Four Skeletons and a Function

Norman Scaife, Greg Michaelson and Andy Wallace

Department of Computing and Electrical Engineering,
Heriot-Watt University, Riccarton, Edinburgh, EH14 4AS

Abstract. Four algorithmic skeletons for low-level image processing and intermediate-level computer vision are presented, related to a common divide and conquer higher order function. The parallel operation of each skeleton is described and their use in a complete parallel multisource image segmentation system is discussed.

1 Introduction

There is a fundamental tension in programming between specificity and genericity. Specificity is characterised by a low degree of abstraction and is common in code tailored heavily to a particular problem. This can lead to excellent performance but at the cost of code maintenance, modification and reuse. In contrast, genericity is based on a high degree of abstraction and aids maintenance, modification and reuse. However, it can also hinder implementation performance due to impedance mismatches between problems and generic constructs.

A central aim of higher order parallel programming is to optimise the balance between specificity and genericity by offering a practical approach which is well founded in theory. Higher order functions (HOFs) and their skeleton equivalents seem to combine the potential for a high degree of reuse with appropriate granularity for program proof, transformation, performance modelling and implementation [1]. Indeed, a very small set of HOFS - `compose, map, filter, foldl/r` - seem adequate for solving a very wide range of problems.

However, there are practical drawbacks to this basic concept. First of all, a small set of HOFs can seem somewhat limiting, as anyone who has attempted to force a problem into a `fold/map` homomorphism formulation may attest. It becomes tempting to introduce additional HOFs or to craft problem-oriented versions of the canon, thus losing known proof and transformation properties. Secondly, there are different parallel skeletons equivalent to each HOF: their choice depends on a combination of concrete architectural, program and data considerations. Even given such a choice, it may still arise that no extant skeleton for a HOF is optimal for the communication/processing balance for a specific problem. Once again, it becomes tempting to craft a new skeleton or to modify significantly an existing one, thus losing known performance characteristics.

Here we illustrate these issues through four equivalent skeletons for low and intermediate level image processing which are related conceptually to a geometric decomposition HOF; a specialisation of divide and conquer. We then discuss their selection in a substantial image processing system based on the composition of eight skeletons.

2 Geometric decomposition

An image may be regarded as a two dimensional grid of pixels, where a pixel may be simple, for example a grey scale value, or composite, for example an RGB triple. For initial processing, it is common to decompose an image into subimages, process each subimage independently and then resassemble the processed subimages. This corresponds to the well known form of divide and conquer:

```
fun DC divide process conquer =
    conquer o (map process) o divide
```

where `divide` breaks a list into sublists, `process` is applied by `map` to each sublist and the processed sublists are recombined by `conquer`.

`DC` may be specialised by providing domain specific functions:

```
val geomdecomp (dc,dr) ffsubimage =
    DC (calgeomdecomp dc dr)
       (map ffsubimage)
       (flattengeomdecomp dc dr);
```

Here, an image is broken up into list of lists of geometric subimages by a standard `divide` function `calcgeomdecomp`. The granularity of the decomposition is set by the `dc` and `dr` parameters which define how many subdivisions are required along rows and columns. `ffsubimage` is applied to each subimage by `map o map` and the processed regions are reconstituted into the output image by the standard `conquer` function `flattengeomdecomp`.

3 Skeleton design

Were one to develop a skeleton directly from `geomdecomp`, the central `map o map` would be the main locus of parallelism. At first sight, it might be tempting to build on the well known association between `map`s, sequential loops and parallel process(or) farms [2], so `map o map` has potential implementations as sequential nested loops, a parallel farm of sequential loops, a sequential loop of parallel farms or a parallel farm of parallel farms.

However, this approach ignores the problem specific relationships amongst `calcgeomdecomp`, `ffsubimage` and `flattengeomdecomp`. For different image subdivisions, equivalent but different skeletons which reflect specific communication/processing patterns may be more appropriate. We have identified four skeletons for low level image processing which are effectively variants of the parallel farm of sequential loops. These are now described in turn, before being related to each other.

