove highlightings of nodes and threads.

9 The Parallel NoFib Suite

While developing GranSim (and GUM) we have started to collect a set of interesting and non-trivial parallel programs written in parallel Haskell. These programs are used as a test suite for GranSim and part of the NoFib Suite of GHC.

Currently this suite contains the following programs:

1. Sieve of Erathostenes (`sieve`), which creates a bidirectional pipeline, where each processor filters the multiples of one prime number out of an input list.
2. Warshall's shortest path algorithm in a graph (`warshall`), which creates a cyclic pipeline of processors.
3. A function `matmult`, which performs a parallel matrix multiplication.
4. A function `determinant`, which computes for a given square matrix its determinant.
5. A function `LinSolv`, which solves a given set of linear equations over integers by using multiple homomorphic images, computing the result in each image via Cramer's Rule. The main part of this algorithm is the parallel determinant computation.
6. A function `coins`, which computes the number of possibilities how to pay a given value with a given set of coins.
7. An award assignment program, that basically has to compute permutations of a given list (`pperms`).
8. A parallel verion of Quicksort (`sort`).
9. A function word search program `soda7`, which searches a given grid of letters for a given set of words.
10. An RSA encryption program `rsa` (taken out of the sequential nofib suite).
11. An n-queens program `queens`.
12. The parallel database manager for GUM `dcbm`. It performs queries to a database in parallel.
13. The bill of materials program `bom` developed (under GranSim) by Phil Trinder as part of the Parallel Databases Project.
14. A ray-tracer `nray` based on a version taken from Kelly's thesis and ported to GRIP by Hammond.
15. A univariant resultant computation, using the SACLIB library for computer algebra. Mainly the basic polynomial arithmetic has been taken form that library.
16. One of the FLARE programs, solving a problem in the area of computational fluid dynamics `cfd`.
17. A function `minimax`, which computes for a given position in the game of tic-tac-toe the best next move using an Alpha-Beta pruning algorithm.

18. Several NESL programs, including a numerical integration algorithm `integrate`, a quick hull algorithm `qhull` for computing the convex hull of a set of points in a plane, a matrix inversion based on gauss-jordan elimination `mi` and a fast fourier transformation `fft`. All algorithms are based on the NESL versions published by Guy Blelloch in CACM 39(3) and translated to Haskell.
19. A Newton-Raphson iteration for finding a root of a polynomial. This program has been taken from the Impala suite of implicitly parallel benchmark programs (written in Id).

Despite its name the parallel nofib suite also contains some fib-ish programs. These programs should be of interest for getting a start with parallel functional programming using GranSim:

1. A parallel factorial function (`parfact`), which computes the sum of all numbers from 1 up to a given value n by bisecting the interval and computing results of the intervals in parallel.
2. A parallel fib-like function (`parfib`), which performs an additional gcd computation and 2 additional multiplications in each recursive call.

