Distributed and Parallel Technology

Glasgow parallel Haskell

Hans-Wolfgang Loidl
http://www.macs.hw.ac.uk/~hwloidl

School of Mathematical and Computer Sciences
Heriot-Watt University, Edinburgh

Semester 2 2016/17

[0]No proprietary software has been used in producing these slides

Parallel Programming

Engineering a parallel program entails specifying

- *computation*: a correct, efficient algorithm
- *coordination*: arranging the computations to achieve "good" parallel behaviour. Metrics include:
  - Speedup, i.e. reduction in execution time, and defined as execution time on 1 processor / time on n processors:
    speedup = t_1 / t_n
  - Efficiency: a speedup of 14 is good on 16 processors, but poor on 100.

Coordination Aspects

Coordinating parallel behaviour entails, *inter alia*:

- partitioning
  - what threads to create
  - how much work should each thread perform
- thread synchronisation
- load management
- communication
- storage management

Specifying full coordination details is a significant burden on the programmer

High Level Parallel Programming

High level parallel programming aims to reduce the programmer's coordination management burden.

This can be achieved by using

- specific execution models (array languages such as *SAC*),
- skeletons or parallel patterns (*MapReduce/Hadoop*, *Eden*),
- data-oriented parallelism (*PGAS* languages),
- dataflow languages (such as *Swan*),
- parallelising compilers (*pH* for Haskell).

GpH (Glasgow parallel Haskell) uses a model of *semi-explicit* parallelism: the programmer only needs to *identify* potential parallelism.

High Level Parallel Programming

GpH (Glasgow parallel Haskell) aims to simplify parallel programming by requiring the programmer to specify only a few key aspects of parallel programming, and leaving the language implementation to automatically manage the rest.

GpH is a parallel extension to the non-strict, purely functional language Haskell.

GpH Coordination Primitives

GpH provides parallel composition to *hint* that an expression may usefully be evaluated by a parallel thread.
We say `x` is *"sparked"*: if there is an idle processor a thread may be created to evaluate it.

Evaluation

`x `par` y ⇒y`

GpH provides sequential composition to sequence computations and specify how much evaluation a thread should perform. `x` is evaluated to Weak Head Normal Form (WHNF) before returning `y`.

Evaluation

`x `pseq` y ⇒y`

Introducing Parallelism: a GpH Factorial

Factorial is a classic *divide and conquer* algorithm.

Example (Parallel factorial)

```
pfact n = pfact' 1 n

pfact' :: Integer -> Integer -> Integer
pfact' m n
  | m == n     = m
  | otherwise = left `par` right `pseq` (left * right)
      where mid   = (m + n) `div` 2
            left  = pfact' m mid
            right = pfact' (mid+1) n
```

Compare this to the sequential version in `simples.hs`

Controlling Evaluation Order

Notice that we must *control evaluation order:* If we wrote the function as follows, then the addition may evaluate `left` on this core/processor before any other has a chance to evaluate it

```
  | otherwise = left `par` (left * right)
```

The `right `pseq`` ensures that `left` and `right` are evaluated before we multiply them.

Controlling Evaluation Degree

In a non strict language we must specify how much of a value should be computed.

For example the obvious quicksort produces almost no parallelism because the threads reach WHNF very soon: once the first cons cell of the sublist exists!

Example (Parallel quicksort)

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


Controlling Evaluation Degree II

Forcing the evaluation of the sublists gives the desired behaviour:

Example (Forcing evaluation)

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

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]
```

Problem: we need a different forcing function for each datatype, and each composition of datatypes, e.g. list of lists.

GpH Coordination Aspects

To specify parallel coordination in Haskell we must

1. Introduce Parallelism
2. Specify Evaluation Order
3. Specify Evaluation Degree

This is much less than most parallel paradigms, e.g. no communication, synchronisation etc.

It's important that we do so without cluttering the program. In many parallel languages, e.g. C with MPI, coordination so dominates the program text that it obscures the computation.

Evaluation Strategies: Separating Computation and Coordination

Evaluation Strategies abstract over `par` and `pseq`,

- raising the level of abstraction, and
- separating coordination and computation concerns

> *It should be possible to understand the semantics of a function without considering its coordination behaviour.*

Evaluation Strategies

An *evaluation strategy* is a function that specifies the coordination required when computing a value of a given type, and preserves the value i.e. it is an identity function.

