Parallel Processing Letters

TUNING TASK GRANULARITY AND DATA LOCALITY OF DATA PARALLEL GPH PROGRAMS

HANS-WOLFGANG LOIDL, PHILIP W. TRINDER and CARSTEN BUTZ
Department of Computing and Electrical Engineering,
Heriot-Watt University, Edinburgh EH14 4AS, Scotland, U.K.
E-mail: {hwloidl, trinder, carsten}@cee.hw.ac.uk

ABSTRACT

The performance of data parallel programs often hinges on two key coordination aspects: the computational costs of the parallel tasks relative to their management overhead — *task granularity*; and the communication costs induced by the distance between tasks and their data — *data locality*. In data parallel programs both granularity and locality can be improved by *clustering*, i.e. arranging for parallel tasks to operate on related sub-collections of data.

The GpH parallel functional language automatically manages most coordination aspects, but also allows some high-level control of coordination using *evaluation strategies*. We study the coordination behavior of two typical data parallel programs, and find that while they can be improved by introducing clustering evaluation strategies, further performance improvements can be achieved by restructuring the program.

We introduce a new generic `Cluster` class that allows clustering to be systematically introduced, and improved by program transformation. In contrast to many other parallel program transformation approaches, we transform realistic programs and report performance results on a 32-processor Beowulf cluster. The cluster class is highly-generic and extensible, amenable to reasoning, and avoids conflating computation and coordination aspects of the program.

Keywords: Data parallelism, functional programming, program derivation.

1 Introduction

One of the prime attractions of parallel functional languages is their high-level, often dynamic, and substantially-implicit description of parallelism. This eases the development of an initial parallel version of a program. But how easy is it to tune the parallel performance of such programs? In previous work we have parallelized and tuned the performance of several large programs in Glasgow parallel Haskell (GpH) [1]. In most cases we managed to improve parallel performance by adding evaluation strategies. However, in a few cases simple code restructuring was necessary in order to increase task granularity or improve data locality. This repeated *ad hoc* program transformation motivated our search for a systematic means to improve these two coordination aspects.

The naive data parallelization of a collection such as a list evaluates every element of the collection in parallel and often yields very fine task granularity. Clustering improves task granularity and data locality by introducing fewer tasks, each operating on a closely-related subset of the collection, e.g. on sub-lists. In order to facilitate code reuse we present a generic way of clustering collections that obey certain algebraic properties. Our design uses Haskell constructor classes, and introduces a new `Cluster` class that extends the well-known monad class. We can therefore make use of a rich set of existing identities in proving identities about functions over clustered data (Section 3). We demonstrate our mechanism on two representative data parallel programs over lists and matrices (Section 4) and report good performance improvements on a 32-node Beowulf-class parallel machine. A third clustered program is discussed in the conclusions. The clustering class and associated identities have the following properties.

Highly Generic: Clustering instances may be defined for *many collection types*, e.g. Section 4 gives examples of instances for lists and matrices represented as lists of lists. A collection type may be clustered with *many granularities*, e.g. a list may be clustered into 5-element or 100-element chunks in the same program. Moreover, it is possible to have *alternate clustering* instances for the same collection type within different modules of the same program, e.g. Section 4.2 gives an example of row and block clustering for matrices.

Extensible: A `Cluster` instance can be defined for a new collection type, or a new application-specific `Cluster` instance can be defined for an existing collection type, examples are given in Section 4.

Amenable to reasoning: The BMF-style identities in Section 3 allow the derivation of a clustered function from an unclustered function and further improvement; example derivations are given in Sections 4.1 and 4.2.

Practical: The clustering class is implemented in GpH and used to improve the parallel performance of non-trivial parallel programs. Section 4 gives performance measurements on a Beowulf platform for both programs.

Separates computation and coordination: It is not necessary to construct special computational functions to operate on clustered data: functions with well-defined properties with respect to the underlying collection type can be automatically lifted to operate over a clustered collection without change (Section 3). This is consistent with our evaluation strategy ethos of cleanly separating the concerns of computation and coordination [2].

2 GpH and Evaluation Strategies

2.1 GpH *Parallelism*

GpH [2] is a modest conservative extension of Haskell [3] using the combinators `seq` and `par` for specifying sequential and parallel composition, respectively. The expression `` p `par` e `` has the same value as `e`. Its coordination effect is to indicate that `p` could be evaluated by a new parallel thread, with the parent thread continuing evaluation of `e`. Since the thread is not necessarily created, `p` is similar to a *lazy future* [4]. The coordination effect of the expression `` e1 `seq` e2 ``, which denotes the value of `e2`, is to evaluate `e1` to *weak head normal form* (WHNF) before returning `e2`. Note that `par` is non-strict in its first argument, whereas `seq` is strict.

2.2 Evaluation Strategies

Evaluation strategies, introduced in [2], use lazy higher-order functions to separate the two concerns of specifying the computation and specifying the coordination. A function definition is split into two parts, the computation (or algorithm) and the coordination (or strategy), with values defined in the former being manipulated in the latter. The description of the computation is consequently uncluttered by the coordination.

A strategy is a function that specifies the coordination required when computing a value of a given type. A strategy is evaluated purely for effect, and hence it returns just the nullary tuple `()`.