3.1 Separable and regular geometric decomposition (GD).

The first skeleton variant is a geometric decomposition over data where the data is divided up into a number of equal sized sections and each section is processed by a different worker, as shown in Fig. 1.

This skeleton is applicable where the data in each element can be processed independently of the other elements and the amount of work associated with each element is the same. The simple expedient of dividing the data up equally among the workers leads to an optimal solution in terms of communications costs.

Although there is no overlap between the subimages in the illustration here, it may be necessary for window operations to access data from outside the subimage. This presents no problems since processed data is only returned from within the subimages. Since there is no communication between the subimages during processing the processor topology is relatively unimportant, especially when the task messages simply consist of indices into preloaded data.

The decomposition could be of several different types, for instance row-based, column-based or rectangular subimages. The subimages need not cover the entire image. For example a single feature in the image could be treated using a geometric decomposition. In this instance, however, some means of describing the subregions is required, for example through a processor allocation iconic map.

The processes of distributing the input data and gathering the output data need not be included in the skeleton's definition. These operations can be viewed as communications skeletons in their own right and can sometimes cancel each other out for optimisation purposes, for instance when data is gathered and then redistributed immediately.

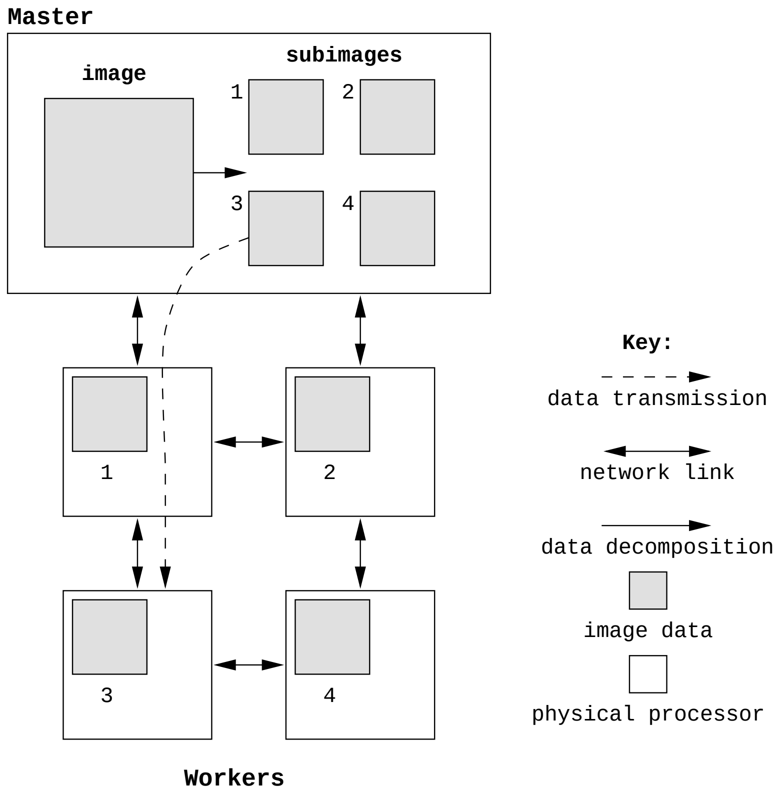

Fig. 1. Operation of the geometric decomposition (GD) skeleton

3.2 Generalised data farming (DF).

The second construct is a general processor farming harness. Several routines require decomposition over irregular data, for instance an arbitrary list of patches of various sizes. This decomposition can be handled by worker farming in which work allocations are doled out by the farmer giving primitive, inherent load balancing. This method is only efficient, however, if there are more data elements than processors and one data element does not dominate all the others.

The data is split up into many more subelements than there are worker processors. The farmer processor transmits work packets on demand until none are left and the workers are no longer processing any data. A worker simply receives a work packet, processes it and returns any output data and a task completion message. Fig. 2 illustrates the basic operation.

The process of modelling this construct will also be slightly different to GD above since a much finer decomposition is required. This will be in a different region of the construct's performance space.