```
type Strategy a = a -> Eval a

data Eval a = Done a
```

We provide a simple function to extract a value from `Eval`:

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

The `return` operator from the `Eval` monad will introduce a value into the monad:

```
return :: a -> Eval a
return x = Done x
```

Applying Strategies

`using` applies a strategy to a value, e.g.

```
using :: a -> Strategy a -> a
using x s = runEval (s x)
```

A typical GpH function looks like:

```
somefun x y = someexpr `using` somestrat
```

Simple Strategies

Simple strategies can now be defined.

`r0` performs no reduction at all. Used, for example, to evaluate only the first element but not the second of a pair.

`rseq` reduces its argument to Weak Head Normal Form (WHNF).

`rpar` sparks its argument.

```
r0 :: Strategy a
r0 x = Done x

rseq :: Strategy a
rseq x = x `pseq` Done x

rpar :: Strategy a
rpar x = x `par` Done x
```

Controlling Evaluation Order

We control evaluation order by using a monad to sequence the application of strategies.

So our parallel factorial can be written as:

Example (Parallel factorial)

```
pfact' :: Integer -> Integer -> Integer
pfact' m n
  | m == n    = m
  | otherwise = (left * right) `using` strategy
      where mid   = (m + n) `div` 2
            left  = pfact' m mid
            right = pfact' (mid+1) n
            strategy result = do
                                rpar left
                                rseq right
                                return result
```

Controlling Evaluation Degree - The `DeepSeq` Module

Both `r0` and `rseq` control the evaluation degree of an expression.

It is also often useful to reduce an expression to *normal form* (NF), i.e. a form that contains *no* redexes. We do this using the `rnf` strategy in a type class.

As NF and WHNF coincide for many simple types such as `Integer` and `Bool`, the default method for `rnf` is `rwhnf`.

```
class NFData a where
  rnf ::  a -> ()
  rnf x = x `seq` ()
```

We define `NFData` instances for many types, e.g.

```
instance NFData Int
instance NFData Char
instance NFData Bool
```

We can define `NFData` for type constructors, e.g.

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

We can define a `deepseq` operator that fully evaluates it's first argument:

```
deepseq :: NFData a => a -> b -> b
deepseq a b = rnf a `seq` b
```

Evaluation Degree Strategies

Reducing all of an expression with `rdeepseq` is by far the most common evaluation degree strategy:

```
rdeepseq :: NFData a => Strategy a
rdeepseq x = x `deepseq` Done x
```

Combining Strategies

As strategies are simply functions they can be combined using the full power of the language, e.g. passed as parameters or composed.

`dot` composes two strategies on the same type:

```
dot :: Strategy a -> Strategy a -> Strategy a
s2 `dot` s1 = s2 . runEval . s1
```

`evalList` sequentially applies strategy `s` to every element of a list

```
evalList :: Strategy a -> Strategy [a]
evalList s []     = return []
evalList s (x:xs) = do x' <- s x
                       xs' <- evalList s xs
                       return (x':xs')
```

Data Parallel Strategies

Often coordination follows the data structure, e.g. a thread is created for each elements of a data structure.

For example `parList` applies a strategy to every element of a list in parallel using `evalList`

```
parList :: Strategy a -> Strategy [a]
parList s = evalList (rpar `dot` s)
```

For tuples, `parTuple2` evaluates both elements in parallel:

```
parTuple2 :: Strategy a -> Strategy b
                        -> Strategy (a,b)
parTuple2 strat1 strat2 =
  evalTuple2 (rpar `dot` strat1)
             (rpar `dot` strat2)
```

Data-oriented Parallelism

`parMap` is a higher order function using a strategy to specify data-oriented parallelism over a list.

Example (Parallel map)

```
parMap strat f xs = map f xs `using` parList strat
```

Use it like this:

```
parMap rdeepseq fact [12 .. 30]
```

Exercise

How many threads are created by the example above?

Control-oriented Parallelism

Example (Parallel quicksort)

```
quicksortS []     = []
quicksortS [x]    = [x]
quicksortS (x:xs) =
  losort ++ (x:hisort) `using` strategy
    where
      losort = quicksortS [y|y <- xs, y < x]
      hisort = quicksortS [y|y <- xs, y >= x]
      strategy res = do
                       (rpar `dot` rdeepseq) losort
                       (rpar `dot` rdeepseq) hisort
                        rdeepseq res
```

Note how the coordination code (in `strategy`) is cleanly separated from the computation.

Thread Granularity

Some programs have *coarse grain* parallelism, i.e. there are only a few threads. The challenge then is to create enough threads to utilise all Processing Elements (PEs).

More commonly programs have massive fine-grain parallelism, and several techniques are used to increase thread granularity.

It is only worth creating a thread if the *cost of the computation will outweigh the overheads* of the thread, including

- communicating the computation
- thread creation
- memory allocation
- scheduling

It may be necessary to transform the program to achieve good parallel performance, e.g. to improve thread granularity.

Thresholds

Basic Idea:
Small tasks can be avoided in divide and conquer programs by not dividing the problem once a *threshold* is reached, and instead solving the small problem sequentially.

Threshold Factorial

Example (Parallel factorial with thresholding)