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

The simplest strategies introduce no parallelism: they specify only the evaluation degree. The default evaluation degree in GPH is WHNF and therefore 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. We make use of overloading in Haskell by defining a type class `NFData` with a function `rnf`, performing a reduction to normal form.

```
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 often relies on its element types, if any, being in class `NFData`, and the instance for lists is given below.

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

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. For example the `parList` strategy below takes a strategy (`strat`) as a parameter and applies it 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)
```

Strategies are applied by the `using` function which applies the strategy `strat` to the data structure `x` before returning it.

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

For example the parallel map function, `parMap`, defined below applies its function argument to every element of a list in parallel.

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

3 A Generic Approach to Data Clustering

3.1 Strategic Clustering

```
parListChunk :: Int -> Strategy a -> Strategy [a]
parListChunk n strat [] = ()
parListChunk n strat xs = seqList strat (take n xs) `par`
                          parListChunk n strat (drop n xs)
```

Fig. 1: A clustering strategy used in `sumEuler`

Clustering is easily introduced into functions returning a collection that is suitable for constructing a clustered process network. For example parallel threads to operate on subsequences, or *chunks*, of a list can be introduced by the `parListChunk` strategy shown in Figure 1. This strategy specifies parallel evaluation of sub-lists of length `n`. As `seqList` is analogous to `parList` above, `seqList strat (take n xs)` performs a sequential evaluation of the first `n` list elements, applying strategy `strat` to every element. For example the expression `map f xs` can be clustered as follows.

```
parMapChunk n strat f xs = map f xs `using` parListChunk n strat
```

The advantage of a purely strategic approach is that the clustered coordination is captured entirely by the evaluation strategy and isolated from the computation.

3.2 Cluster Class

Strategies operate on the result of a function, but unfortunately many functions have no collection result. A common example is a *fold* that collapses a collection to a single value. To cluster such functions, we introduce a generic *Cluster* class with functions to *cluster* the input data, *lift* the function to operate on the clustered data, and sometimes *decluster* the result. Although clustering changes the computation component of a parallel program, equivalence to the sequential program is maintained by introducing clustering systematically using semantics-preserving identities.

To be more precise, we require that every collection type c that is to be clustered is a monad, as shown by the subclass dependency for *Cluster* below. We use a formulation of monads based on the functions *munit*, *mjoin* and *mmap* [5], which is more suitable for our purposes than the Kleisli category [6][Section 10.4] with *return* and *bind* used in the standard Haskell libraries.

```
class MMonad c where
  munit  :: a -> c a
  mjoin  :: c (c a) -> c a
  mmap   :: (a -> b) -> (c a -> c b)
```

We introduce a new Haskell class, *Cluster*, parameterized by the collection type c with three operations: *cluster* n maps a collection into a collection of sub-collections each of size n; *decluster* flattens a collection of collections into a single collection; and *lift* takes a function on $c\ a$ and applies it to a collection of collections. For *decluster* and *lift* we use existing definitions in the *MMonad* class.

