IMPLEMENTING MATHEMATICAL MORPHOLOGY IN ISETL-LINDA

A.I.T. Rowstron and A.M. Wood

The University of York, United Kingdom

Abstract

We present here the description of a parallel implementation of mathematical morphology using the LINDA model of parallel communication embedded within ISETL. The morphological operation of dilation of binary images is considered in detail, with an ISETL implementation given initially, then a description of how this was converted to run in a parallel version of ISETL, ISETL-Linda.

INTRODUCTION

We are interested in the implementation of binary mathematical morphology using ISETL-LINDA. ISETL (Baxter et al. (1)) is an interactive extension of SETL, which is a programming language based on mathematical notation and objects, primarily sets and functions. ISETL is a ***high level imperative programming language*** in which all objects (including functions) are first class.

LINDA (Carriero and Gelernter (2)) is a parallel coordination language, which uses a shared ***tuple space*** to allow multiple processes to communicate. It is independent of any machine architecture and all communication between processes is asynchronous which must occur through the tuple space – an unordered *bag* of tuples. Tuples can be placed into a tuple space, and read or removed from a tuple space. Because LINDA is only concerned with process coordination, it is embedded within another language, providing five new primitives for the language in which it is embedded. ISETL-Linda (Douglas et al. (3)(4)) is an interactive parallel set-based language. It is the embedding of LINDA within ISETL. In the following, work on the implementation of mathematical morphology in ISETL is presented, and then a discussion of how this was converted into a parallel implementation. First we present a brief description of both binary mathematical morphology and LINDA.

MATHEMATICAL MORPHOLOGY OPERATORS

The main operations of mathematical morphology are *erosion* ($\ominus$), *dilation* ($\oplus$) and set operations. By creating expressions using different combinations of these operators it is possible to create other operations, such as *opening* and *closing*. For example opening is defined as an erosion followed by a dilation. The operations of dilation and erosion are dyadic, requiring an image and a structuring element. Both are represented by subsets of Euclidean 2-space (E^2) which we will consider as a set of tuples containing two integers. A (binary) image is then represented by the presence (1-pixels) or absence (0-pixels) of a coordinate pair in the set. The set representing the structuring element is derived similarly.

Given an image **A** and a structuring element **B** containing elements **a** and **b** respectively then dilation is defined by:

$$A \oplus B \equiv \bigcup_{b \in B} (A)_b \qquad (1)$$

Where, $(A)_x$ is the *translation* of set A by the point x, such that:

$$(A)_x \equiv \{c \in E^2 | c = a + x \text{ for some } a \in A\} \qquad (2)$$

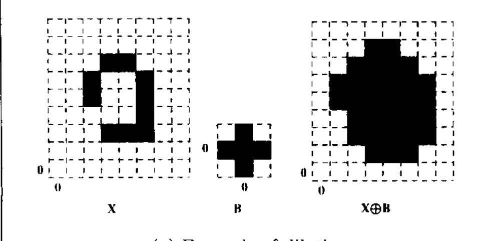
Figure 1(a) shows an example of dilation. Erosion is defined by:

$$A \ominus B \equiv \bigcap_{b \in B} (A)_{-b} \qquad (3)$$

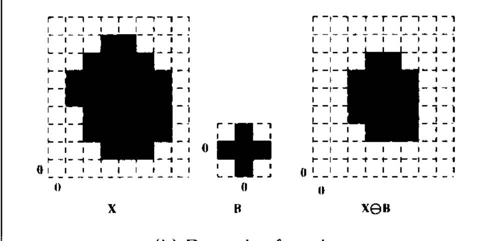
Figure 1(b) shows an example of erosion. Figure 1 as a whole shows the effect opening would have on an image (opening is dilation followed by erosion). Haralick et al. (5) present a complete overview of both binary and grayscale mathematical morphology which includes a description of the properties of these operations.

LINDA

The originators of LINDA (2) describe it as a ***co-ordination language***; its principal purpose is to orchestrate interprocess communication. The model only allows communication between different processes involved in a (concurrent) computation via



(a) Example of dilation



(b) Example of erosion

Figure 1: Example of opening (dilation then erosion)

a *tuple space*, which can be thought of as a bag into which tuples are inserted and withdrawn by the processes. LINDA is only concerned with process co-ordination, and thus is always embedded into some other *host* language e.g. C, Prolog, Lisp, etc. which provides the conventional computational mechanisms. LINDA provides four operations on tuple spaces:

out (tuple) This places the given tuple into the tuple space.

in (template) This attempts to match the template with a tuple in the tuple space and return it. If a match does not exist the operation *blocks* until a suitable tuple becomes available (if ever). If the match succeeds, the matched tuple is removed from the tuple space.

rd (template) This is the same as **in** except the matched tuple is not removed from the tuple space — a copy is returned.

eval (tuple) This creates so-called *active* tuples — each element of the tuple is evaluated *concurrently* and, when complete, the resulting tuple of values is placed in a tuple space. We do not allow any other primitives to act on an active tuple. This is LINDA's mechanism for spawning new processes.