```
pfactThresh :: Integer -> Integer -> Integer
pfactThresh n t = pfactThresh' 1 n t

-- thresholding version
pfactThresh' :: Integer -> Integer -> Integer -> Integer
pfactThresh' m n t
  | (n-m) <= t = product [m..n]   -- seq solve
  | otherwise = (left * right) `using` strategy
      where mid   = (m + n) `div` 2
            left  = pfactThresh' m mid t
            right = pfactThresh' (mid+1) n t
            strategy result = do
                                rpar left
                                rseq right
                                return result
```


Chunking Data Parallelism

Evaluating individual elements of a data structure may give too fine thread granularity, whereas evaluating many elements in a single thread give appropriate granularity. The number of elements (the size of the chunk) can be tuned to give good performance.

It's possible to do this by changing the computational part of the program, e.g. replacing

```
parMap rdeepseq fact [12 .. 30]
```

with

```
concat (parMap rdeepseq
          (map fact) (chunk 5 [12 .. 30]))

chunk :: Int -> [a] -> [[a]]
chunk _ [] = [[]]
chunk n xs = y1 : chunk n y2
  where
    (y1, y2) = splitAt n xs
```


Strategic Chunking

Rather than change the computational part of the program, it's better to change only the strategy.

We can do so using the `parListChunk` strategy which applies a strategy `s` sequentially to sublists of length `n`:

```
map fact [12 .. 30] `using` parListChunk 5 rdeepseq
```

The definition of `parListChunk` is provided in the strategies library:

Example (Chunking)

```
parListChunk :: Int -> Strategy [a] -> Strategy [a]
parListChunk n s =
  parListSplitAt n s (parListChunk n s)

parListSplitAt :: Int -> Strategy [a]
                  Strategy [a] -> Strategy [a]
parListSplitAt n stratPref stratSuff =
 evalListSplitAt n (rpar `dot` stratPref)
                   (rpar `dot` stratSuff)
```

The definition of `parListChunk` is based on this, sequential strategy:

Example (Chunking)

```
evalListSplitAt :: Int -> Strategy [a] ->
                   Strategy [a] -> Strategy [a]
 evalListSplitAt n stratPref stratSuff [] = return []
 evalListSplitAt n stratPref stratSuff xs
  = do
      ys' <- stratPref ys
      zs' <- stratSuff zs
      return (ys' ++ zs')
      where
        (ys,zs) = splitAt n xs
```

NB: This strategy only specifies evaluation degree and order, **not** parallelism!

Systematic Clustering

Sometimes we require to aggregate collections in a way that cannot be expressed using only strategies. We can do so systematically using the `Cluster` class:

- `cluster n` maps the collection into a collection of collections each of size `n`
- `decluster` retrieves the original collection
  `decluster . cluster == id`
- `lift` applies a function on the original collection to the clustered collection

Example (Cluster class)

```
class (Traversable c, Monoid a) => Cluster a c where
  cluster    :: Int -> a -> c a
  decluster  :: c a -> a
  lift       :: (a -> b) -> c a -> c b

  lift = fmap       -- c is a Functor, via Traversable
  decluster = fold  -- c is Foldable,  via Traversable
```


Systematic Clustering (cont'd)

An instance for lists requires us only to define `cluster`

```
instance Cluster [a] [] where
  cluster = chunk
```

Read this as: In order to "cluster" parts of a list, combine them into lists, or in short cluster lists by lists.
This means that the algorithm will operate over lists-of-lists in the clustered version, rather than flat lists.

A Strategic Div&Conq Skeleton

Example (Parallel divide-and-conquer)