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

  decluster = mjoin
  lift      = mmap
```

From the monad identities [5][Rules (I)–(III),(i)–(iv)] we obtain the following equations relating *lift*, *decluster*, and *munit*.

$$
\begin{aligned}
\mathit{decluster} \circ \mathit{munit} &= \mathit{id} && \text{(M I)}\\
\mathit{decluster} \circ \mathit{lift}\ \mathit{munit} &= \mathit{id} && \text{(M II)}\\
\mathit{decluster} \circ \mathit{decluster} &= \mathit{decluster} \circ \mathit{lift}\ \mathit{decluster} && \text{(M III)}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{lift}\ \mathit{id} &= \mathit{id} && \text{(M i)}\\
\mathit{lift}\ (f \circ g) &= (\mathit{lift}\ f) \circ (\mathit{lift}\ g) && \text{(M ii)}\\
\mathit{lift}\ f \circ \mathit{munit} &= \mathit{munit} \circ f && \text{(M iii)}\\
\mathit{lift}\ f \circ \mathit{decluster} &= \mathit{decluster} \circ \mathit{lift}\ (\mathit{lift}\ f) && \text{(M iv)}
\end{aligned}
$$

We further require for each n that *cluster* n is a one-sided inverse of *decluster*.

$$
\mathit{decluster} \circ \mathit{cluster}\ n = \mathit{id} \qquad \text{(C I)}
$$

We now examine the properties of functions that modify the structure of the base domain. We call a function $\mathit{malg} :: c\ a \to a$ an (Eilenberg-Moore) algebra for the monad c if the following two identities hold (Section 3.4 examines these properties for functions over lists).

$$
\begin{aligned}
\mathit{malg} \circ \mathit{munit} &= \mathit{id} && \text{(A I)}\\
\mathit{malg} \circ \mathit{mmap}\ \mathit{malg} &= \mathit{malg} \circ \mathit{mjoin} && \text{(A II)}
\end{aligned}
$$

We can depict the identities for an algebra as the following commuting diagrams.

$$
\begin{tikzcd}
a \arrow[r, "\mathit{munit}"] \arrow[dr, "\mathit{id}"'] & c\ a \arrow[d, "\mathit{malg}"] \\
& a
\end{tikzcd}
\qquad
\begin{tikzcd}
c\ (c\ a) \arrow[r, "\mathit{mmap}\ \mathit{malg}"] \arrow[d, "\mathit{mjoin}"'] & c\ a \arrow[d, "\mathit{malg}"] \\
c\ a \arrow[r, "\mathit{malg}"] & a
\end{tikzcd}
$$

Given two algebras $\alpha :: c\ a \to a$ and $\beta :: c\ b \to b$, a *homomorphism* between them is a function $f :: a \to b$ such that $f \circ \alpha = \beta \circ \mathit{mmap}\ f$.

3.3 Transforming Clustered Programs

The categorical identities on collections are useful for transforming clustered programs. This section discusses the two main identities we use, (lift1) and (lift2), and shows that they follow from those of monads and algebras stated in the previous section, together with the identity for *cluster*.

We note that, for every a, the function $\mathit{mjoin} :: c\ (c\ a) \to c\ a$ is an algebra for the monad c (called the *free* algebra on a), and that, for every $f :: a \to b$, $\mathit{mmap}\ f$ is an algebra homomorphism between these free algebras.

Clustering can be introduced into a program by the following identity that holds for algebra homomorphisms, $f :: c\ a \to c\ b$. Algorithmically the right-hand side splits the input into clusters, applies f to every cluster and flattens the result.

$$f \quad = \quad \mathit{decluster} \circ \mathit{lift}\ f \circ \mathit{cluster}\ n \tag{lift1}$$

$$\begin{tikzcd}
c\ (c\ a) \arrow[r, "\mathit{lift}\ f"] & c\ (c\ b) \arrow[d, "\mathit{decluster}"] \\
c\ a \arrow[u, "\mathit{cluster}\ n"] \arrow[r, "f"] & c\ b
\end{tikzcd}$$

Since f is an algebra homomorphism $\mathit{decluster} \circ \mathit{lift}\ f = f \circ \mathit{decluster}$ (using that $\mathit{mmap} = \mathit{lift}$ and $\mathit{mjoin} = \mathit{decluster}$), and (lift1) follows from (C I).

The (A II) identity can be used as the following rewrite rule in order to apply the function *malg* to all clusters before combining the results using *malg* again.

$$\mathit{malg} \quad = \quad \mathit{malg} \circ \mathit{lift}\ \mathit{malg} \circ \mathit{cluster}\ n \tag{lift2}$$

Again, this identity has an easy proof. Since *malg* is an algebra for the monad the identity $\mathit{malg} \circ \mathit{lift}\ \mathit{malg} = \mathit{malg} \circ \mathit{decluster}$ holds (using as above that $\mathit{mmap} = \mathit{lift}$ and that $\mathit{mjoin} = \mathit{decluster}$), and (lift2) follows again from (C I).

3.4 Lists as an Example of a Monad

In this section we recapitulate the list monad and the notion of its algebras. The monad instance for lists uses the prelude functions *concat* and *map*.

```
instance MMonad [] where
  munit x = [x]
  mjoin   = concat
  mmap    = map
```

The proofs of the basic monad identities for these functions are standard (see for example [7][Section 10.3]). *Every* monad c comes equipped with the notion of (Eilenberg-Moore) algebras [6][Section 10.4], which are functions $c\ a \to a$ satisfying identities (A I) and (A II) above. For the list monad, Eilenberg-Moore algebras $\mathit{malg} :: [a] \to a$ are in one-to-one correspondence with monoid structures on a. Algorithmically this property, also called First Homomorphism Theorem [8] for lists, states that any Eilenberg-Moore algebra can be written in a fold-of-map structure. This result has been generalized to categorical data types [9,10], and is also known as the factorization of catamorphisms [11].

For example, for one direction of the above correspondence, a monoid $(a, \oplus, 0)$ gives rise to an algebra $[a] \to a$ satisfying (A I) and (A II) by setting

$$\mathit{malg} \ = \ \mathit{fold}\ \oplus\ 0\,.$$

We refer to [12] for a detailed discussion of the recursion operator *fold*.

Along this correspondence of Eilenberg-Moore algebras for the list monad and monoids, our notion of algebra homomorphism introduced above corresponds exactly to homomorphisms of monoids, as the reader will easily verify.

Of particular interest are the so-called free algebras for the list-monad, which are the algebras $concat :: [[a]] \to [a]$. Any function $f :: a \to b$ gives rise to an algebra homomorphism $map\ f$ between such algebras.

4 Example Programs

In this section we apply the generic clustering techniques, developed in the previous sections, to two typical data parallel programs. The programs use lists and matrices, represented as lists of lists, that satisfy the `MMonad` properties we require [5]. All measurements use our implementation of GUM 4.06 [13], a parallel graph reduction machine built on top of the Glasgow Haskell Compiler. We present performance results on a 32-node Beowulf cluster [14] consisting of Linux RedHat 6.2 workstations with a 533MHz Celeron processor, 128kB cache, 128MB of DRAM and 5.7GB of IDE disk. The workstations are connected through a 100Mb/s fast Ethernet switch with a latency of 142μs, measured under PVM 3.4.2.

4.1 Example I: sumEuler

4.1.1 The Sequential Version