3.3 Task farming with task spawning (TF).

This is a generalisation of the DF skeleton. The same worker farming methodology is used but each task sent to the workers can generate more tasks which are then returned to the master and added to a task queue.

For this skeleton, the data has to be returned upon completion of each task since the output may be required in processing the spawned tasks. Note also that, in some instances, no decomposition is required prior to running this construct since a single initial task can spawn many others. The main disadvantage of this approach is that there can be significant startup and shutdown times if the granularity of the tasks is too coarse and that communication costs become more significant

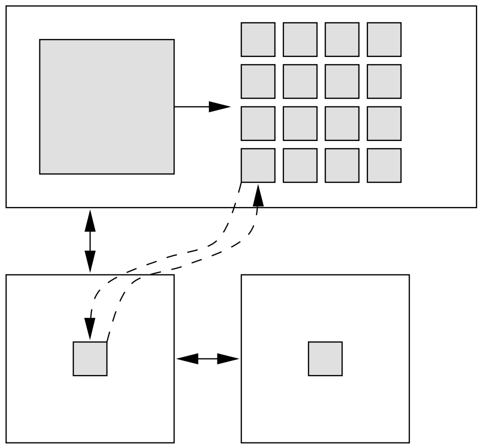

Fig. 2. Operation of the data farm (DF) skeleton

if the granularity is too fine. The efficient use of this skeleton therefore depends on choosing the correct granularity.

There is a problem with this skeleton in terms of representing the generated tasks in a functional language. Such tasks are generated arbitarily and are essentially imperative in nature.

The following shows an SML representation of the TF skeleton:

```
fun taskfarm (dc,dr) ff prune comb img =
  let
    val queue = flatten (calcgeomdecomp dc dr img)
    fun iter [] res = res
      | iter queue res =
        let
          val nq = map (ff res) queue
          fun accum (sft,sfr) (t,r) = (sft @ t,r :: sfr)
          val (tsks,res') = foldl accum ([],res) nq
          val tsks' = filter prune tsks
        in
          iter tsks' res'
        end
  in
    comb (iter queue [])
  end
```

Here, the computation is divided up into discrete iterations whereby all the tasks generated by each iteration are processed in subsequent iterations. The problem of handling generated tasks is achieved by wrapping an iterative layer around the farming operation (`map`). Separate functions are provided for pruning of tasks (`prune`) and for combining the results from the individual elements (`comb`). Note that the growing results list is passed to the task processing function: this is necessary to prevent duplication of work, the pruning function has to eliminate duplicate tasks between iterations.

This alters the order in which the tasks are processed compared to the parallel implementation but should generate the same numbers and types of tasks allowing summation of communications and computational costs for performance modelling. The parallel implementation is illustrated in Fig. 3.

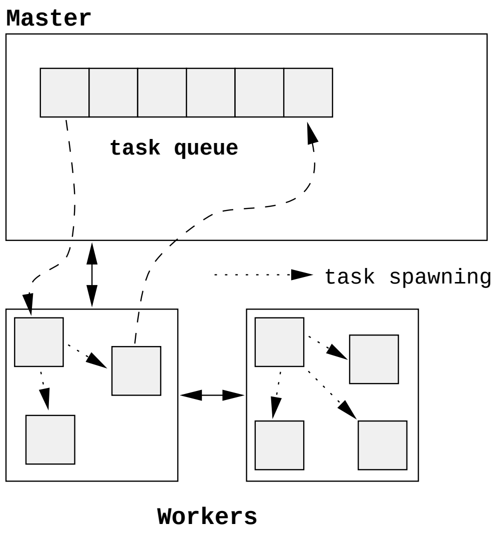

Fig. 3. Operation of the task farm (TF) skeleton

In this skeleton, the data is decomposed into a queue of task definitions. The data is then farmed out and the task queue maintained as new task definitions are generated by the worker processors. The workers simply process the task definitions and return output data, new tasks and a termination message.