```
divConq :: (a -> b)            -- compute the result
        -> a                   -- the value
        -> (a -> Bool)         -- threshold reached?
        -> (b -> b -> b)       -- combine results
        -> (a -> Maybe (a,a)) -- divide
        -> b
divConq f arg threshold conquer divide = go arg
 where
  go arg =
    case divide arg of
     Nothing      -> f arg
     Just (l0,r0) -> conquer l1 r1 `using` strat
    where
     l1 = go l0
     r1 = go r0
     strat x = do r l1; r r1; return x
               where r | threshold arg = rseq
                       | otherwise     = rpar
```

Evaluation Strategy Summary

- use laziness to separate algorithm from coordination
- use the `Eval` monad to specify evaluation order
- use overloaded functions (`NFData`) to specify the evaluation-degree
- provide high level abstractions, e.g. parList, parSqMatrix
- are functions in algorithmic language ⇒
  - comprehensible,
  - can be combined, passed as parameters etc,
  - extensible: write application-specific strategies, and
  - can be defined over (almost) any type
- general: pipeline, d&c, data parallel etc.
- Capable of expressing complex coordination, e.g. Embedded parallelism, `Clustering`, skeletons

Uses of GpH

Many Haskell Programs about 1 in 3 *existing* functional programs will give acceptable speedups on multicores [TFP10].

Parallel Prototyping. A high-level coordination notation means that the programmer can explore alternative parallelisations with relatively little effort. With low-level notations a single parallelisation must be must be designed into the program from the start.

High-performance computational finance, e.g. performing data-intensive, large-scale risk assessment of derivatives etc (e.g. at Standard Chartered, Jane Street Capital etc).

Data-intensive computational in general, e.g. data mining for user profiles (e.g. Facebook).

Parallel symbolic applications, e.g. natural language processors, symbolic algebra systems, etc.

Teaching parallel programming because the clear concepts, such as separation of computation and coordination.

A Methodology for Parallelisation

1. *Sequential implementation.* Start with a correct implementation of an inherently-parallel algorithm.
2. *Parallelise Top-level Pipeline.* Most non-trivial programs have a number of stages, e.g. lex, parse and typecheck in a compiler. Pipelining the output of each stage into the next is very easy to specify, and often gains some parallelism for minimal change.
3. *Time Profile* the sequential application to discover the "big eaters", i.e. the computationally intensive pipeline stages.
4. *Parallelise Big Eaters* using evaluation strategies. It is sometimes possible to introduce adequate parallelism without changing the algorithm; otherwise the algorithm may need to be revised to introduce an appropriate form of parallelism, e.g. d & c or data-parallelism.

A Methodology for Parallelisation

1. *Idealised Simulation.* Simulate the parallel execution of the program on an idealised execution model, i.e. with an infinite number of processors, no communication latency, no thread-creation costs etc. This is a "proving" step: if the program isn't parallel on an idealised machine it won't be on any real machine. A simulator is often easier to use, more heavily instrumented, and can be run in a more convenient environment, e.g. a desktop.
2. *Realistic Simulation.* Some simulators, like GranSim, can be parameterised to emulate a particular parallel architecture, forming a bridge between the idealised and real machines. A major concern at this stage is to improve thread granularity so as to offset communication and thread-creation costs.
3. *Tune on Target Architecture.* Use performance visualisation tools (generally less detailed) to improve performance.

At the latter 3 stages, consider alternative parallelisations.

Comparison with Low & Medium-level Methodologies

This methodology is *heavily tool-based*, probably even more than for medium- and low-level approaches, where it is also important for getting good sequential and parallel performance.

Typically only the top-level code is modified by adding a strategy; *no restructuring* of code is necessary to achieve parallelism.

There is *no risk of deadlock* due to parallel coordination. However, programs might not terminate if there is infinite recursion or full evaluation of an infinite data structure.

Parallelism is largely *independent of the underlying architecture*. No program changes should be necessary when moving to e.g. a larger parallel machine.

Comparison with Low & Medium-level Methodologies

In contrast, in low & medium-level methodologies:

it is very unusual to start with a sequential program; rather, the parallel coordination is usually designed into the program from the beginning;

enforcing a specific evaluation order and parallelism pattern is easier, because models such as C+MPI enforce sequential evaluation in the host language;

because producing a parallel version is so time-consuming, only a very small number of alternative parallelisations are considered (e.g. just one).

Developing a Parallel Matrix Multiplication

As an example of a parallel program, we return to our old friend: matrix multiplication.

Problem If matrix A is an $m \times n$ matrix $[a_{ij}]$ and B is an $n \times p$ matrix $[b_{ij}]$, then the product is an $m \times p$ matrix C where $C_{ik} = \Sigma_{j=1}^{n} a_{ij} b_{jk}$

1: Sequential Implementation

