LinSolv: a Case Study in Strategic Parallelism

Hans-Wolfgang Loidl

Department of Computing Science
University of Glasgow
E-mail: hwloidl@dcs.gla.ac.uk

Abstract. This paper discusses the parallelisation and performance tuning of a typical computer algebra algorithm, LinSolv, using evaluation strategies. We present three steps in the parallelisation process starting with a naive parallel version. As this algorithm uses infinite data structures as intermediate values it is necessary to define very sophisticated strategies in order to improve parallel performance.
We also compare the strategic parallel code with pre-strategy code. This comparison shows how evaluation strategies help to localise changes needed for parallelisation. In particular, the separation between algorithmic and parallel code makes the structure of the parallelism much clearer.

1 Introduction

Tuning the performance of a parallel algorithm can be a long, tiresome process. A parallel programming model should aid the programmer especially in this stage, allowing him to experiment with different patterns of parallel behaviour. Based on our experiences with developing parallel functional programs we have developed evaluation strategies [THLP98], which separate algorithmic from behavioural code. This separation has played a crucial role in the performance tuning stage of parallelising several medium- to large-scale functional programs [LT97].

In this paper we use evaluation strategies to parallelise a linear system solver, LinSolv. A short description of the final parallel version has been presented in [LT97] already. This paper focuses on the performance tuning process and compares the pre-strategy with the strategic versions of the code. LinSolv is of special interest to us because it has a structure that is common to a whole class of computer algebra algorithms: a multiple homomorphic images structure. It is based on the idea of solving a problem in several "simpler" domains and recombining the results into a result in the original domain. Although this approach contains a high degree of parallelism, a naive parallel version exhibits several performance bottlenecks. Therefore, we refine the naive parallel algorithm and discuss three stages in the performance tuning process. For the parallel program development we use the GranSim [HLP95] simulator for the parallel execution of GpH programs. GpH is a parallel extension of Haskell, which requires the programmer only to expose parallelism. Other aspects of the parallel program execution are hidden inside the runtime-system.

Prior to developing a parallel strategic version of LinSolv we have developed a parallel pre-strategy version. The comparison of both versions in this paper shows that the use of strategies yields a far cleaner structure of the algorithm, with almost all of the parallelism specified on top level. This clarifies the essential differences between the parallel versions of the algorithm.

The structure of this paper is as follows. Section 2 introduces evaluation strategies. Section 3 first presents a sequential version of LinSolv and then discusses its parallelisation and performance tuning. Finally, Section 4 concludes.

2 Evaluation Strategies

This section gives an abridged description of evaluation strategies. A complete description and discussion of strategies can be found in [THLP98].

2.1 GpH Parallelism

The essence of the problem facing the parallel programmer is that, in addition to specifying *what* value the program should compute, explicitly parallel programs must also specify *how* the machine should organise the computation. In GpH we have to specify only a few aspects of the parallel execution, with the runtime system managing details like synchronisation of threads and transfer of data. The programmer is only required to indicate those values that might usefully be evaluated by parallel threads and, since our basic execution model is a lazy one, perhaps also the extent to which those values should be evaluated. We term these programmer-specified aspects the program's *dynamic behaviour*.

Parallelism is introduced in GpH by the `par` combinator, which takes two arguments that are to be evaluated in parallel. The expression `p 'par' e` has the same value as `e`, and is not strict in its first argument, i.e. $\perp$ `'par' e` has the value of `e`. Its dynamic behaviour is to indicate that `p` could be evaluated by a new parallel thread, with the parent thread continuing evaluation of `e`. We say that `p` has been *sparked*, and a thread may subsequently be created to evaluate it if a processor becomes idle. Since the thread is not necessarily created, `p` is similar to a *lazy future* [MKH91].

Since control of sequencing can be important in a parallel language [Roe91], we introduce a sequential composition operator, `seq`. If `e1` is not $\perp$, the expression `e1 'seq' e2` has the value of `e2`; otherwise it is $\perp$. The corresponding dynamic behaviour is to evaluate `e1` to weak head normal form (WHNF) before returning `e2`. Note that this presentation of strategies is based on Haskell 1.2.

2.2 Evaluation Strategies

Evaluation strategies use lazy higher-order functions to separate the two concerns of specifying the algorithm and specifying the program's dynamic behaviour. A function definition is split into two parts, the algorithm and the strategy, with values defined in the former being manipulated in the latter. The algorithmic code is consequently uncluttered by details relating only to the parallel behaviour. In fact the driving philosophy behind evaluation strategies is that *it should be possible to understand the semantics of a function without considering its dynamic behaviour*. A complete description and discussion of strategies can be found in [THLP98].

A strategy is a function that specifies the dynamic behaviour required when computing a value of a given type. A strategy makes no contribution towards the value being computed by the algorithmic component of the function: it is evaluated purely for effect, and hence it returns just the nullary tuple `()`.