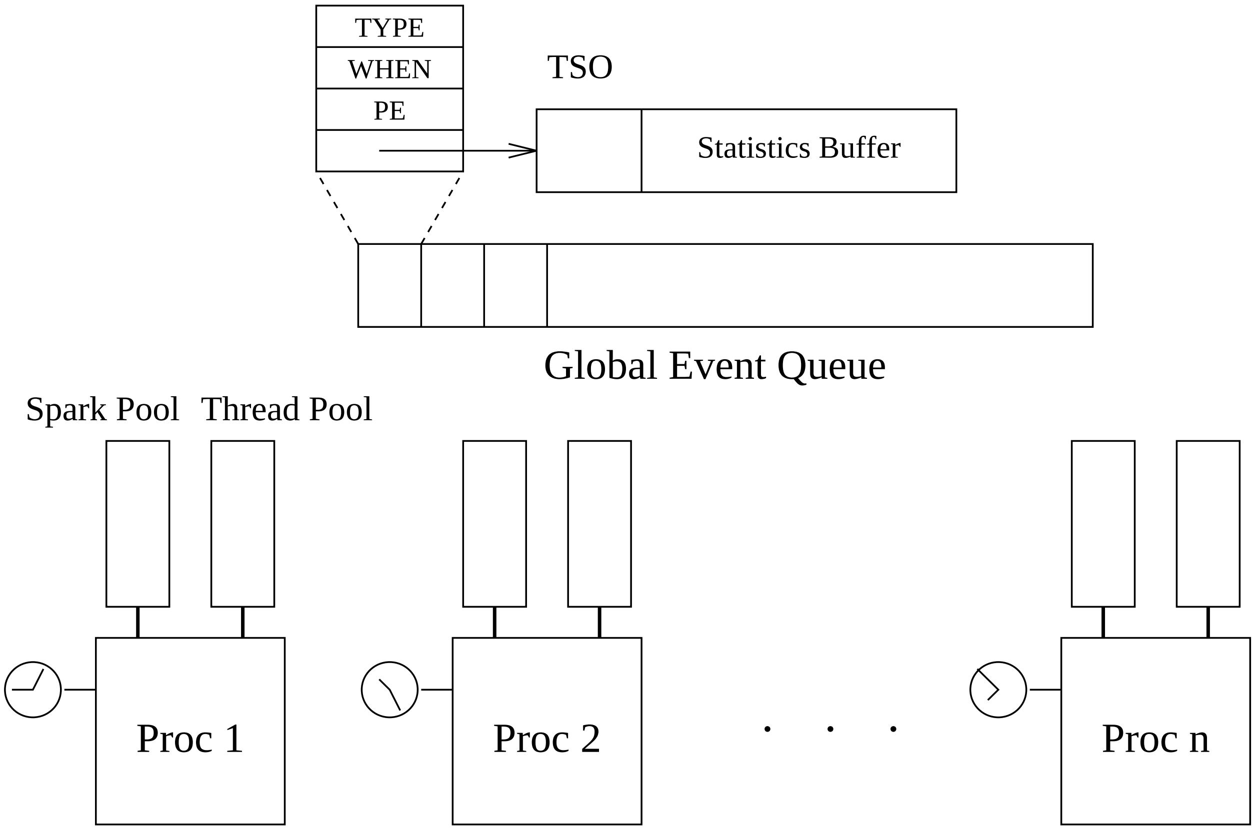
10 Internals

This chapter discusses details of the implementation of GranSim.

[This text is taken from a draft of a previous paper. It needs updates to cover the more recent changes like adding GranSim-Light. See also the HPFC'95 paper for the basics]

10.1 Global Structure

The following picture depicts the global structure of GranSim



GranSim performs an event driven simulation with a centralised event queue. Each of the simulated processors has its own spark queue and thread queue as well as its own clock. The synchronisation of the clocks is performed via accessing the event queue which is sorted by the time stamps of the events.

10.2 Accuracy

GranSim is built on top and therefore makes use of a state-of-the-art compiler for Haskell including a huge variety of possible optimisations that can be performed. The instruction count function, which determines the number of instructions that are executed for a given piece of abstract C code, has been carefully tuned by analysing the assembler code generated by GHC and the results have been compared with the number of instructions executed in real Haskell programs. These comparisons have shown that the instruction count of the simulation lies within 10% for arithmetic operations, within 2% for load, store operations, within 20% for branch instructions and within 14% for floating point instructions of the real values.

To allow the modelling of different kinds of architectures the instructions have been split into five classes, with different weights: arithmetic operations, floating point operations (1 cycle each), load, store operations (4 cycles each) and branch instructions (2 cycles). These weights model a SPARC processor and have been verified with Haskell programs in the nofib suite. The weights are tunable in order to simulate other kinds of processors.

10.3 Flexibility

GranSim has a big number of tunable parameters (see ⟨undefined⟩ [RTS Options], page ⟨undefined⟩). One set of parameters allows to model a broad variety of processors. To achieve a similar accuracy as for SPARC processors the same kind of measurements can be performed. Another set of parameters allows to tune the costs for communication latency, message pack time etc. Finally, the overhead imposed by the runtime-system can be captured by adjusting the parameters for task creation time, context switch time etc. This allows us to model architectures ranging from shared memory to distributed memory machines. And we can quite easily simulate the effect of some small changes in the runtime system on the overall behaviour.

Additionally, the GranSim-Light setup allows to study the parallelism inherent in a program rather than choosing a fixed architecture to run the program on. Our experiences with parallelising rather big programs show that this is an important additional feature for tuning the performance of a parallel program.

10.4 Visualisation

Especially for analysing the granularity of the created tasks, we need visualisation tools. We have developed a set of tools for that purpose and used it on a quite wide range of example programs to analyse their performance (see Chapter 9 [Parallel NoFib Suite], page 59). With these tools we were able to show a very high correlation between execution time and the amount of allocated heap space. Apart from these tools showing the granularity of the resulting tasks, we have also developed a set of tools for showing the activity of the simulated machine as a whole, of all processors and of all tasks in the system. These tools have proven indispensable in the parallelisation and optimisation of a linear system solver. Based on the experience from this implementation, such tools are essential when working with a lazy language, in which the order of evaluation is not at all obvious from the program source. See Chapter 7 [Visualisation Tools], page 39 for a more detailed discussion of these tools.

10.5 Efficiency

The high accuracy in the simulation also implies a rather high overhead for the simulation as more bookkeeping has to be done. This is mainly due to the exact modelling of communication. Therefore, programs that perform a lot of communication will impose a higher overhead on the simulation. Early measurements of some example programs on GranSim showed a factor between 14 and more than 100 between a GranSim simulation and an optimised sequential execution. In the meantime we have spent some effort in improving the efficiency of the simulator. The most recent measurements show factors between 10 and 15. Regarding the amount of information we generate out of a simulation we think that is an acceptable factor. Especially a comparison with another simulator for annotated Haskell programs, HBCPP, is of interest. Using a set of small example programs GranSim was between 1.5 and 2 times slower than HBCPP. In some cases the GranSim-Light setup was about as fast the HBCPP version.

We observed one potential problem of the GranSim-Light setup, though. If the parallel program creates a huge number of parallel threads (several thousand) the bookkeeping of all these running threads slows down the simulation significantly. This might make the GranSim-Light setup slower than the plain GranSim setup, which has a limited number of running threads. In such a setup it is advisable to increase the time slice for each thread in the simulation (`-bw`*N* option).