```
-- Type synonyms
type Vec a = [a]
type Mat a = Vec (Vec a)

-- vector multiplication ('dot-product')
mulVec :: Num a => Vec a -> Vec a -> a
u `mulVec` v = sum (zipWith (*) u v)

-- matrix multiplication, in terms of vector multiplicatior
mulMat :: Num a => Mat a -> Mat a -> Mat a
a `mulMat` b =
  [[u `mulVec` v | v <- bt ] | u <- a]
     where bt = transpose b
```

3: Time Profile[1]

See GHC profiling documentation `http://www.haskell.org/ghc/docs/latest/html/users_guide/profiling.html`

Compile for sequential profiling `-prof -auto-all`. Note naming convention for profiling binary.

Run for a 200 by 200 matrix with time `-pT` and space `-hC` profiling turned on

```
jove% ghc -prof -auto-all --make -threaded
  -o MatMultSeq_prof MatMultSeq.hs
jove% MatMultSeq_prof 200 20 20 20 13   +RTS -pT -hC
```

Inspect profiles:

```
jove% more MatMultSeq_prof.prof
 Fri Feb  3 13:07 2012 Time and Allocation Profiling Report  (Final)

   MatMultSeq_prof +RTS -pT -hC -RTS 200 20 20 20 13

 total time  =        2.24 secs   (112 ticks @ 20 ms)
 total alloc = 1,416,495,544 bytes  (excludes profiling overheads)
```

[1] "2: Top-level Pipeline" does not apply here.

Time profile

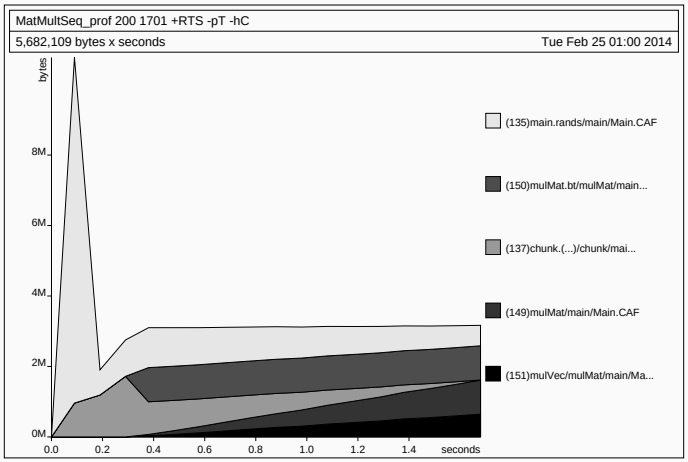
```
COST CENTRE      MODULE                %time %alloc

mulVec           Main                   94.6   92.8
main             Main                    4.5    6.2

                                         individual    inherited
COST CENTRE      MODULE    no. entries  %time %alloc   %time %alloc

MAIN              MAIN       1       0    0.0    0.0   100.0  100.0
 CAF              Main     338       8    0.0    0.0   100.0  100.0
  mulMat          Main     346       1    0.0    0.3    94.6   93.1
   mulVec         Main     347       1   94.6   92.8    94.6   92.8
  main            Main     344       1    4.5    6.2     5.4    6.9
   chunk          Main     345     404    0.9    0.7     0.9    0.7
 CAF           GHC.Read     315       1    0.0    0.0     0.0    0.0
 CAF      Text.Read.Lex     303       8    0.0    0.0     0.0    0.0
 CAF           GHC.Int     299       1    0.0    0.0     0.0    0.0
 CAF  GHC.IO.Handle.FD     275       2    0.0    0.0     0.0    0.0
 CAF System.Posix.Internal274       2    0.0    0.0     0.0    0.0
 ...
jove%
```


Space Profile

Improving space consumption is important for sequential tuning: minimising space usage saves time and reduces garbage collection.