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

2.3 Strategies Controlling Evaluation Degree

The simplest strategies introduce no parallelism: they specify only the evaluation degree. The simplest strategy is termed `r0` and performs no reduction at all. Perhaps surprisingly, this strategy proves very useful, e.g. when evaluating a pair we may want to evaluate only the first element but not the second.

```
r0 :: Strategy a
r0 _ = ()
```

Because reduction to WHNF is the default evaluation degree in GpH, a strategy to reduce a value of any type to WHNF is easily defined:

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

Many expressions can also be reduced to *normal form* (NF), i.e. a form that contains *no* redexes, by the `rnf` strategy. The `rnf` strategy can be defined over built-in or datatypes, but not over function types or any type incorporating a function type as few reduction engines support the reduction of inner redexes within functions. Rather than defining a new `rnfX` strategy for each data type `X`, it is better to have a single overloaded `rnf` strategy that works on any data type. The obvious solution is to use a Haskell type class, `NFData`, to overload the `rnf` operation. Because NF and WHNF coincide for built-in types such as integers and booleans, the default method for `rnf` is `rwhnf`.

```
class NFData a where
  rnf :: Strategy a
  rnf = rwhnf
```

For each data type an instance of `NFData` must be declared that specifies how to reduce a value of that type to normal form. Such an instance relies on its element types, if any, being in class `NFData`. Consider lists and pairs for example.

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

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

2.4 Combining Strategies

Because evaluation strategies are just normal higher-order functions, they can be combined using the full power of the language, e.g. passed as parameters or composed using the function composition operator. Elements of a strategy are combined by sequential or parallel composition (`seq` or `par`). Many useful strategies are higher-order, for example, `seqList` below is a strategy that sequentially applies a strategy to every element of a list. The strategy `seqList r0` evaluates just the spine of a list, and `seqList rwhnf` evaluates every element of a list to WHNF.

```
seqList :: Strategy a -> Strategy [a]
seqList strat []     = ()
seqList strat (x:xs) = strat x `seq` (seqList strat xs)
```

2.5 Data-oriented Parallelism

A strategy can specify parallelism and sequencing as well as evaluation degree. Strategies specifying data-oriented parallelism describe the dynamic behaviour in terms of some data structure. For example `parList` is similar to `seqList`, except that it applies the 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)
```

Data-oriented strategies are applied by the `using` function which applies the strategy to the data structure `x` before returning it.

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

A parallel map is a useful example of data-oriented parallelism; for example the `parMap` function defined below applies its function argument to every element of a list in parallel. Note how the algorithmic code `map f xs` is cleanly separated from the strategy. The `strat` parameter determines the dynamic behaviour of each element of the result list, and hence `parMap` is parametric in some of its dynamic behaviour.

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

2.6 Control-oriented parallelism

In control-oriented parallelism subexpressions of a function are selected for parallel evaluation. A control-oriented strategy is typically a sequence of strategy applications composed with `par` and `seq` that specifies which subexpressions of a function are to be evaluated in parallel, and in what order. The sequence is loosely termed a strategy, and is invoked by either the `demanding` or the `sparking` function. The Haskell `flip` function simply reorders a binary function's parameters.

```
demanding, sparking :: a -> () -> a

demanding = flip seq
sparking = flip  par
```

The parallel Fibonacci function below with (divide-and-conquer) control-oriented parallelism also demonstrates how to describe a thresholding mechanism as part of a strategy. More sophisticated applications of thresholding are discussed in the accompanying paper [LMT+97].