3.4 GD with interworker communications (GDIW).

The final construct is a variant of the geometric decomposition skeleton (GD) with the added complication of worker processors communicating with each other directly. This requires a more complex scheme on the workers for the management of incoming messages and, in particular, clean termination of the algorithm. In this mode, the master divides the image up equally amongst the workers which process their subimages independently. Here data can cross inter-processor boundaries and the workers communicate among themselves to allow processing to cross these boundaries.

This requires more complex control than the other skeletons since each worker needs to be aware of which worker is processing which data so it can send boundary data to the correct worker. Termination is also more complex since all the workers must remain active until all the subimages have been processed. Another problem is that if global consistency is to be maintained then the farmer must relate workers' local data to global data. This construct is also much more difficult to load balance, especially in the presence of sparse data. The main advantage of this method is that it is more optimal in terms of communications costs and memory usage than the farming methods. An interesting comparison can be made with the task farm skeleton since an algorithm expressed using interworker communication can also be expressed using the task farm simply by treating boundary information as a new task. In fact the source language construct for this skeleton is essentially the task farm skeleton but when a task is generated it is placed in a list associated with a particular element of the decomposition. Fig. 4 illustrates the operation of this skeleton.

4 Skeleton application

There are a number of issues which occur in the application of these constructs.

The data handled by all these constructs can be of different topologies. For instance input and output data can be one dimensional, such as a list of points or a list of lines, or it could be iconic,

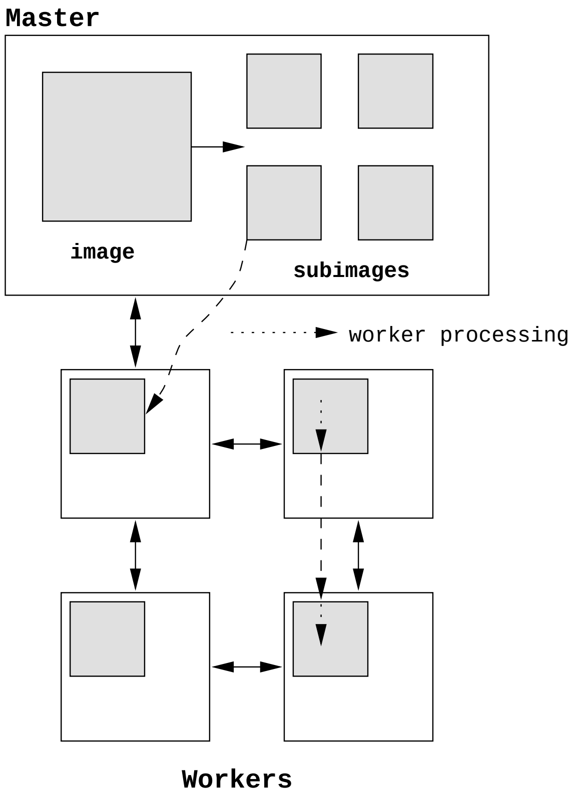

Fig. 4. Operation of the interworker communications (GDIW) skeleton

for example a depth image. In each case, however, the farmer has to be able to reconstruct the output data in the correct form from the local data uploaded from the workers. It is also possible for the same data to be represented in more than one way, for instance patches in depth data can be represented by an iconic array of patch numbers or as a list of (x, y) points. The choice of data representation is decided by the nature of the algorithm with suitable conversions between data representations depending on the cost of reorganisation versus algorithmic benefits. Note that the method of handling data input and output is determined by the representation. Where access to large global data is required, for instance a region growing algorithm, preloading is likely to be more efficient. The equivalent for output data is to retain data during processing and upload upon completion.

There are often optimisations of these constructs that can be made for specific circumstances. One example is the interworker communications skeleton where processing the boundary prior to processing the interior results in much less idle time at the end of the algorithm while the residual boundary information is processed. It is difficult to see how such a feature could be represented and predicted from source language constructs.

