

Parallel Programming in Lucid

Ed Ashcroft

*Declarative Languages and Advanced Architectures Group
Arizona State University*

Abstract

This paper essentially looks at how Lucid programs can be made more implicitly parallel. In the process, it introduces a few interesting Lucid programs and raises a few points about Lucid itself.

Introduction

Proponents of Lucid, including myself, have been guilty of glibly saying that, because Lucid programs are implicitly parallel, we don't need to deal with the subject of parallel programming. This reasoning is based on categorizing parallel programming as something that has to be done when one has explicit parallelism (involving setting up parallel processes, monitors, message-passing, etc.) and, of course, Lucid is above all that. In fact, parallel programming is a concept that makes sense for implicitly parallel languages such as Lucid. The amount of implicit parallelism in a Lucid program depends on how the program is written. Two programs that solve the same problem can have widely different amounts of implicit parallelism. In Lucid it is true that "parallelism is the norm" but the data dependencies in a program may enforce sequentiality. *Doing parallel programming in Lucid means writing programs that do not unnecessarily impose sequentiality.*

Currently, we do not have a parallel implementation of Lucid, so it may seem foolish to be worrying about implicit parallelism. In fact, this is shortsighted; parallelism *should* be kept in mind. One of the things that has been found in practice is that it is not much more difficult to think in the "reduce sequentiality" mode than in the normal "ignore parallelism" mode, and the resulting programs are often easier to understand and sometimes can be more efficient than the sequential programs, even when run on a sequential machine. What will be done here is illustrate these ideas using a few examples, and, in the process, show some interesting Lucid programs.

The paper will tend to concentrate on programs that use a particular idea for reducing sequentiality: the use of tournaments. It will be shown how the use of tournaments leads naturally and simply to programs for mergesort and the generalized Hamming Problem, for example. Also, it will be shown how tournaments can be used in widely different applications, from the n Body Problem to LU Decomposition of Matrices. Toying with parallel programming has

stimulated me into writing quite a lot of programs in Lucid and has definitely shown me that the use of space (and not just time) is really important and liberating. Also, it has suggested that higher-order functions in Lucid might well be desirable after all.

The Tournament Method: Summing

It is very common to want to calculate the sum of a set of numbers. In Lucid, if the numbers are the consecutive values in 'time' of a variable l , say, it is natural to calculate the running sum using the following "sequential" definition

$$s = \text{first } l \text{ fby } s + \text{next } l;$$

It is natural because one tends to think of the values of l being calculated at successive times anyway, so the running sums can be calculated as the l 's are calculated. However, when the values to be summed are in a variable, A , say, that varies in *space*, it is natural to expect all the values of A to be available simultaneously. In this case, the corresponding space-based definition

$$s = \text{initial } A \text{ sby } s + \text{succ } A;$$

imposes too much sequentiality (though it does compute the correct answer, of course). Ideally, we would like all the values of A to be added together simultaneously. Lacking such an arbitrary-arity addition operation, one solution is to use a divide and conquer approach, and set about, recursively, adding up the first half of the values of A and the second half of the values of A , simultaneously. When this is done, the two resulting values are added together, with a normal addition operation. This uses a recursively defined summation function (and, the way I have described it, it really only works when A has exactly 2^n values, for some n). If one looks just at what additions get done, and when, one sees that the values of A are added together in pairs *simultaneously*, giving 2^{n-1} sums, then these sums are added together in pairs simultaneously, giving 2^{n-2} sums, and so on. In $\lg n$ steps we have the sum (rather than in $n-1$ steps with the sequential sum). We are using a *tournament*. (Historically, tournaments have used binary relations, such as "is greater than" or "is better with a lance" but here we will also use binary operations such as addition.)

Can we do this in Lucid? Initially, I thought not, but in fact it is very easy. The expression

$$A + \text{succ } A$$

denotes the sums of pairs of A , but it sums too many pairs; it adds A_0 to A_1 but it also adds A_1 to A_2 . We need every other one of these sums, for which we really need

$$(A + \text{succ } A) \text{ atspace } (2 * \text{here}).$$

(The Lucid constant *here* corresponds to the definition

here = 0 sby here + 1;)

This gives us an “array” that is the result of one round of the tournament. We must keep doing this, till we get the sum we need. The question is, where do we keep all these arrays? We need another dimension, orthogonal to that containing the values of A that we want to sum. That could be the time dimension or another space dimension. Here, we will consider using time, as follows:

*pairsum = A fby (pairsum + succ pairsum) atspace (2*here);*

If there are 2^n values of A to add, we need the value of *pairsum* at time n .