```
pfibT n
  | n <= 1    = 1
  | otherwise = (n1+n2+1) `demanding` strategy
    where
      n1 = pfibT (n-1)
      n2 = pfibT (n-2)
      strategy = if n > 10
                 then rnf n1 `par` rnf n2
                 else ()
```

3 LinSolv

The linear system solver we discuss in this paper uses an approach that is very common in the area of computer algebra: a multiple homomorphic images approach [BCLA82]. This approach consists of the following three stages:

1. map the input data into several homomorphic images,
2. compute the solution in each of these images, and
3. combine the results of all images to a result in the original domain.

Since computer algebra algorithms aim at finding *exact solutions* to mathematical problems unbounded data types like arbitrary precision integers are frequently used. In algorithms operating on arbitrary precision integers the original domain is typically $\mathbb{Z}$ and the homomorphic images are $\mathbb{Z}$ modulo p, written $\mathbb{Z}_p$, with p being a prime number. The advantage of this approach becomes clear when the input numbers are very big and each prime number is small enough to fit into one machine word. In this case the basic arithmetic in the homomorphic images is ordinary fixed precision arithmetic with the results never exceeding one machine word. No additional cost for handling arbitrary precision integers has to be paid. Only in the combination phase will the big numbers appear again. In the case of $\mathbb{Z}$ as original domain the well-studied Chinese Remainder Algorithm can be used in the combine step [Lip71].

The linear system solver (LinSolv) we discuss in this section uses such a multiple homomorphic images approach. Thus, it must be emphasised that this algorithm is *not meant to represent a highly-tuned numerical algorithm for finding just an* approximation *of a solution, but a typical symbolic algorithm for finding an* exact *solution*, which represents a wide class of computer algebra algorithms.

It is obvious that this approach lends itself to parallel processing: all solutions in the homomorphic images can be computed independently. An obvious bottleneck is the final combination stage. In the following sections we first discuss the structure of the sequential algorithm. Then we start with a straightforward, parallel version and improve its behaviour by eliminating the two main sequential bottlenecks.

3.1 The Sequential Algorithm

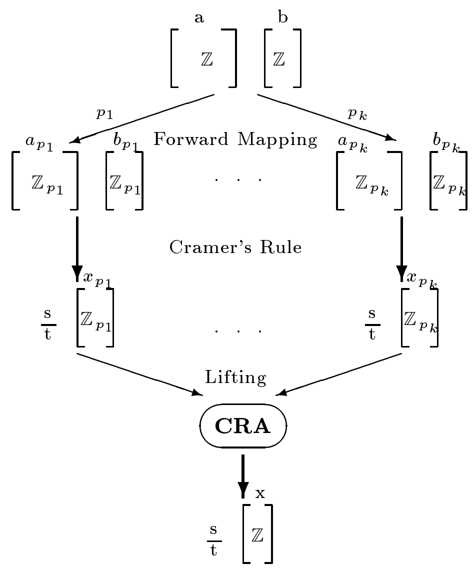

Fig. 1. Structure of the LinSolv algorithm

This section describes the basic structure of the sequential LinSolv algorithm. For a given matrix a and vector b, both ranging over integers, this algorithm finds a solution x to the equation $ax = b$. More formally, the problem we are tackling can be specified as follows:

$$\begin{array}{ll} \textbf{Input}: & a, b \quad \text{where } a \in \mathbb{Z}^{n\times n},\ \det a \neq 0,\ b \in \mathbb{Z}^n \\ \textbf{Output}: & s, t, x \text{ where } a(\frac{s}{t}x) = b, \\ & \qquad s, t \in \mathbb{Z},\ x \in \mathbb{Z}^n \\ & \qquad \gcd(s,t) = 1,\ \gcd_{i=1,\ldots,n} x_i = 1 \end{array}$$

where $\mathbb{Z}$ denotes the set of all integers; for a domain $\mathbb{D}$ and an integer n, $\mathbb{D}^n$ denotes the set of all vectors of length n with components from $\mathbb{D}$; and $\mathbb{D}^{n\times n}$ denotes the set of all 2-dimensional square matrices of size n over $\mathbb{D}$. For an integer n, $\mathbb{Z}_n$ denotes the set of integers $\{0, \ldots, n-1\}$ (the homomorphic image of $\mathbb{Z}$ with base n). Note that we are computing a vector x of integer values and factor out the rational part of the solution into $\frac{s}{t}$. This is convenient when using the result in a bigger application because later stages can avoid most of the expensive rational number arithmetic on the result vector.

We need an algorithm that will compute an exact solution of a system of linear equations over integers of arbitrary size. The main questions to be considered for the efficiency of the sequential algorithm are:

1. How big are the intermediate values in the computation?

2. How high is the overhead associated with using rationals instead of integers?
3. Are there inherently sequential parts in the algorithm?

The first question is directly addressed by using a multiple homomorphic images approach, which bounds every value by the base of the image. The next two questions are crucial in picking a concrete algorithm for the solution phase. The following paragraphs discuss the individual stages of the algorithm with the paragraph on the solution phase discussing the advantages and disadvantages of three alternatives with respect to the questions raised above. Figure 1 summarises the overall structure of the algorithm.

Forward mapping: This stage is trivial: for a given prime number p the function `'mod' p` is mapped over all elements of a and b. This stage is easily parallelised.

Homomorphic solutions: We have investigated several candidates for computing the homomorphic solutions, which have the following characteristics:

- *Gaussian Elimination:* This is a very efficient algorithm often used for solving linear systems of equations. However, since it works over rational numbers the basic arithmetic operations are much more expensive than those over fixed precision integers. An alternative to the classical algorithm would be to introduce rational numbers only in the back-substitution phase by using for example Bareiss' variant of the algorithm. However, this variant requires $O(n^3)$ additional integer divisions, so it is not clear whether it gives an improved performance in practice.
- *LU-Decomposition:* The *LU-Decomposition* method has very strong data dependencies, which yields an inherently sequential algorithm. Our initial parallelisation of LU-Decompostion achieved only a speedup of 3.8 on an idealised machine. Some significant restructuring would be necessary to obtain a reasonable parallel algorithm.
- *Cramer's Rule:* Although this algorithm is less efficient in the sequential case, it is very attractive because of its high potential of parallelism. In this algorithm the result is computed by evaluating $n+1$ independent determinants. We describe its main structure below.

We did not consider the iterative algorithms often used in numerical applications because our goal is to find an exact solution. Furthermore, we insist on using a parallel algorithm for computing a homomorphic solution in order to maintain scalability of the overall algorithm for cases where the number of available processors is higher than the number of homomorphic images used by the algorithm. Using an efficient sequential algorithm might achieve better results for small number of processors but is inherently limited in its parallelism.

The method we use is based on *Cramer's rule.* This rule states that the solution of the equation $ax = b$ can be computed as a vector, with ratios of two determinants as components. In each component the denominator is the determinant of the original matrix. The numerator of the j-th component is the determinant of a matrix where the j-th column has been replaced by the vector b. More formally, let a_{p_i}, b_{p_i} be the homomorphic images of a and b w.r.t. the prime number p_i. Then the solution $x_{p_i} = [x_{p_{i_1}}, \ldots, x_{p_{i_n}}]$ can be computed by:

$$x_{p_{i_j}} = \frac{\det a'_{p_{i_j}}}{\det a_{p_i}}$$

where $a'_{p_{i_j}}$ is a_{p_i} with the j-th column replaced by b_{p_i}.

When applying the above formula in a homomorphic domain $\mathbb{Z}_{p_i}$, the determinant $\det a_{p_i}$ might become 0. Obviously, no solution can be computed in such a domain. Prime numbers p_i which result in $\det a_{p_i}$ being 0 are termed *unlucky* and must be filtered from the list of prime numbers which are used as bases for the homomorphic domains.

Combination: The final stage of the algorithm consist of combining the homomorphic solutions to a solution in the original domain $\mathbb{Z}$ ('lifting'). This combination can be done by using the Chinese Remainder Algorithm (CRA) [Lip71]. This algorithm finds the "original" to two images i.e. a value r, which maps to the given values r_1, r_2 in the images generated by the prime numbers p_1, p_2, respectively. More formally we can specify the algorithm as follows:

Input: r_1, r_2, p_1, p_2 where p_1, p_2 prime, $r_1 \in \mathbb{Z}_{p_1},\ r_2 \in \mathbb{Z}_{p_2}$
Output: r where $r \in \mathbb{Z}_{p_1 p_2},\ r_1 = r \bmod p_1,\ r_2 = r \bmod p_2$

Although the CRA operation is associative, for two lists it is most efficient to use a left associative fold operation over the binary version above (Garner's algorithm [Knu81, p.274]). The reason for this is that all computations in the binary CRA operate in the domain $\mathbb{Z}_{p_2}$, which can be chosen to be a fixed precision domain in each stage. Hence, the large accumulated input values p_1 and r_1 in the folding process are mapped to small numbers, making the binary CRA almost equally cheap in every step of the folding. Unfortunately, this is also an obvious sequential bottleneck.