Fig. 5 summarises the relationships amongst the skeletons. In general, the GD and DF skeletons are equivalent at the source level but have different parallel interpretations and so are effectively the same skeleton. The TF and GDIW skeletons can be considered equivalent if a task definition and interworker message can be considered equivalent. Again, the difference boils down to one of parallel implementation.

Apart from this it is also possible to achieve the effect of one skeleton with a combination of two skeletons. Consider, for example, the GDIW skeleton which could be implemented as a straight GD phase but collecting the messages followed by subsequent farming out of the collected messages. The problem is that even once a skeletal solution exists in the source language there are still many architectural decisions that have to be made.

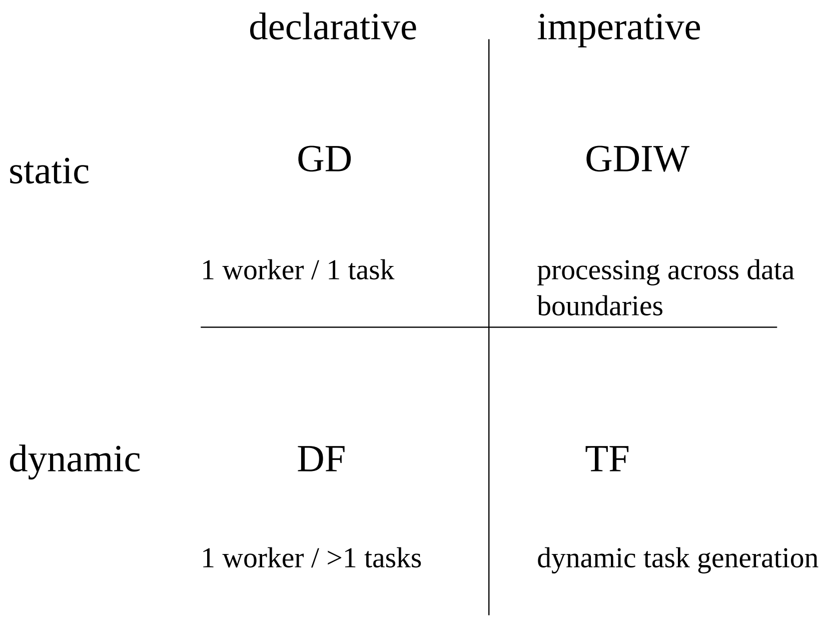

Fig. 5. Skeleton relationships

5 Skeleton use for parallel computer vision

The above skeletons were developed for the implementation of the low-level and intermediate-level stages of a multi-source parallel vision system which recognises 3D objects from registered intensity and depth images of a scene. For an intensity image: *Gaussian filtering* removes noise; *Canny edge detection* identifies pixels which may fall on lines; *edge tracking* identifies lines by grouping pixels. For the corresponding depth image: *mean/Gaussian curvature* classifies pixels according to the type of surface they are associated with; *connected components analysis* groups pixels into probable surfaces; *planar region growing* verifies surfaces identified by connected components. At the end of these stages, the original intensity and depth images have been processed to find lines and surfaces suitable for high-level model matching.

Fig. 6 shows the system as a flow of transformed images of a widget. Note that Gaussian filtering is subsumed under Canny edge detection and that connected components analysis is subsumed under planar region growing. The alignment of the original intensity image with the intended CAD model is shown at the bottom of the Figure: note that the depth image is also used for final model matching. Each module is now discussed in slightly more detail.

The *Gaussian filtering* module is a straight instantiation of the GD skeleton. The instantiation function simply performs Gaussian filtering upon the subimages. The only complication is that data has to be included at the worker from outside the subimage to allow calculation to reach the edges of the subimages. Note that, as an experiment, this module was implemented without preloading resulting in much lower performance.

The *Canny edge detector* is again a straight instantiation of the GD skeleton. This computation actually comprises several convolution phases which could be parallelised separately. Very good speedups were achieved using Canny edge detection within subimages so more complex solutions were not attempted.