The LINDA model is intended to be an abstraction, and as such is independent of any specific machine architecture. This has meant that alternatives and extensions to the basic LINDA model have been proposed and investigated. The extensions that are used currently by the University of York implementations are:

multiple tuple spaces The addition of multiple tuple spaces has been discussed for some time. Currently, York has adopted the idea that each tuple space is independent of any other tuple space; there are however other ways of implementing multiple tuple spaces, such as using a hierarchical structure (Gelernter (8) and Hupfer (9)). The addition of multiple tuple spaces is achieved by incorporating a `tuple space` type and a primitive to create a new tuple space. For instance, the type `ts` might represent a tuple space, and the primitive `tsc()` could create values of type `ts`. Here is an example which (using a C like syntax) creates a new tuple space with handle `tsNew` and then performs some simple operations using the new tuple space:

```
ts tsNew = tsc();

tsNew.out("hello");
tsNew.out("foo", 5");
tsNew.in("foo", ?int);
```

collect primitive A new tuple-space primitive (Butcher et al. (10)), `collect()`, has been proposed which subsumes the semantically problematic `inp` and `rdp` primitives[1]. This primitive by its very nature requires multiple tuple spaces. Given a tuple space handle `ts` and a tuple template, the command `collect(ts1, ts2, template)` transfers tuples that match the template in `ts1` to `ts2`, returning a count of the number of tuples transferred. This operation appears to encapsulate just sufficient *global* information about the state of a distributed computation without giving rise to the semantic problems of `inp`.

copy-collect primitive This is a new tuple-space primitive, which has recently been proposed in the light of work on the implementation of many different algorithms in LINDA. This primitive is very similar to `collect` but it is non-destructive. Hence it *copies* all the tuples that match the tuple template to a separate tuple space rather than *moving* them. As with `collect` it returns a count of the number of tuples copied.

In ISETL-LINDA tuple spaces are incorporated in the language by providing a new type, the *bag*. A bag is similar to a set except that repetition of elements is allowed. Therefore, the bag $\{|1,2,3,3,4|\}$ is the same as $\{|3,4,2,3,1|\}$. However, the operations of union and intersection are not provided for bags, the only way to manipulate a bag is via the LINDA primitives.

[1] `inp` and `rdp` are predicate versions of `in` and `rd` which return either a tuple or 'false'. For a description of their semantic problem see (10.

MAPPING MORPHOLOGICAL OPERATIONS TO ISETL

Initially, we considered the implementation of the morphological operations in ISETL. ISETL supports sets as first class objects, and subsequently most of the set operations are already provided (cardinality, union and intersection). This means that there is a natural mapping between mathematical morphology, which is based on sets and ISETL.

In order to demonstrate how this natural mapping occurs, let us consider the implementation in ISETL of the morphological operation of dilation.

Dilation

Dilation is the union of sets of points produced when every member of an image is translated by every member of a structuring element. Therefore, in order to achieve dilation we need to be able to translate an image by a given member of the structuring element (a two-dimensional vector). The result will be a set that represents the image translated by the given vector. This function written in ISETL is:

```
translate := func(image,vec);
  local ans, member, lp2, ans2;
  ans := {};
  for member in image do
    ans := ans union {add_vec(member,vec)};
  end for;
  return ans;
end func;
```

This function takes each member of the set image, produces a set that contains that member translated by the vector, and then unions the result with a set that is being used to store the answer. This answer set is then returned at the end. The function add_vec checks the cardinality of the two tuples, and then returns a tuple representing the vector addition of the two tuples passed.

The dilation operation is then achieved by taking each element within the structuring element and producing its translation of the image, and then unioning the results. Here is the function for dilation:

```
dilate := func(image, structelement);
  local ans, member;
  ans := {};
  for member in structelement do
    ans := ans union translate(image,lp);
  end for;
  return ans;
end func;
```

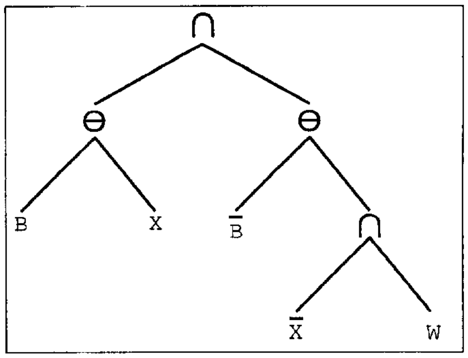


Figure 2: Tree Representation

The implementation of erosion is very similar, with the union operator in the dilation function being changed to an intersection operator, and also the structuring element needs to be reflected. Now we consider the parallel implementation of the morphological operations.