```
euler :: Int -> Int
euler n = length (filter (relprime n) [1..(n-1)])

sumEuler :: Int -> Int
sumEuler = sum . map euler . mkList
```

Fig. 2: Sequential version of `sumEuler`

The first program we use to demonstrate clustering is `sumEuler`, as shown in Figure 2. It is a very simple program, exhibiting a general fold-of-map structure. It computes the sum over the application of the Euler totient function (`euler`) over an integer list of length `n`. The expression `euler n` computes the number of integers that are relatively prime (`relprime`) to `n`.

4.1.2 A Strategic Clustering Version

In the map phase of `sumEuler` it is easy to exploit data parallelism, by computing every `euler` function in parallel. However, such an unclustered version yields a large number of very fine grained threads, resulting in a speedup close to one. For example, for the list of integers from 1 to 5,000 as input, the overall runtime is 92.4s yielding an average of 18.5ms per function application.

In order to improve the granularity of the algorithm we arrange for a whole list segment to be computed by just one task. The resulting *"strategic clustering version"* of `sumEuler` is shown in Figure 3. It takes the cluster size `z` as an additional

```
sumEuler :: Int -> Int -> Int
sumEuler z n  = sum ( map euler (mkList n)
                      `using`
                      parListChunk z rnf )
```

Fig. 3: Strategic clustering version of `sumEuler`

parameter and adds the evaluation strategy `parListChunk` from Section 3.1 to the inner expression, which generates the list of result values that should be added.

Unfortunately, the parallel performance of the resulting algorithm is rather unsatisfactory. Sufficient parallelism is generated early on in the program, up to about one fifth of the computation. After that the sequential fold at the end of the computation becomes a serious bottleneck yielding a poor average parallelism of only 3.2 on 20 processors with a total runtime of 24,239ms.

In summary, we can identify two reasons for the poor performance of this parallel program. Firstly, the entire sum operation is sequential and therefore bound to generate some sequential tail in the computation, despite a small amount of overlapping computations of the map and the sum operations. Secondly, the result of each parallel thread is a list, and therefore requires a significant amount of communication. Even more importantly one such result list may have to be sent in several packets of data, if the thread generating the list is still active upon receiving a request for the result data.

4.1.3 A Generic Clustering Version

The derivation of a generic clustering version of `sumEuler` proceeds as follows, using that `sum`, defined in the Haskell prelude as `fold (+) 0`, is an algebra for the `List` monad, and that `map euler` is an algebra homomorphism between free algebras.

$$
\begin{aligned}
sumEuler &= sum \circ map\ euler && \text{(unfold)}\\
&= sum \circ decluster \circ lift\ (map\ euler) \circ cluster\ z && \text{(lift1)}\\
&= sum \circ lift\ (sum) \circ lift\ (map\ euler) \circ cluster\ z && \text{(A II)}\\
&= sum \circ lift\ (sum \circ map\ euler) \circ cluster\ z && \text{(M ii)}
\end{aligned}
$$

```
sumEuler :: Int -> Int -> Int
sumEuler z n = sum ((lift worker) (cluster z (mkList n))
                    `using` parList rnf)
               where worker = sum . map euler
```

Fig. 4: Clustered version of `sumEuler`

Figure 4 shows the transformed `sumEuler` code, which retains some separation between the algorithmic and coordination code. In particular the sequential code can be used as a `worker` function, and the clustering operations are "wrapped" around the (lifted) worker in order to achieve an efficient parallel version.

This version exhibits a much improved average parallelism of 12.3 on 20 processors with a total runtime of 5,777ms. The sequential tail in the computation has been vastly reduced, to only about 900ms (15.6% of the total runtime), compared to about 19s (78.4% of the total runtime) in the strategic clustering version. On larger parallel machines it might be advantageous to use a different clustering scheme, which collects every z-th element of the list into one block in order to achieve a better load balance. Such a change would effect only the definition of `(de)cluster` but not the code of `sumEuler` itself.

4.1.4 Performance Results

The left hand graph in Figure 5 compares the number of packets sent in the strategic and the generic clustering versions of `sumEuler` (note the log-scale on the y-axis). It highlights the fact that the strategic version performs much more communication in total: for 16 processors 14,580 vs. 498 packets. A good measure for the communication degree in a program is the number of packets sent per second of execution time: the strategic version sends 455 packets per second, whereas the clustering version sends only 93 packets per second. This behavior is explained by the fact that the strategic version has to send entire lists as the results of one task, whereas the generic clustering version only has to send one integer, the sum of the sub-list, as a result.

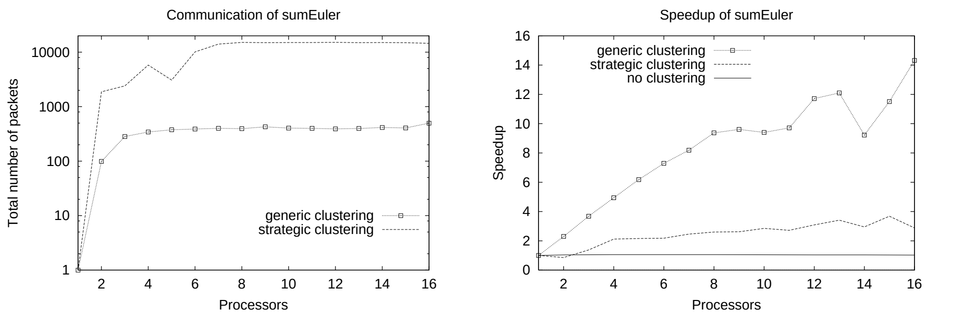

Fig. 5: Number of packets (left) and speedup (right) in strategic and generic clustering versions of `sumEuler`

The right hand graph in Figure 5 compares the relative speedups of the unclustered, the strategic and generic clustering versions of `sumEuler`. The unclustered version produces hardly any speedup at all. The strategic version produces only speedups up to 3.7, mainly owing to the sequential sum operation at the end, whereas the improved generic clustering version shows a speedup of 14.3 on 16 processors. The latter version, however, shows for some numbers of processors uncharacteristically low speedups. This is due to bad automatic task placement, which is more likely to happen when the number of parallel tasks is close to the number of processors. We have observed bad task placement before and plan to add task migration to our system in order to alleviate this general problem.

4.2 Example II: Matrix Multiplication

4.2.1 Sequential Version

In this section we investigate parallel implementations of matrix multiplication of integers, i.e. given $A, B \in \mathbb{Z}^{n \times n}, n \in \mathbb{N}$ find $A * B \in \mathbb{Z}^{n \times n}$. We represent a matrix as a list of lists. In order to overload the functions in the class `Num`, such as multiplication, we introduce a type constructor `M`. In order to define strategies over any element type of a matrix we require this type to be an instance of the class `NFData`.