This is fine, except when A varies in time as well as in space. In that case we need to freeze variable A at each point in time (giving us corresponding sums at each point in time). This is exactly what *is current* does for us. (In case there are different numbers of values to be added at each time, we had better freeze n too.) We will express the final result as a function:

tournamentSum(A, k) = pairsum attime N

where

N is current lg k;

AA is current A;

*pairsum = AA fby (pairsum + succ pairsum) atspace (2*here);*

end

Here, k is the number of elements of array A that we want to sum; it is assumed to be a power of two. (In describing what this program means, we have assumed that we are considering position 0 in dimension 0, in which case it gives us the sum of the first k values of A in dimension 0. In fact, at position 1 of dimension 0 we will have the sum of the *second* k values of A , and so on.)

There are cases (as we will see later) in which it is not possible to use time as the extra dimension, and we must use a space dimension. In this case we will need a space version of *is current*. This proliferation of *is currents* will not go down well with some Lucid buffs, who would rather have *is current* banned altogether. Such buffs claim that the availability of space dimensions makes *is current* superfluous. I, on the other hand, claim that *is current* is necessary, and, moreover, the availability of space dimensions makes *is currents* necessary for each dimension.

The Tournament Method: General

The tournament technique can be used for any associative binary operation, such as *max* and *min*. In general, the associative binary operation that we do the tournament for must be pointwise in the extra dimension that we keep the paired results in, as well as pointwise in the dimension holding the values being considered.

One fascinating use of the tournament method is with the binary operation *merge*, which merges together ordered sequences. This operation has been defined as a Lucid function (in the Lucid book) as follows:

```
merge(a,b) = if xx <= yy then xx else yy fi
  where
    xx = a upon xx <= yy;
    yy = b upon yy <= xx;
  end;
```

(The sequences that it merges are time sequences.) The operation is associative (it doesn't matter in which order we do a multiple merge), so the tournament technique should work. In fact, it does. If *A* is an array (dimension 0) of time sequences, we can hold a tournament with merge provided the extra dimension that we use is not the dimension that the ordered sequences are in (i.e., time). We have to use a new space dimension (say, dimension 1), and, in that case, we also must have a space version (*is current_1*?) of *is current*. (We could have rewritten the part of the program that produces *A* and have it produce space sequences rather than time sequences, but that would have qualified as unreasonable extra work. I said that it would be easy to get into reduce sequentiality mode, and it should be.) The appropriate function is

```
tournamentMerge(A, k) = pairmerge atspace_1 N
  where
    N is current_1 lg k;
    AA is current_1 A;
    pairmerge = AA sby_1 merge(pairmerge, succ pairmerge) atspace (2*here);
  end
```

With this function we can now write a program to sort *A*:

```
tournamentMerge(A fby eod, k)!
```

This is mergesort. (At each point in dimension 1, this program gives us the time sequence of the first *k* elements of *A* in dimension 0 (for that point in dimension 1) in order. If *A* varies in time, only the elements of *A* at time 0 are sorted.)

The implicit parallelism in tournaments in Lucid is exactly the parallelism described when talking about tournaments in general: at the first stage there are $k/2$ operations on adjacent array elements that can be simultaneously performed, at the next stage there are $k/4$, and so on. In a parallel implementation, the computation

would be finished in $\lg k$ steps, and in a sequential implementation $k-1$ steps would be needed (which is exactly how many steps would be needed by the straightforward sequential program).

The last paragraph suggests that we could sort using the straightforward sequential merging program

```
s atspace K
where
    K is current k;
    s = initial(A fby eod) sby merge(s, succ(A fby eod));
end
```

and it would be just as fast as the tournament based mergesort program! In fact, though both programs do the same number of merges $(k)-1$, when the computation time taken by the merges is taken into account in the two cases, the mergesort wins. This is an example of how avoiding sequentiality sometimes has the serendipitous effect of reducing computation times even on sequential machines.