```
% hp2ps MatMultSeq_prof.hp
% ghostview -orientation=seascape MatMultSeq_prof.ps
```


Space Profile



4: Parallelise Big Eaters

1st attempt: parallelise every element of the result matrix, or both ‘maps’

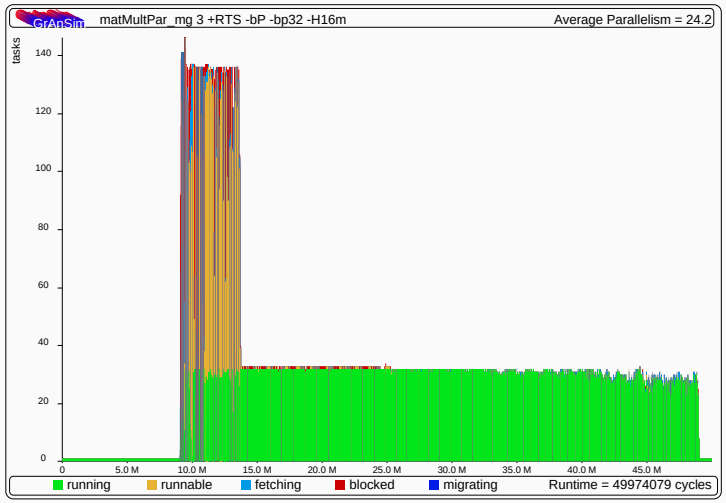
```
mulMatPar :: (NFData a, Num a) =>
                 Mat a -> Mat a -> Mat a
mulMatPar a b = (a `mulMat` b) `using` strat
 where
  strat m = parList (parList rdeepseq) m
```


5: Idealised Simulation

The simulated measurements are all for pairs of 96x96 matrices.

Compile for simulation & simulate the program on an idealised 32-processor machine.

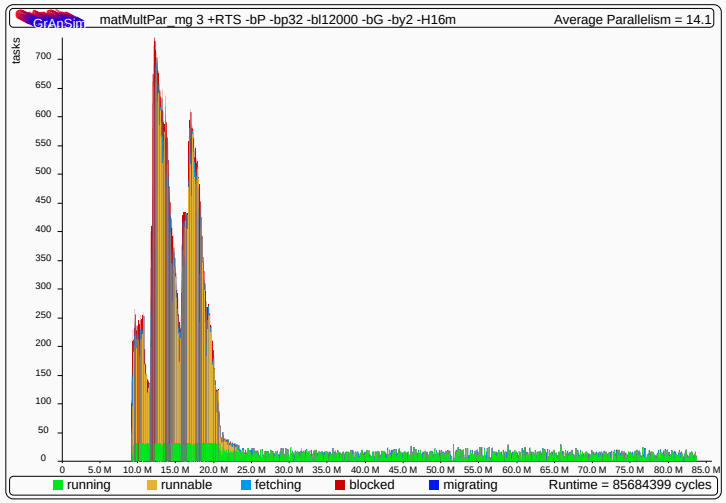
Postprocess & view results



7: Tune on Target Architecture

Run program on a simulated 32-processor Beowulf cluster, i.e. high latency with thread overheads, etc.

Postprocess & view results



Shared-Memory Naive Results

600 x 600 matrices on an 8-core shared memory machine (Dell PowerEdge) `lxpara2`.

Compile with profiling; run on 4 cores; view results

```
% ghc --make -O2 -threaded -eventlog
  -o MatMultPM MatMultPM.hs
% ./MatMultPM 600 90 20 20 13 +RTS -N7 -sstderr -ls
% threadscope MatMultPM.eventlog
```

No. Cores	Runtime (s)	Relative Speedup	Absolute Speedup
Seq	56.1		1.0
1	62.6	1.0	*0.89*
2	56.9	*1.10*	0.99
4	59.7	1.04	0.95
7	60.2	1.04	0.96

Improving Granularity

Currently parallelise both maps (outer over columns, inner over rows)

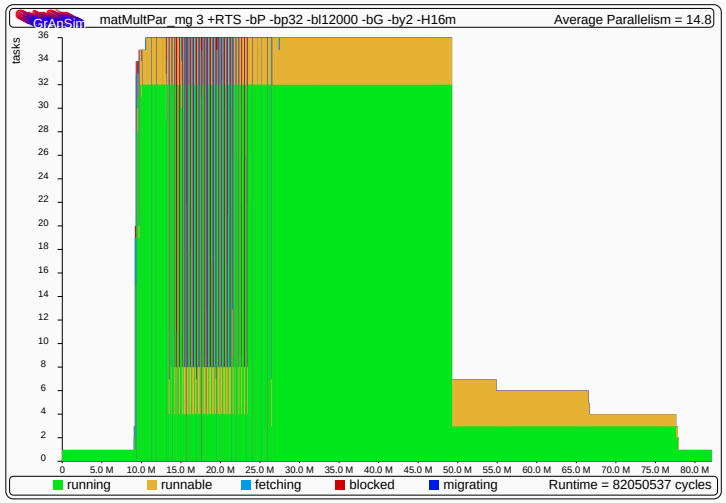
Parallelising only the outer, and performing the inner sequentially will *increase thread granularity*.