10.6 Integration into GHC

GranSim is part of the overall GHC system. Therefore, it is possible to use all the features of a compilation via GHC for GranSim, too. Especially, the `ccall` mechanism can be used to call C functions in a parallel program (indeed, we have experimented with the use of a computer algebra library in implementing a parallel resultant algorithm). Also the interaction between certain optimisations like deforestation and the parallel execution of a program should be of interest (deforestation might eliminate intermediate lists that are crucial for the parallel execution of the program).

Finally, in parallelising the LOLITA natural language processing system it was crucial to have an profiler for the sequential version of the program available. This allowed us to determine the important parts of the computation early on in the parallelisation process.

11 GranSim vs GUM

[Mainly a placeholder for a real comparison]

In a nutshell: The GUM runtime-system is very close to GranSim. To a fairly large degree the same code is used by both versions. The main source of inaccuracy is the modelling of stealing sparks and threads. GUM uses a fishing mechanism: a fish message is sent from processor to processor, looking for sparks. In GranSim we use the global knowledge of the system and weigh the costs for stealing a spark with the inverse of the probability of hitting a processor with available sparks. Furthermore, there is no counterpart of GUM's FETCHME closures as pointers to remote data in GranSim.

On the other hand, GranSim is also significantly more powerful than GUM. It supports thread migration, offers different strategies for packing a graph (based on the number of thunks), and it allows to choose between different fetching strategies, when deciding what to do while one thread fetches remote data. These features mainly deal with the aspect of data locality in the program.

Another important aspect for the performance of the granularity of the parallel program. This can be tuned by choosing among three basic methods for granularity control (see Section 5.6 [Granularity Control Mechanisms], page 23).

12 Future Extensions

This version of GranSim should be fairly stable. It has been stress tested on a number of non-trivial programs (including a program of more than 100,000 lines of Haskell and wee bit of C). It contains most of the features I want to have in it.

One future extension of GranSim might be the implementation of a more accurate modelling of the spark/thread stealing mechanism (based on GUM's fishing model).

13 Bug Reports

Send all bug reports, including the source of the failing program, the compile and RTS options and the error message to

hwloidl@dcs.gla.ac.uk

Be sure to add "GranSim Bug" in the subject line.

Earlier versions of GranSim sometimes had problems when using the generational garbage collector (default). If you meet problems with GranSim you can try the following:

- Use the option `-Sstderr` to get garbage collection messages. This will tell you whether there is any garbage collection going on.
- Increase the heap size. This should give you a hint whether the problem is related to garbage collection at all.
- Try to use the runtime-system options `-F2s` to force two-space garbage collection. This garbage collector uses more heap than the generational one. Whenever I had problems with a test program before it just went away with this garbage collector.
- Perhaps also use `-Z` (this omits update frame squeezing; you don't have to worry about what that means, though).

In any case, please report it as a bug to the above address.

Appendix A Example Program: Determinant Computation

The example program in this chapter is a parallel determinant computation. It uses the data type `SqMatrix a` to represent a matrix as a list of list together with its bounds.

For getting a parallel version of the program strategies (see Section 6.5 [A Class of Strategies], page 35) are used. Thus, much of the parallelism comes from applying the `parList` strategy. In order to demonstrate the use of strategies in a parallel functional program this example contains parallelism down to a very fine granularity.

```
determinant :: (NFDataIntegral a) => SqMatrix a -> a

determinant (SqMatrixC ((iLo,jLo),(iHi,jHi)) mat)
  | jHi-jLo+1 == 1 =  let
                        [[mat_1_1]] = mat
                      in
                        mat_1_1
  | jHi-jLo+1 == 2 =  let
                        [[mat_1_1,mat_1_2],
                         [mat_2_1,mat_2_2] ] = mat
                        a = mat_1_1 * mat_2_2
                        b = mat_1_2 * mat_2_1
                        strategy r = a `par` b `par` ()
                      in
                        a - b `using` strategy
  | otherwise      =
      sum (l_par `using` parList rnf)
          where
            newLine _ [] = []
            newLine j line = pre ++ post `using` strategyN
                             where
                               pre  = [ line !! (k-1) | k <- [jLo..j-1] ]
                               post = [ line !! (k-1) | k <- [j+1..jHi] ]
                               strategyN r = pre `par` post `par` ()
            determine1 j = (if pivot > 0 then
                              sign*pivot*det' `using` strategyD1
                            else
                              0) `using` sPar sign
                            where
                              sign = if (even (j-jLo)) then 1 else -1
                              pivot = (head mat) !! (j-1)
                              mat' = SqMatrixC ((iLo,jLo),(iHi-1,jHi-1))
                                                 (map (newLine j) (tail mat))
                              det' = determinant mat'
                              strategyD1 r =
```

```
                        parSqMatrix (parList rwhnf) mat' `seq`
                        det' `par` ()
        l_par = map determine1 [jLo..jHi]
```

Runtime-System Options Index

-

Concept Index

Table of Contents

Short Contents