```
linSolv a b =
  let
    {- forward mapping and solution via Cramer's rule -}
    ...
    xList :: [[Integer]]  -- infinite list of solutions in hom images
    xList = gen_xList primes

    gen_xList (p:ps) =
       let
          modDet = toHom p (determinant (toHom p a))
          pmx = [ toHom p (determinant (replaceColumn j (toHom p a) (toHom p b) ))
                | j <- [jLo..jHi] ]
          ((iLo,jLo),(iHi,jHi)) = bounds a
       in
       if modDet /= 0
         then (p : modDet : pmx)  : gen_xList ps
         else gen_xList ps

   {- combination via CRA -}
    ...
    detList = projection 1 xList

    det = list_cra pBound primes detList detList
    x_i i = list_cra pBound primes x_i_List detList
            where x_i_List = projection (i+2) xList

    x = map x_i [0..n-1]
  in
  x `using` strat
```

Fig. 2. Top level code of the sequential LinSolv algorithm

Figure 2 shows the top level of the algorithm based on Cramer's rule. Note that `xList` is an infinite list of solutions in homomorphic images corresponding to prime numbers in the infinite list `primes`. The CRA computation itself is hidden in `list_cra`, which basically performs a left associative fold operation, accumulating the product of all prime numbers met so far until this product becomes larger than $s^n n!$ (n is the size of the matrix a and s is the maximal element in a and b). The `gen_xList` function has to check whether the modular determinant is 0 in order to avoid picking unlucky prime numbers. The strategy `strat` in the body of the let construct describes the dynamic behaviour of the code separately from the algorithmic code. For the sequential version the default strategy `rwhnf` can be used. Figures 4, 6, and 8 give different definitions of `strat` for parallel execution without changing the code in Figure 2 at all.

3.2 Naive Parallel Algorithm

Figure 3 shows a naive parallel version of LinSolv, written without strategies by parallelising `gen_xList`. The idea of this code is to create a single parallel thread to evaluate both the forward mapping (via `toHom`) and the determinant computations for each prime p_i. However, the actual dynamic behaviour is quite different: the thread sparked for `homsol` will only evaluate the top-level cons cell, which does not trigger the computation of the actual homomorphic solution (`pmx`) at all.

```
linSolv a b =
  let
    {- forward mapping and solution via Cramer's rule -}
    ...
    xList :: [[Integer]]  -- infinite list of solutions in hom images
    xList = gen_xList primes

    gen_xList (p:ps) =
     let
       ...
       homSol = (p : modDet : pmx)
       pmx = parmap ( \ j ->         -- parallelism within each hom im
                      let   a1 = replaceColumn j a0 b0
                      in    modHom p (determinant a1) )
                    [jLo..jHi]
       ((iLo,jLo),(iHi,jHi)) = matBounds a

       restList = gen_xList ps
     in
     if modDet == 0
       then gen_xList ps
       else par homSol (homSol : restList) -- parallelism between hom ims