```
mulMatParRow :: (NFData a, Num a) =>
                Mat a -> Mat a -> Mat a
mulMatParRow a b =
  (a `mulMat` b) `using` strat
    where
      strat m = parList rdeepseq m
```


Improving Granularity (cont'd)

Granularity can be further increased by 'row clustering', i.e. evaluating `c` rows in a single thread, e.g.

```
mulMatParRows :: (NFData a, Num a) =>
  Int -> Mat a -> Mat a -> Mat a
mulMatParRows m a b =
  (a `mulMat` b) `using` strat
    where
      strat m = parListChunk c rdeepseq m
```



Shared-Memory Row-Clustered Results

600 x 600 matrices with clusters of 90 rows:

No. Cores	Runtime (s)	Relative Speedup	Absolute Speedup
Seq	56.1		1.0
1	60.4	1.0	0.93
2	31.4	1.9	1.8
4	18.0	3.4	3.4
7	9.2	6.6	*6.6*

Reducing Communication

Using blockwise clustering (a.k.a. Gentleman's algorithm) reduces communication as only part of matrix B needs to be communicated.

N.B. Prior to this point we have preserved the computational part of the program and simply added strategies. Now additional computational components are added to cluster the matrix into blocks size `m` times `n`.

```
mulMatParBlocks :: (NFData a, Num a) =>
  Int -> Int -> Mat a -> Mat a -> Mat a
mulMatParBlocks m n a b =
 (a `mulMat` b) `using` strat
  where
   strat x = return (unblock (block m n x
               `using` parList rdeepseq))
```


`block` clusters a matrix into a matrix of matrices, and `unblock` does the reverse.

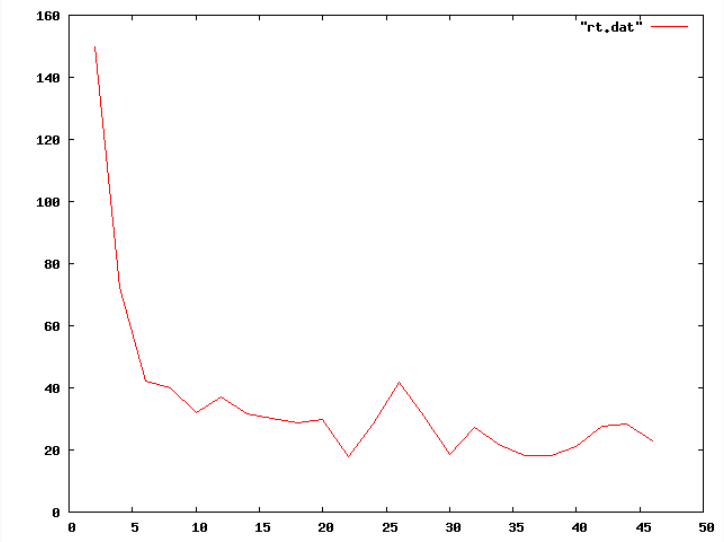
```
block :: Int -> Int -> Mat a -> Mat (Mat a)
block m n = map f . chunk m where
  f :: Mat a -> Vec (Mat a)
  f = map transpose . chunk n . transpose

-- Left inverse of @block m n@.
unblock :: Mat (Mat a) -> Mat a
unblock = unchunk . map g where
  g :: Vec (Mat a) -> Mat a
  g = transpose . unchunk . map transpose
```

7: Tune on Target Architecture: Shared Memory

600 x 600 matrices with block clusters: 20 x 20

No. Cores	Runtime (s)	Relative Speedup	Absolute Speedup
Seq	56.1		1.0
1	60.4	1.0	0.93
2	26.9	*2.2*	2.1
4	14.1	*4.2*	3.9
7	8.4	*7.2*	*6.7*

Runtimes on a 48-core server



Parallel Threadscope Profiles

For parallelism profiles compile with option `-eventlog` (all on one line!)