```
data (Num a, NFData a) => Matrix a = M [[a]]
```

The code for sequential matrix multiplication in Figure 6 is the usual double nested loop, expressed as a list comprehension, computing a cross product (`multVec`) in each step. We use an auxiliary function `multMat2`, which performs matrix multiplication on a pair of matrices with the second matrix already transposed.

```
multMat :: (Num a, NFData a) => Matrix a -> Matrix a -> Matrix a
multMat (M m1) (M m2) = M (multMat2 (m1, (transpose m2)))

multMat2 :: (Num a) => ([[a]], [[a]]) -> [[a]]
multMat2 (m1, m2) = [ [multVec row col | col <- m2] | row <- m1]

multVec :: (Num a) => [a] -> [a] -> a
multVec v1 v2 = sum (zipWith (*) v1 v2)
```

Fig. 6: Sequential matrix multiplication

4.2.2 Unclustered Parallel Version

The unclustered parallel version exploits parallelism for every element of the result matrix. To achieve this behavior we add a strategy to the top-level list expression, as shown in Figure 7. The first two components cause the input matrices `m1` and `m2` to be fully evaluated. Although, this introduces some sequentialization in the beginning, it avoids competition for the unevaluated input structures among the parallel tasks generated by the matrix multiplication itself. The third component, `parList (parList rnf) m`, specifies that every element in the result matrix should be evaluated in parallel. Therefore, this version will generate up to n^2 parallel tasks.

```
multMatPar :: (Num a, NFData a) => Matrix a -> Matrix a -> Matrix a
multMatPar m1 m2 = multMat m1 m2
                    `using` \ (M m) -> rnf m1 `seq`
                                       rnf m2 `seq`
                                       parList (parList rnf) m
```

Fig. 7: Parallel unclustered matrix multiplication

4.2.3 Row-wise Clustered Parallel Version

In order to perform a row-wise clustering of the parallel operations we can use the `parListChunk z rnf` strategy of Section 3.1 on the result matrix. This strategy causes clusters of z rows to be evaluated to normal-form in parallel. It leaves the algorithmic code unchanged and requires only a change in the top level definition of the matrix multiplication, as shown in Figure 8.