   {- combination via CRA -}
    ...
    detList = projection 1 xList

    det = list_cra pBound primes detList detList
    x_i i = list_cra pBound primes x_i_List detList
            where x_i_List = projection (i+2) xList

    x = map x_i [0..n-1]
  in
  x
```

Fig. 3. Naive parallel pre-strategy code

Only when the result is required in the combination stage the `parmap` will be triggered, creating parallelism within a homomorphic image but sequentialising all stages. The combination stage is basically a `fold` operation. This causes a sequentialisation of the homomorphic images.

The resulting activity profile at the left hand side of Figure 5 reveals two stages in the computation:

- In the first stage, up to approximately one third of the total execution time, the overall determinant $\det a$ is computed using the same structure as for the overall computation. This causes a sequence of computations in the homomorphic domains, which is visualised as a sequence of small peaks.
- In the second stage, the solution is computed in each homomorphic image. All components of the solution are evaluated in parallel using a parallel determinant computation in each case. This yields a higher degree of parallelism within each stage.

Note that the number of parallel peaks in both stages is determined by the number of homomorphic images necessary to construct the result in the original domain (13 in this case).

The dynamic behaviour of this code becomes much clearer when reformulating the code with strategies. Figure 4 shows the definition of `strat` in the body of the `linSolv` function. Note that in contrast to the pre-strategy version the algorithmic code is unchanged. In the strategic version of the code it becomes clear that two nested strategies are used:

- the outer strategy, `seqListAccum` in this case, traverses the infinite list of solutions (`xList`), and
- the inner strategy, `sol_strat` in this case, traverses the homomorphic solutions (`pmx`).

Each of these strategies can be done either sequentially or in parallel. From the above description of the dynamic behaviour of the naive parallel code it should be clear that both dimensions

```
rnf det                               `seq`
seqListAccum 1 seq_sol_strat xList    `par`
parList rnf x
where
  seqListAccum :: Integer -> Strategy [Integer] -> Strategy [[Integer]]
  seqListAccum accum s =
    \ (xs:xss) -> if accum>pBound
                    then ()
                    else s xs  `seq`
                         seqListAccum (accum*(head xs)) s xss

  seq_sol_strat :: Strategy [Integer]
  seq_sol_strat = \ (p:modDet:pmx) -> rnf modDet  `seq`
                                      if modDet /= 0
                                        then seqList rnf pmx
                                        else ()
```

Fig. 4. Strategy version of a naive parallel LinSolv algorithm

are done sequentially. The outer `seqListAccum` strategy encodes the dynamic behaviour of the algorithm when traversing `xList`: it accumulates the product of all prime numbers in order to decide how many homomorphic solutions to generate. The explicit use of `seq` in `seqListAccum` reflects the evaluation order, which is implicit in the pre-strategy code. The inner `sol_strat` strategy describes a dependency between the `modDet` component of the homomorphic solution and the rest. Although the `parmap` construct in Figure 3 specifies parallelism over the elements of the homomorphic solution it is hidden by the first two elements of the result list in `sol`, which are demanded first when computing the overall determinant `det`. Figure 5 shows that the dynamic behaviours of the pre-strategy and the strategic version are almost identical.

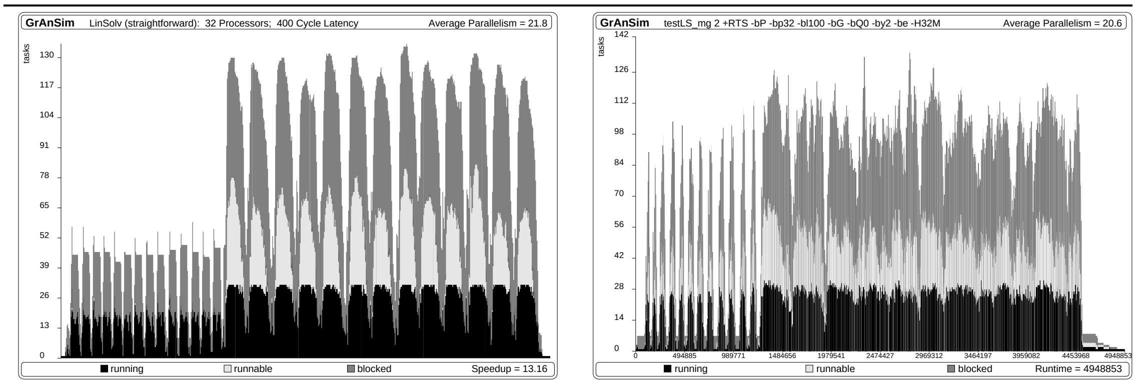

Fig. 5. Activity profile of pre-strategy and strategic naive LinSolv

3.3 Improved version

Reflecting the performance tuning in the pre-strategy version of the code the strategy in Figure 6 shows two changes compared to the previous strategy: it does not force the computation of the determinant as a first step and it computes all components of the homomorphic solution in parallel using the `par_sol_strat` strategy. This avoids the delay in generating parallel processes for performing the most time consuming computations in the solution phase.

The activity profiles in Figure 7 show that the first stage of peaks has been merged with the second stage. The data dependency between the overall CRA and the homomorphic solutions has disappeared. However, by using the `seqListAccum` strategy over `xList` the combination stage is still sequential leading to regular drops in utilisation. In the pre-strategy code this corresponds to the dynamic behaviour generated by the `list_CRA` function.