Several different versions of the *edge tracking* module were developed before settling upon the composition of two DF farms. The problem was reconciling global accesses. The first farm performs tracking within subimages leading to a set of partial lines. This requires farming due to the sparseness of the data. The second farm performs merging of partial lines. Geometric constraints are required to prune noise from the first phase and reduce the complexity of the second phase.

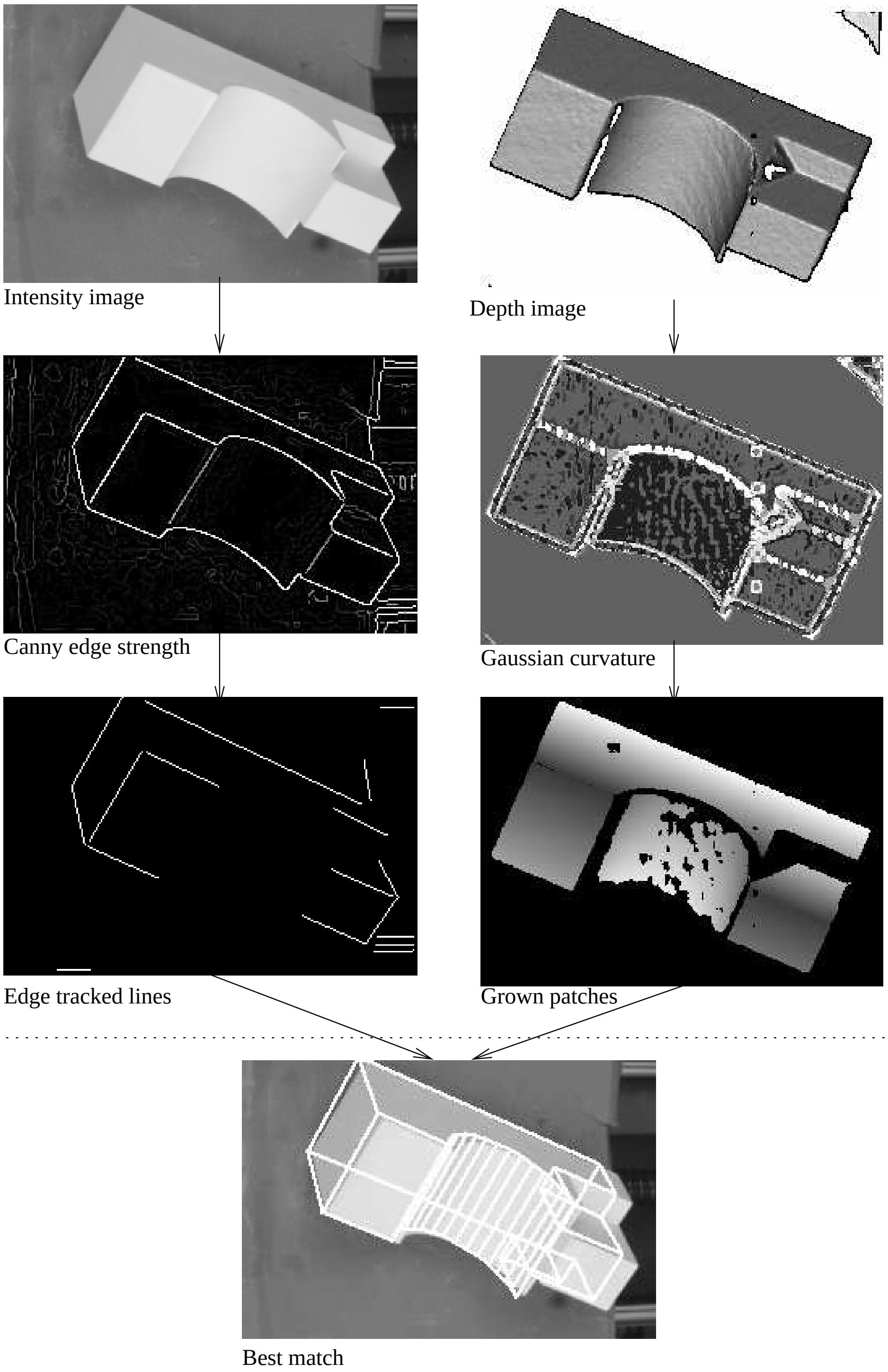

Fig. 6. Complete system

Mean and Gaussian curvature calculation is again based on convolution operations and was quickly implemented using the GD skeleton.