ISETL-LINDA IMPLEMENTATION

Introducing parallelism to mathematical morphology

Mathematical morphology like many image processing operations is inherently parallel. However, within mathematical morphology it is possible to identify parallelism at two levels. The first is at the expression level. Consider an expression using morphological operations, such as this expression $(B \ominus X) \cap (\overline{B} \ominus (W \cap \overline{X}))$ which can be used to recognise patterns in an image, (taken from Crimmins and Brown (11)). By examining the tree representation of this expression, as shown in Figure 2, it becomes clear that it is possible to evaluate certain parts of the expression in parallel. In this example, it is possible to evaluate the expression $B \ominus X$ in parallel with the expression $\overline{B} \ominus (W \cap \overline{X})$. At this level it is also sensible to look for common expressions, as they need only be evaluated once.

The second level of parallelism is at the morphological operator level. Each operation can be seen as a number of tasks that have to be applied to each member of a coordinate set. For example, within the dilation operation the translation of each element within the coordinate set can be done in parallel. Within this paper we only consider parallelism at this level.

Mapping morphological operations to ISETL-LINDA

In order to utilise the parallel features of ISETL-LINDA we need to use bags rather than sets in the implementation. A bag is very similar to a set except that it is possible to have repetition of members.

Due to this similarity bags are often referred to as *multisets*. The set operations of union and intersection can be defined for bags. Given a bag **A** which is $\{|1,1,3,3|\}$ and a bag **B** which is $\{|1,2,3|\}$ then $A \cup B$ is the bag $\{|1,1,2,3,3|\}$ where this bag is created by taking the maximum number of occurrences of a member in **A** or **B**. Similarly the insection of A and B ($A \cap B$) is the bag $\{|1,3|\}$ where this bag is created by taking the minimum number of times an element appears in **A** and **B**. As well as intersection and union for bags, there is also the *sum* of two bags. The sum of A and B is the bag $\{|1,1,1,2,3,3,3|\}$, where this bag is created by taking the sum of the number of occurrences of an element in both **A** and **B**. For more information on the theory of bags and sets see Manna and Waldinger (12).

From a morphological point of view there is no reason why each coordinate has to be unique in an image set. The operations of dilation and erosion do not require this as a starting state. The operations of intersection and union are defined for bags and it is possible to convert a bag to a set by removing replication of elements. In the light of this let us reconsider dilation. Let us alter the definition of dilation slightly, defining it in terms of bags rather than sets:

$$A \oplus B \equiv \mathrm{SUM}_{b \in B}(A)_b \qquad (4)$$

Therefore, dilation of a bag returns a bag, with duplicated elements within it. If this bag were converted to a set, the answer would be the same as if the dilation had been working with sets. Extending this, consider a dilation followed by another dilation. The first dilation will produce a bag, which will be dilated to produce another bag. If this bag were then converted to a set it would be the same as the one produced if both dilations were using only sets. Therefore, in our implementation we provide a parallel dilation, and a function that will remove repetition from a bag.

There is a drawback with this approach, it provides a fast dilation but the size of the bag representing an image grows rapidly with many of the coordinates being replicated. Therefore, it is often more efficient to remove repetition from within a bag before doing a dilation. This may be seen as a kind of explicit garbage collection of a bag.

Erosion is rather more interesting. We wish to attempt to keep the replication of coordinates in a bag to a minimum, as this reduces the number of wasted operations (operations that have to be completed for the same coordinate more than once). It so happens that it is as easy to implement an intersection that acts like a set intersection as one that acts like bag intersection. Therefore, in Sessence the erosion operation automatically removes repetition from within a bag.

Next we will consider the implementation of dilation and the bag to set function in more detail.

Dilation

When we redefined the dilation operation for bags, we used bag sum rather than bag union. We need to consider how the union and sun operations might be implemented. The union operation requires a comparison between the bags. So we take an element from one bag, see how many times it occurs in that bag, and see how many times it occurs in the other bag. With the sum operation we just take every element in each bag and place them in a new bag. Because the sum operation requires no comparisons it is a much cheaper operation.