```
seqListAccum 1 par_sol_strat xList        `par`
parList rnf xs
where
  seqListAccum :: Integer -> Strategy [Integer] -> Strategy [[Integer]]
  seqListAccum accum s =
    \ (xs:xss) -> if accum>pBound
                    then ()
                    else s xs `seq`
                         seqListAccum (accum*(head xs)) s xss

  par_sol_strat :: Strategy [Integer]
  par_sol_strat = \ (p:modDet:pmx) -> rnf modDet `seq`
                                      if modDet /= 0
                                        then parList rnf pmx
                                        else ()
```

Fig. 6. Strategy version of an improved parallel LinSolv algorithm

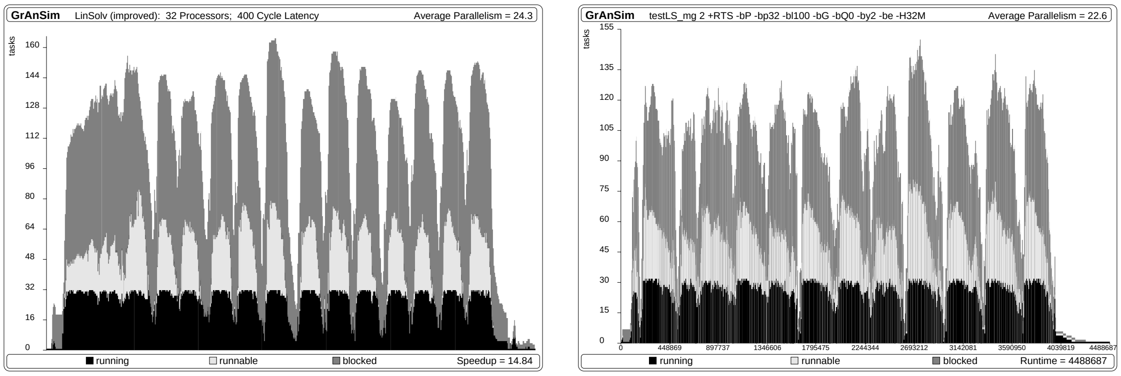

Fig. 7. Activity profiles of pre-strategy and strategic improved LinSolv

3.4 Parallelism over the Homomorphic Images

The strategy in Figure 8 eliminates the sequential traversal of `xList` by guessing the number of primes needed to compute the overall result and using a `parListN` strategy to generate data parallelism over that segment of `xList`. Using `parList` inside the `par_sol_strat` strategy causes each component of the result to be evaluated in parallel. However, we still need the check for zero in order to avoid redundant computation. In order to minimise data dependencies in the algorithm we do not already check for unlucky prime numbers when computing `noOfPrimes`. If some prime numbers turn out to be unlucky the `list_cra` will evaluate more results by demanding a so far unevaluated list element. The final strategy application `parList rnf x` specifies that all elements of the result should be combined in parallel. Without this component there would be a sequence of combination steps at the end of the execution, one for each element in the result vector. In the activity profiles of Figure 9 the individual peaks have been merged into a period of consistently high utilisation.

```
rnf noOfPrimes                              `seq`
parListN noOfPrimes par_sol_strat xList     `par`
parList rnf xs
where
  par_sol_strat :: Strategy [Integer]
  par_sol_strat = \ (p:modDet:pmx) -> rnf modDet `seq`
                                      if modDet /= 0
                                        then parList rnf pmx
                                        else ()
```

Fig. 8. Strategy of the final parallel LinSolv algorithm

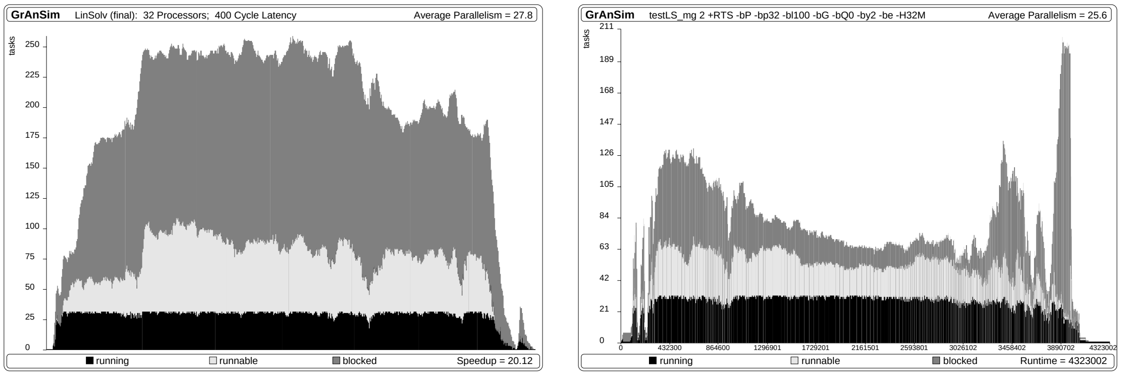

Fig. 9. Activity profiles of pre-strategy and strategic final LinSolv

3.5 Summary

Historically, the development and performance tuning of LinSolv predated the development of evaluation strategies. In hindsight the lack of separation between algorithmic and behavioural code severely complicated the program development. The most striking example is the `tree_CRA` algorithm we used in the pre-strategy version in order to guarantee parallelism between the homomorphic images. In order to handle an infinite list of solutions based on a guess how many solutions are needed, the `tree_CRA` algorithm keeps track of the number of unlucky primes and uses a "fail handler" in order to compute more results if necessary. This leads to the rather complicated algorithm in Figure 10, which combines the computation of the result with a specific dynamic behaviour suitable for parallelism. In contrast, the strategic version uses a much simpler sequential code, which is basically a fold operation which also tests for unlucky primes and accumulates the product of all lucky prime numbers. To add parallelism it is sufficient to change the `seqListAccum` in Figure 4 into a `parListN` in Figure 8. Again the different dynamic behaviour can be described by the top level strategy.

It turns out that the additional parallelism of the combinations in `tree_CRA` does not improve the performance at all because combining two large values (in the nodes of the tree) is far less efficient than combining a large with a small value, which is done in each step of the `list_CRA`. Thus, although the `tree_CRA` generates parallelism at the end of the computation the total runtime actually increases. This can be seen in Table 1 where adding a tree CRA to the basic version of the algorithm (with a parallel determinant computation) does not further improve the efficiency of the algorithm. It only increases the total amount of work compared to a sequential version that uses a `list_CRA`. However, this example shows that the use of strategies allows the programmer to explore different variants of the parallel code without performing a major restructuring of the algorithm.

Table 1 compares the runtimes (in kilocycles), average parallelism, total amount of work (as percentage compared to the work in the sequential setup), and speedups for the three versions