```
ghc -O2 -rtsopts -threaded -eventlog
    -o parsum_thr_l parsum.hs
```

then run with runtime-system option `-ls`

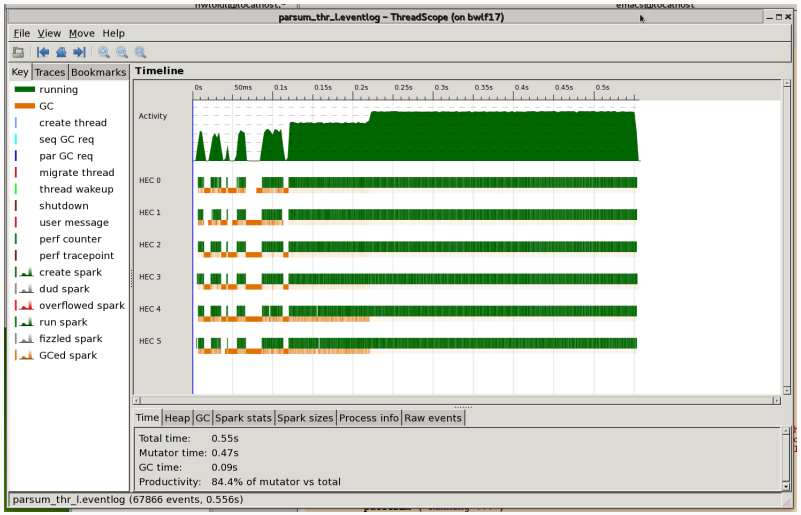
```
./parsum_thr_l 90M 100 +RTS -N6 -ls
```

and visualise the generated eventlog profile like this:

```
/home/hwloidl/.cabal/bin/threadscope parsum_thr_l.eventlog
```

You probably want to do this on small inputs, otherwise the eventlog file becomes huge!

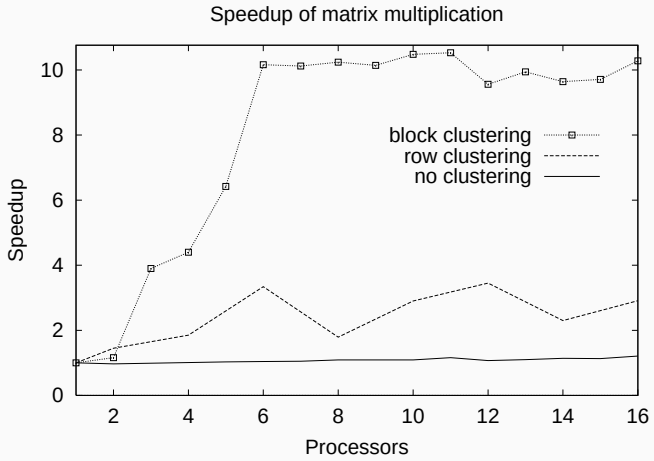
Parallel Threadscope Profiles

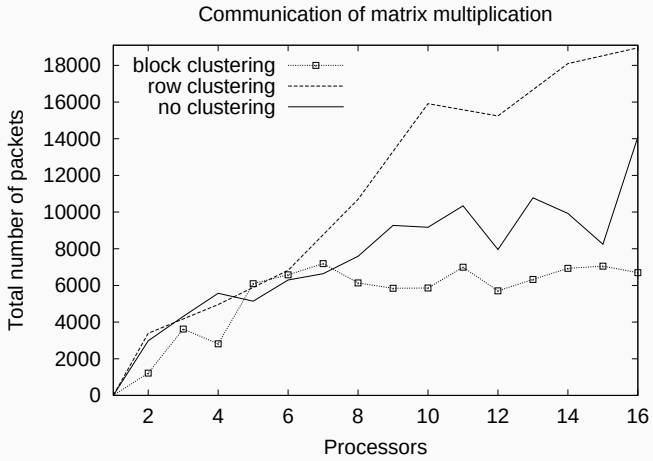


7: Tune on Target Architecture: Beowulf Cluster

Performance measurement on our Beowulf cluster.

Matrix size: 300 $\times$ 300 arbitrary precision integers
Row-wise clusters: 15 rows
Block-wise clusters: 60 x 60





Summary

We have considered a methodology for developing programs with high-level parallelism.

The methodology is

- Tool-based, using both sequential and parallel profiling.
- Iterative
- Requires careful consideration of coordination aspects such as thread granularity and communication

GpH Exercise

- The naive fibonacci program from `parfib.hs` sparks a huge number of small grained tasks. Add a threshhold to reduce the number of sparks produced.
  Hint: Check the thresholding used in `parsum.hs` for an example.
- Produce a data parallel version of the `queens.hs` program.
  Hint 1: you will need to use strategies.
  Hint 2: for perf. measurement use a board of at least 12×12.
  Hint 3: there are several ways of parallelising the program, and you should explore which works best.

For a full list of (parallel) Haskell exercises see: `http://www.macs.hw.ac.uk/~hwloidl/Courses/F21DP/tutorial0.html`

HERIOT WATT UNIVERSITY