```
multMatPar :: (Num a,NFData a) => Int -> Matrix a -> Matrix a -> Matrix a
multMatPar z m1 m2 = multMat m1 m2
                     `using`
                     \ (M m) -> rnf m1  `seq`
                               rnf m2  `seq`
                               parListChunk z rnf m
```

Fig. 8: Parallel row-wise clustered matrix multiplication

4.2.4 Block-wise Clustered Parallel Version

The block-wise clustered version first transforms the input matrices into matrices of matrices ("blocks"), then applies the sequential matrix multiplication and finally declusters the result matrix. One big advantage of this version is the reduced amount of communication owing to the block-clustering. In order to compute the block at position (i, j) in the result matrix only the blocks of the i-th row of A and of the j-th column of B are needed.

As notational convention we use in the following derivation subscripts to parameterize functions over overloaded operations such as addition and multiplication. We define $\boxplus = \mathit{map}\ \oplus\ \circ\ \mathit{zip}$, i.e. component-wise application of $\oplus$ over lists.

The denotational equivalence of sequential and block-wise matrix multiplication is common knowledge in the parallel programming community [15][Chapter 7]. We therefore focus on the systematic introduction of clustering, rather than giving a complete derivation. Corresponding to the identity (lift2), which allows us to defer declustering, we require one basic identity (lift3) relating clustering with the basic arithmetic functions in order to derive a block-wise clustering version.

$$f_{\oplus,\otimes}\ \circ\ \mathit{decluster}\quad=\quad\mathit{decluster}\ \circ\ f_{\boxplus, f_{\oplus,\otimes}} \tag{lift3}$$

Note that the *decluster* on the left hand side works component-wise over pairs. In the case of $f :: (c\ a,\ c\ a) \to c\ a$ being matrix multiplication the function has the following structure.

$$\begin{array}{rcl} f_{\oplus,\otimes}\ (m_1, m_2) & = & \mathit{map}\ (\lambda x \to \mathit{map}\ (\lambda y \to \mathit{multVec}_{\oplus,\otimes}\ (x, y))\ m_2)\ m_1 \\ & & \texttt{where}\ \mathit{multVec}_{\oplus,\otimes} = \mathit{fold}\ \oplus\ 0\ \circ\ \mathit{map}\ \otimes\ \circ\ \mathit{zip} \end{array}$$

We can exploit the fact that the binary operation $\oplus$ is associative with a zero, and that $\otimes$ is associative. Note that in this notation `multVec` in Figure 6 is $\mathit{multVec}_{+,\times}$ with $+, \times$ over integers. By definition we have $\mathit{zipWith}\ \otimes = \mathit{map}\ \otimes\ \circ\ \mathit{zip}$.

The following derivation shows how to use the identities, including the as yet unproven identity (lift3) in the main step, in order to arrive at the block-clustered version of the code. We write m^T for *transpose* m and $+, \times$ as addition, multiplication over integers, $\boxplus$ as addition over matrices of integers.

$$
\begin{array}{lll}
\multicolumn{2}{l}{\mathit{multMat}\ (m_1, m_2)} & \\
= & \mathit{multMat2}_{+,\times}\ (m_1, m_2^T) & \text{(unfold)} \\
= & \mathit{multMat2}_{+,\times} \circ \mathit{decluster} \circ \mathit{cluster}\ z\ (m_1, m_2^T) & \text{(C I)} \\
= & \mathit{multMat2}_{+,\times} \circ \mathit{decluster} \circ (\mathit{cluster}\ z\ m_1, \mathit{cluster}\ z\ m_2^T) & \text{(cluster (,))} \\
= & \mathit{decluster} \circ \mathit{multMat2}_{\boxplus, \mathit{multMat2}_{+,\times}} \circ (\mathit{cluster}\ z\ m_1, \mathit{cluster}\ z\ m_2^T) & \text{(lift3)}
\end{array}
$$

Note that in the last step, where we apply (lift3), the matrix multiplication of integers is replaced by one over matrices over integers, indicated by the different functions used as subscripts.