```
-- n ... guess on how many hom sols needed
-- ms ... infinite list of modules
-- as ... infinite list of values
-- ds ... infinite list of homomorphic determinants
tree_CRA :: Integer -> [Integer] -> [Integer] -> [Integer] -> (Integer, Integer)
tree_CRA n ms as ds =
 let
  res@(m, a, fails) = tree_CRA' ms' as' ds'
                      where ms' = take n ms
                            as' = take n as
                            ds' = take n ds
  handle_fails :: Integer -> Integer -> Integer ->
                  [Integer] -> [Integer] -> [Integer] -> (Integer, Integer)
  handle_fails n m a (m1:ms) (a1:as) (d1:ds)
   | n == 0    = (m, a)
   | d1 == 0   = handle_fails n m a ms as ds
   | otherwise = handle_fails (n-1) m' a' ms as ds
                where
                  m'  = m * m1
                  a'  = par_binCRA m m1 inv a a1  -- NB: parallel version
                  inv = modInv m1 m
 in
  handle_fails fails m a ms as ds

-- here all lists are finite
tree_CRA' :: [Integer] -> [Integer] -> [Integer] -> (Integer, Integer, Integer)
tree_CRA' [p] [a] [0] = (1, 1, 1)  -- unlucky prime
tree_CRA' [p] [a] [_] = (p, a, 0)  -- normal case
tree_CRA' ps as ds =
  let
    n = length ps

    (left_ps, right_ps) = splitAt (n `div` 2) ps
    (left_as, right_as) = splitAt (n `div` 2) as
    (left_ds, right_ds) = splitAt (n `div` 2) ds

    left@(left_P, left_CRA, left_fails)  =
          tree_CRA' left_ps left_as left_ds

    right@(right_P, right_CRA, right_fails)  =
          tree_CRA' right_ps right_as right_ds