Connected components analysis shares the same difficulties as the edge tracking phase but with the added complication that the data is much more voluminous. The GDIW skeleton was developed specifically for this problem. A geometric decomposition is used and the sequential algorithm is run within the subimages. Boundary processing is sent directly to the worker handling that data which requires a list of processor/data allocations. The heuristic of processing boundary information before interior information is used and termination of the algorithm is more complex than for the other modules. The image is quite sparse (about 50-75% data) which accounts for the performance of this module.

Planar region growing is implemented as the composition of a DF skeleton and a TF skeleton. This module also went through several versions. The first DF phase handles sampling within the patches to determine seed regions. These are then passed forward into the TF skeleton which treats finely divided subimages as tasks. As these are grown they generate other tasks and the current queue of tasks is managed by the master processor. Again this skeleton was developed specifically for this module but could be applied in other situations.

It became apparent during system development that there was sometimes more than one way of implementing a particular algorithm using the basic constructs. For instance, an edge tracking algorithm was implemented as the combination of a geometric decomposition followed by farming to merge the results from subimages[3] whereas this algorithm could also be implemented by the interworker communications method. Similarly, a connected components algorithm on depth data was implemented using the interworker communications method but edge tracking was implemented as a geometric decomposition followed by a separate merge phase. The reason for this is simply the volume of depth data versus intensity data.

6 Performance

The above modules were implemented in occam2 on a Meiko Computing Surface with 32 T800 transputers. Table 1 shows, for each module, the skeletons used and the maximum speedup for 25 processors.

Table 1. Summary of existing vision algorithm implementations

Module	Skeleton	Max Speedup
Gaussian filtering	GD	10
Canny edge detector	GD	24
Edge tracking	DF $\rightarrow$ DF	15
Mean/Gaussian curvature	GD	23
Connected components	GDIW	13
Planar region growing	DF $\rightarrow$ TF	22

The performance of the individual modules is very good but the performance of the overall system is limited by sequential overheads to some of these modules. An example of this is planar fitting which is carried out sequentially using singular value decomposition to invert design matrices. The size of the matrices can be up to 10k in depth for the images processed which places a major burden on the farmer processor. Loading and saving of images is another overhead but not quite as significant as might be expected. Typically, speedups for the whole system of about 5 to 6 for 18 worker processors are found.

7 Conclusion

While the four skeletons discussed here are related to a common divide and conquer higher order function, their developments were driven by their intended uses rather than being well theorised. Nonetheless, we are persuaded informally of their equivalences.

Good reuse was made of the geometric decomposition (GD) and task farm (TF) skeletons. This is not surprising given the internal coherence of the application area: in general, software component reuse depends on semantic similarities between problem domains.

Overall, the use of these skeletons was exemplified by the construction of a substantial parallel application. This provides further evidence of the efficacy of skeleton based system development.

8 Acknowledgements

We would like to thank our colleagues in the Computer Vision Group at Heriot-Watt University, in particular Bill Austin who developed the high-level model matching stage of the system discussed here.

This work was supported from 1993 to 1996 by EPSRC Grant GR/J07884.

References

1. M. I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* Pitman/MIT, London, 1989.
2. A. M. Wallace, G. J. Michaelson, P. McAndrew, K. Waugh, and W. Austin. Dynamic Control and Prototyping of Parallel Algorithms for Intermediate and High-level Vision. *IEEE Computer*, 25(2):43–53, Feb 1992.
3. N. R. Scaife, G. J. Michaelson, and A. M. Wallace. A Method for Developing Parallel Vision Algorithms with an Example of Edge Tracking. Technical Report RM/96/1, Department of Computing and Electrical Engineering, Heriot-Watt University, Riccarton, Edinburgh, Jan 1996.