```
multMatPar :: (Num a,NFData a) => Int -> Matrix a -> Matrix a -> Matrix a
multMatPar z m1 m2 =
  decluster (multMat2 (cluster z m1, cluster z (transposeMat m2))
             `using` \ (M m) -> rnf m1 `seq`
                                  rnf m2 `seq`
                                  parList (parList rnf) m )
```

Fig. 9: Block-wise clustered matrix multiplication

Figure 9 shows the block-wise clustered version. The (de-)clustering functions (code omitted) use the list versions of (de-)clustering twice, together with a transpose, to generate a matrix of matrices and are defined component-wise on pairs.

4.2.5 Performance Results

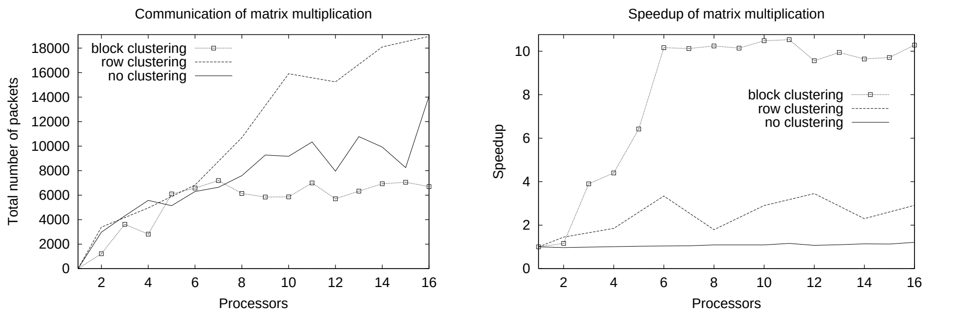

Fig. 10: Number of packets (left) and speedup (right) in unclustered and block-wise clustered matrix multiplication

Figure 10 summarizes the performance results on up to 16 processors for all three versions of the parallel matrix multiplication. The input values are two 300×300 matrices over arbitrary precision integers. The cluster size is 15 for row-wise clustering and 60 for block-wise clustering, i.e. 25 blocks of size 60×60 are generated in the third version. The speedup graphs on the right hand side of Figure 10 illustrate the importance of introducing clustering in order to utilize effectively a high-latency machine. The unclustered version produces hardly any speedup at all. This is mainly due to the overhead of managing (potentially) 90,000 threads, one for each element in the result matrix. Since each thread only computes the cross product of two vectors, the relative overhead of generating a thread, with an independent stack, is very high. The row-wise clustering version improves the task granularity by producing only 20 tasks in total. However, the communication is still very high, in fact even higher than in the unclustered version, because each task needs the entire matrix B in order to compute a complete row. Finally, the block-wise clustering algorithm reduces the total amount of communication by computing blocks, rather than entire rows.

The superlinear speedup of the block-clustering version is due to reduced garbage collection costs in the parallel version. Each processor can use as much heap as one sequential instantiation of the program. Provided that the algorithm exhibits good data locality and only a small amount of data is duplicated during the computation, fewer garbage collections will be necessary. The better data locality in the block clustered version compared to the row clustered version is shown by the smaller number of global pointers in the former: 137 vs. 387 on average for 16 processors.

5 Related Work

Our clustering approach shows strong similarities to *skeleton-based parallel programming* [16], exemplified by systems such as SCL [17], P3L [18], SkelML [19] or Skil [20], with an emphasis on compositionality and genericity. Compared to skeletons we trade an optimized parallel implementation of some higher-order functions for a more flexible framework where new coordination constructs, and their granularity and locality can be defined. Our use of program derivation for improving parallel performance resembles the derivation of skeleton code in SCL [21].

Closely related to evaluation strategies are *first class schedules* [22]. They use Haskell monads, with the usual bind and return operators, in order to integrate operating system interaction, e.g. for obtaining load information, with purely functional computations and to construct virtual topologies of processors. Since GpH does not provide placement constructs we do not construct topologies, but tune data locality by combining computations on different parts of a data structure.

The usefulness of basic categorical properties in order to generalize results from Bird's theory for lists [8] has been shown by Spivey [9]. Motivated by this observation language designs based on the use of *categorical data types* have been developed [10]. The use of categorical data types for specifying data parallel programs is discussed by Skillicorn [11]. Concrete examples of using these properties to improve parallel performance via semantics preserving program transformation are given for example in [23,11]. In a similar spirit, shapely types separate the shape from the contents of a data type enabling a polymorphic use of functions over different shapes and the development of parallel cost models [24]. However, to the best of our knowledge our generic formulation of clustering, based on category theory, together with its parallel implementation are novel.

At the moment, GPH lacks a well-defined *cost model*, which would permit formal reasoning about the performance, as well as the correctness, of clustered parallel programs. Compared to the well-developed cost models in the skeleton community [24,25], a cost model for GPH is complicated by its non-strict semantics, which makes it harder to predict computation costs. Consequently, only few cost models have been developed for non-strict languages in general [26,27]. The main problem is to capture the notion of demand on a program expression, e.g. by using projections [28], and to use this information in assessing the evaluation degree of a program expression in a particular context. In the longer term we plan to develop a cost model based on our operational semantics for GPH [29]. A detailed survey of cost analysis for functional languages is given in [30][Section 6.7].

In clustering larger programs the *reclustering* of collections between separate stages of the computation will become important. In order to provide such possibilities in our Haskell class-based approach we plan to use techniques developed in SCL as configuration skeletons [21] and possibly data distribution algebras [31,32] and their compositional properties.

6 Conclusion

In this paper we have introduced a generic method for tuning task granularity and spatial data locality in data parallel GPH programs. Our method reduces the programmer's coding effort and proof obligations by exploiting category theoretic properties of collection types. In using rigorous program transformation, and maintaining a separation between algorithmic and clustering code, the correctness of the resulting optimized program is easier to assert than in most parallel programming languages. We consider program derivation based on the `Cluster` class as an advanced tool for the expert programmer, in order to optimize a parallel application. In order to achieve moderate performance improvements the addition of evaluation strategies to a few top-level functions is sufficient for most parallel applications. This choice of different levels of detail in specifying a parallel program is one of the major advantages of GPH over more conventional parallel programming languages.