    inv = modInv right_P left_P
    cra = par_binCRA left_P right_P
                     inv left_CRA right_CRA
  in
    left  `par` right `par` inv `par` (
    cra   `seq`         -- force computation of cra first
           (left_P * right_P,
            cra,
            left_fails + right_fails) )
```

Fig. 10. A tree CRA used in the pre-strategy version

discussed above with different setup variants. Overall the three stages of the parallelisation, from a naive to the final version, show an increasing average parallelism and speedup. The percentage of total work is roughly unchanged, indicating that no speculative parallelism is added during the performance tuning. In each of the three stages the best results are obtained from a setup using a parallel determinant computation in the solution stage. However, the parallel determinant computation performs some redundant work shown by the constantly high percentage of total work. This is mainly due to repeated traversals of data structures when constructing the sub-matrices defined in Cramer's Rule.

As further work it would be interesting to compare this LinSolv version with one using a Gaussian elimination algorithm in the solution phase. Such an implementation would use rational arithmetic rather than integer arithmetic. The tighter data dependencies would probably reduce the parallelism inside a the solution stage. However, the overall structure of the parallelism generated by the multiple homomorphic images approach should be unchanged.

Setup	Runtime	Avg par	Tot work	Speedup
Naive parallel strategy				
Sequential	78,651			
Par Det (default)	4,948	20.6	130%	15.9
Par Det & Tree CRA	5,509	20.6	144%	14.3
Improved strategy				
Sequential	78,651			
Par Det (default)	4,488	22.6	129%	17.5
Par Det & Tree CRA	5,675	20.0	144%	13.9
Final strategy				
Sequential	78,651			
Par Det (default)	4,323	25.6	141%	18.2
Par Det & Tree CRA	5,130	22.1	144%	15.3

Table 1. Measurements of all versions of LinSolv

4 Discussion

This paper has presented a case study in parallelising and tuning a computer algebra algorithm using evaluation strategies. The multiple homomorphic images structure of this algorithm is common to a class of computer algebra algorithms, which makes the results of this case study applicable to algorithms like parallel p-adic arithmetic [LL93] and parallel resultant computation [HL94]. Although the parallel versions of these two algorithms are implemented in an imperative language the top level strategy we have developed in parallelising LinSolv could be reused for these algorithms. A direct comparison of the strategic with the pre-strategy code of LinSolv shows the practical advantages of separating algorithmic from behavioural code. This proves to be particularly important in the performance tuning stage of parallel program development.

To date we have parallelised several medium- to large-scale functional programs, including real applications like Lolita, a natural language engineering system of more than 47,000 lines of Haskell [LMT+97], and a program for detecting accident blackspots out of a database of police accident records [THL+96]. In all of these programs our goal was to achieve moderate parallelism with minimal coding effort. In most cases it was sufficient to describe the parallelism only on the top level of the program, leaving the algorithmic part of the code unchanged.

The experiences in developing these algorithms have led us to the formulation of guidelines for parallel program development using evaluation strategies [LT97]. The outlined methodology makes use of several components of our parallel engineering environment such as a sequential profiler to detect expensive components in the program [SP97], a simulator, which allows both idealised and realistic simulation of the target architecture [HLP95], a set of visualisation tools for examining the behaviour of the program on several levels of detail, and a portable parallel runtime system [THM+96]. The integration of these components into a single parallel engineering environment facilitates the performance tuning stage and encourages the programmer to experiment with different forms of parallel behaviour.

References

[BCLA82] B. Buchberger, G. E. Collins, R. Loos, and R. Albrecht, editors. *Computer Algebra — Symbolic and Algebraic Computation*, chapter Computing by Homomorphic Images, by M. Lauer, pages 139–168. Springer, Vienna, second edition, 1982.

[HL94] H. Hong and H-W. Loidl. Parallel Computation of Modular Multivariate Polynomial Resultants on a Shared Memory Machine. In B. Buchberger and J. Volkert, editors, *CONPAR'94 — Conference on Parallel and Vector Processing*, volume 854 of *Lecture Notes in Computer Science*, pages 325–336, Linz, Austria, September 6–8, 1994.

[HLP95] K. Hammond, H-W. Loidl, and A. Partridge. Visualising Granularity in Parallel Programs: A Graphical Winnowing System for Haskell. In *HPFC'95 — High Performance Functional Computing*, pages 208–221, Denver, Colorado, April 10–12, 1995.

[Knu81] D.E. Knuth. *The Art of Computer Programming*, volume Vol. II: Seminumerical Algorithms. Addison-Wesley, 2nd edition edition, 1981.

[Lip71] J. D. Lipson. Chinese Remainder and Interpolation Algorithms. In *SYMSAM*, pages 372–391, 1971.

[LL93] C. Limongelli and H-W. Loidl. Rational Number Arithmetic by Parallel P-adic Algorithms. In J. Volkert, editor, *ACPC'93 — Parallel Computation — Second Intl. ACPC Conference*, volume 734 of *Lecture Notes in Computer Science*, pages 72–86, Gmunden, Austria, October 4–6, 1993. Austrian Center for Parallel Computation (ACPC), Springer-Verlag.

[LMT+97] H-W. Loidl, R. Morgan, P. Trinder, R. Garigliano, S. Peyton Jones, and C. Cooper. Parallelising a Large Functional Program; or Keeping Lolita Busy. In *IFL'97 — Intl. Workshop on the Implementation of Functional Languages*, September 10–12, St. Andrews, Scotland, 1997.

[LT97] H-W. Loidl and P. Trinder. Engineering Large Parallel Functional Programs. In *IFL'97 — Intl. Workshop on the Implementation of Functional Languages*, September 10–12, St. Andrews, Scotland, 1997.

[MKH91] E. Mohr, D.A. Kranz, and R.H. Halstead Jr. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, July 1991.

[Roe91] P. Roe. *Parallel Programming using Functional Languages.* PhD thesis, Department of Computing Science, University of Glasgow, February 1991.

[SP97] P.M. Sansom and S.L. Peyton Jones. Formally based profiling for higher-order functional languages. *ACM Transactions on Programming Languages and Systems*, 19(2):334–385, March 1997.

[THL+96] P.W. Trinder, K. Hammond, H-W. Loidl, S.L. Peyton Jones, and J. Wu. A Case Study of Data-intensive Programs in Parallel Haskell. In *Glasgow Workshop on Functional Programming*, Workshops in Computing, Ullapool, Scotland, July 8–10, 1996. Springer-Verlag.

[THLP98] P.W. Trinder, K. Hammond, H-W. Loidl, and S.L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1), 1998. To appear.

[THM+96] P. Trinder, K. Hammond, J.S. Mattson Jr., A.S Partridge, and S.L. Peyton Jones. GUM: a Portable Parallel implementation of Haskell. In *PLDI'96 — Programming Languages Design and Implementation*, pages 79–88, Philadelphia, USA, May 1996.