A parallel dilation is achieved by using a data parallel approach. A worker process is created for each element within a structuring element. Each worker then creates a process for each coordinate in the image, and each of these performs the translation of a single point of the image by a single element of a structuring element. This is a *fine grained* approach. We have built in the ability to control the number of processes spawned at each level, therefore providing control over the granularity of the processes.

This leaves us in the state where we have a number of bags each containing an image translated by one element of the structuring element. We need then to do the sum operation. However, if we make all the processes write their results to the same bag, we remove the need to perform the sum operation, as this has been achieved already.

Bag to set conversion

This appears at first to be a trivial problem. A sequential implementation would be to take an element from a bag, see if there are any other elements in the bag that match it and remove them if there are. However, to do this in parallel using LINDA is not so easy. The solution is to split the Euclidean space that the bag covers into sections and then spawn a number of workers and explicitly give each of them a section of the Euclidean space to reduce. In order to do this the size of the image must be calculated, which can be done using a parallel fold (3).

A similar approach is taken for the intersection required by erosion. However in this case it is possible to use the size of the image and structuring element to calculate the size of the resultant image, therefore the calculation of the size of the resultant image can be done in parallel with the actual erosion.

THE ADVANTAGES OF USING LINDA

There are several hardware architectures specifically designed to provide fast parallel implementations of mathematical morphology. For example architectures that are based on pipelines, such as the Cytocomputer by Lougheed et al. (6), or implementations based on systolic architectures, for example Lenders and Schröder (7). Our implementation has many advantages due mainly to its flexibility. There are few hardware implementations that are capable of dealing with truly non-uniform images. Many place restrictions on the size of the image or require it to be square. Our implementation is able to cope with images of any shape and size because it deals purely with sets of coordinates. Our implementation also has the advantage that it can alter its granularity to make full use of the resources that are available.

When considering the parallel decomposition of any morphological operation, we have attempted to create an implementation where the granularity of the solution is easily controllable. Therefore, if the program were to run on a two or three processor system, it would be possible to have coarse grained parallelism, where if the program were to run on a thousand processor computer a finer granularity can be achieved.

CONCLUSION

We have presented a description of the development of a parallel mathematical morphology system using LINDA. We described the initial work on an ISETL version, and how the conversion from this to a full parallel version was achieved. Within the limitation of this paper we have not considered many of the other problems that were encountered, such as dealing with the complements of images and structuring elements.

Although we have a basic working system, there is still much development to be completed, including the extension of the system to deal with grayscale images and introducing parallelism at the expression level as well as at the data level.

Acknowledgements

The authors wish to thank Andrew Douglas for his helpful comments and suggestions. During this work Antony Rowstron was supported by a CASE studentship from British Aerospace Military Aircraft Division Ltd and the EPSRC of the UK.

References

1. N. Baxter, E. Dubinsky and G. Levin, 1989, "Learning Discrete Mathematics with ISETL", Springer Verlag.
2. N. Carriero and D. Gelernter, 1989, "Linda in context", Comms. ACM, 32,444-458.
3. A. Douglas, A. Rowstron and A. Wood, 1995, "ISETL-LINDA: Parallel programming with Bags", Submitted for EURO-PAR 95.
4. A. Douglas, A. Rowstron and A. Wood, 1995, "Linda Implementation Revisited", 18th World occam and Transputer User Group Conference, Transputer and occam Engineering Series, IOS Press.
5. R. Haralick, S. Sternberg and X. Zhuang, 1987, "Image Analysis using Mathematical Morphology", IEEE Trans. Pat. Anal. and Mach. Intel., 9, 532-549.
6. R. Lougheed, D. McCubbery and S. Sternberg, 1980, "Cytocomputers: Architectures for parallel image processing", Proc. IEE Workshop on Pic. Data Rep. and Man.,281-286.
7. P. Lenders and H. Schröder, 1990, "A programmable systolic device for image processing based on Mathematical Morphology", Par. Comput., 13, 337-344.
8. D. Gelernter, 1989, "Multiple tuple spaces in Linda", PARLE '89: Par. Arch. and Lang.,Volume II: Par. Lang.,Springer Verlang LNCS Vol. 366, 20-27.
9. S. Hupfer, 1990, "Melinda: Linda with Multiple Tuple Spaces", Tech. Rep. YALEU/DCS/RR-766, Yale University.
10. P. Butcher, A. Wood and M. Atkins, 1994, "Global synchronisation in Linda", Conc.: Prac. and Exp., 6, 505-516.
11. T. Crimmins and W. Brown, 1985, "Image algebra and automatic shape recognition", IEEE Trans. Areo. and Elec. Sys., AES-21, 60-69.
12. Z. Manna and R.Waldinger, 1984, "The logical basis for computer programming: Volume 1", Addison-Wesley.