We have applied our method of data clustering to two typical data parallel programs and assessed the concrete performance benefit in our implementation of GPH on a high-latency Beowulf machine. The performance improvements show the importance of controlling task granularity and data locality, which we do in a high-level, architecture-independent way. For a simple fold-of-map program, `sumEuler`, the speedup increases from 3.7 to 14.3 on 16 processors. For a parallel matrix multiplication a slowdown in the unclustered version is turned into a speedup of 10.3 on 16 processors. Space precludes a description of a third program, a simple data-parallel raytracer, which is clustered using a single application of the (lift1) property, and we are currently measuring its performance.

As future work we anticipate applying our clustering method to more GPH programs, many significantly larger than those discussed here. In particular, we want to apply our generic techniques to a range of collection types, for example graph structures as used in computational geometry. In larger programs we will have to tackle the issues of data restructuring and nested clusters. In order to use alternative clusterings of the same collection it is possible to define different instances of `Cluster` in different modules. It remains to be seen how practical such a separation of clusterings along module boundaries is in large applications.

Acknowledgements

The authors would like to express their thanks to the Austrian Academy of Sciences (APART fellowship 624) and UK's EPSRC (research grant GR/M 55633) for their financial support. We also would like to thank Joy Goodman, Rita Loogen, Greg Michaelson, Ricardo Peña Mari, Robert Pointon, Steffen Priebe, Fernando Rubio Diez, and the anonymous referees for helping us to improve the quality of the paper.

References

1. H-W. Loidl, P.W. Trinder, K. Hammond, S.B. Junaidu, R.G. Morgan, and S.L. Peyton Jones. Engineering Parallel Symbolic Programs in GPH. *Concurrency — Practice and Experience*, 11(12):701–752, October 1999.
2. P.W. Trinder, K. Hammond, H-W. Loidl, and S.L. Peyton Jones. Algorithm + Strategy = Parallelism. *J. of Functional Programming*, 8(1):23–60, January 1998.
3. S.L. Peyton Jones and J. Hughes (editors). Haskell98: A Non-strict, Purely Functional Language. <URL:`http://www.haskell.org/`>, February 1999.
4. 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.
5. P. Wadler. Comprehending Monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992.
6. M. Barr and C. Wells. *Category Theory for Computer Science*. Prentice-Hall, 1995.
7. R.S. Bird. *Introduction to Functional Programming using Haskell*. Prentice-Hall, second edition, 1998.
8. R.S. Bird. An Introduction to the Theory of Lists. In *Logic of Programming and Calculi of Discrete Design*, pages 3–42. Springer, 1987.
9. J.M. Spivey. A Categorical Approach to the Theory of Lists. In *Mathematics of Program Construction*, LNCS 375, pages 399–408. Springer, 1989.
10. T. Hagino. A Typed Lambda Calculus with Categorical Type Constructors. In *Category Theory and Computer Science*, LNCS 283, pages 140–157. Springer, 1987.
11. D.B. Skillicorn. *Foundations of Parallel Programming*. Cambridge Intl. Series in Parallel Computation. Cambridge University Press, 1994.
12. G. Hutton. A Tutorial on the Universality and Expressiveness of Fold. *J. of Functional Programming*, 9(4):355–372, July 1999.
13. P.W. Trinder, K. Hammond, J.S. Mattson, A.S. Partridge, and S.L. Peyton Jones. GUM: a Portable Parallel Implementation of Haskell. In *PLDI'96 — Programming Language Design and Implementation*, pages 78–88, Philadephia, USA, May 1996.
14. D. Ridge, D. Becker, P. Merkey, and T. Sterling. Beowulf: Harnessing the Power of Parallelism in a Pile-of-PCs. In *IEEE Aerospace*, 1997.

15. M.J. Quinn. *Parallel Computing.* McGraw-Hill, 1994.

16. M.I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* Research Monographs in Parallel and Distributed Computing. MIT Press, 1989.

17. J. Darlington, Y. Guo, and H.W. To. Structured Parallel Programming: Theory meets Practice. In *Research Directions in Computer Science.* Cambridge University Press, 1996.

18. B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P^3L: A Structured High Level Programming Language and its Structured Support. *Concurrency — Practice and Experience*, 7(3):225–255, May 1995.

19. T.A. Bratvold. *Skeleton-Based Parallelisation of Functional Programs.* PhD thesis, Heriot-Watt University, Edinburgh, U.K., November 1994.

20. G.H. Botorog and H. Kuchen. Skil: An Imperative Language with Algorithmic Skeletons for Efficient Distributed Programming. In *HPDC'96 — Intl. Symp. on High Performance Distributed Computing*, pages 243–252, 1996.

21. Hing Wing To. *Optimising the Parallel Behaviour of Combinations of Program Components.* PhD thesis, Department of Computing, Imperial College, University of London, U.K., September 1995.

22. R. Mirani and P. Hudak. First-Class Schedules and Virtual Maps. In *FPCA'95 — Conf. on Functional Programming Languages and Computer Architecture*, pages 78–85, La Jolla, California, June 1995. ACM Press.

23. D.B. Skillicorn. The Bird-Meertens Formalism as a Parallel Model. In *Software for Parallel Computation*, Cosenza, Italy, June 1992.

24. C.B. Jay, M.I. Cole, M. Sekanina, and P. Steckler. A Monadic Calculus for Parallel Costing of a Functional Language of Arrays. In *EuroPar'97 — European Conf. on Parallel Processing*, LNCS 1300, pages 650–661. Springer, August 1997.

25. D.B. Skillicorn and W. Cai. A Cost Calculus for Parallel Functional Programming. *J. of Parallel and Distributed Computing*, 28(1):65–84, 1994.

26. D. Sands. *Calculi for Time Analysis of Functional Programs.* PhD thesis, Department of Computing, Imperial College, University of London, U.K., September 1990.

27. B. Bjerner and S. Holmström. A Compositional Approach to Time Analysis of First Order Lazy Functional Programs. In *FPCA'89 — Conf. on Functional Programming Languages and Computer Architecture*, pages 157–165. ACM Press, 1989.

28. J. Launchbury and G. Baraki. Representing Demand by Partial Projections. *J. of Functional Programming*, 6(4):563–585, July 1996.

29. C. Baker-Finch, D.J. King, and P.W. Trinder. An Operational Semantics for Parallel Lazy Evaluation. In *ICFP'00 — Intl. Conf. on Functional Programming*, Montreal, Canada, 18–20 September, 2000.

30. H-W. Loidl. *Granularity in Large-Scale Parallel Functional Programming.* PhD thesis, Department of Computing Science, University of Glasgow, U.K., March 1998.

31. M. Südholt. *The Transformational Derivation of Parallel Programs using Data-Distribution Algebras and Skeletons.* PhD thesis, TU Berlin, Germany, August 1997.

32. T. Nitsche. Skeleton Implementations based on Generic Data Distributions. In *CMPP'00 — Intl. Workshop on Constructive Methods for Parallel Programming*, pages 19–34, Ponte de Lima, Portugal, July 2000.