Tournaments for Arbitrary Length Sequences

The tournament method need not be restricted to only being applied to sequences whose length is a power of two. If k is the size of the sequence A , and A is found in dimension 0, the following tournament sums the elements of A .

```
tournamentSum(A, k) = new attime power_of_two
where
    K is current k;
    AA is current A;
    new = AA fby (if width mod 2 eq 0 then new + succ new
    else new sby new + succ new fi
    at (2*here));
    width = K fby ceil(width/2);
    power_of_two = now asa width eq 1;
end
```

(The Lucid constant *now* corresponds to the definition

```
now = 0 fby now + 1;)
```


This modification of the tournament method is due to Alan Jorgensen, a graduate student at ASU. Its main advantage is that it applies to any use of tournaments, not just summation. In the rest of the paper we will continue to use straightforward tournaments that really only work on sequences whose length is some power of two. In reality, these programs will be “Jorgansenned”.

Hamming's Problem Revisited

In the Lucid book there is a program for Hamming's Problem. In it, three sequences are merged together, using two applications of the merge function. We now know how to merge an arbitrary number of sequences, which suggests a modification of the Hamming program for a generalized problem. The new problem is “Given a list of prime numbers, produce the ordered sequence of all positive integers that have only those primes in their prime decomposition.” (The original Hamming Problem is the special case where the list just consists of 2, 3, 5.) It is quite easy to produce the new program. (As mentioned above, we will assume that the number of primes on the list is a power of two, but we can also make it work for any number of primes, thanks to Jorgansen.)

hamming

where

PRIMES is current primes;

K is current initial k;

*hamming = 1 fby tournamentMerge(PRIMES * initial hamming, K);*

end

The input variable *primes* has to vary in dimension 0, while *k* must *not* vary that dimension. (Actually, the program ignores all values of *k* for dimension 0 positions other than 0.) The program extends serendipitously to any of the other dimensions or to time, thanks to the *is currents*.

The serendipitous extension of programs like this to arbitrary dimensions is a great source of implicit parallelism. Part of the discipline of parallel programming in Lucid should be to ensure that this extension happens. Doing so by judicious use of *is current* also has the beneficial effect of eliminating demands for input values that are not really necessary. For example, if the declaration

PRIMES is current primes;

were missing (and the program just used *primes* instead of *PRIMES*), the program would keep asking for values of *primes* at different times, even if you didn't want to

consider *primes* to be varying in time. That is because *primes* will be multiplied by *hamming*, and *hamming* varies in time.

Other Applications

Other programs have been written at ASU in order to see if Lucid is suitable for applications that are generally considered to be useful for exploiting parallelism. A program for the n-body problem was included in a paper I co-authored at the last ISLIP. That program used *tournamentSum* (actually it was called *binary_sum*) for each body, to add together the accelerations caused all the other bodies.

Here I will include a program for the LU-decomposition of matrices.

The following equations define LU decomposition without pivoting in terms of two simple recurrence relations:

$$L_{ij} = A_{ij} - \sum_{k=1}^{j-1} L_{ik} U_{kj} \quad j \leq i, i=1,2,3,..$$

$$U_{ij} = \frac{A_{ij} - \sum_{k=1}^{i-1} L_{ik} U_{kj}}{L_{ii}} \quad i < j, j=2,3,4,...$$

The above recurrence relations can be written directly in Lucid as

```

if j <= i then l else u fi
where

l = A - sigma(M, j-1);
u = (A - sigma(M, i-1))/diag(l);

M = row(l)*col(u);

sigma(P,n) = sum attime (n+1)
where
    sum = 0 fby sum + P;
end;
i = here;
j = here_1;
row(N) = swap_1 N;
col(N) = swap N;
diag(L) = L atspace_1 i;

end

```

(The operator *swap_n* interchanges the contexts of time and the *n*-th dimension. As I explained at ISLIP 88, *swap_n* has the effects of both *all_n* and *elt_n*.)

Clearly, the Lucid program expresses only those data dependencies inherent in the recurrence relations. This means that any parallel implementation will not be affected by data dependencies introduced by features or constructs of the programming language. This enables a parallel implementation to exploit all the available parallelism.

The naive LU decomposition defined by the above recurrence relations and implemented by the above Lucid program will not work on singular matrices. Despite this drawback of the naive LU decomposition method, I have included it here to illustrate how it is often simple to take mathematical recurrence relations and to implement them directly as Lucid programs.

A more practical algorithm for computing LU decomposition is one that uses pivoting, as might be expressed by the following Pascal-like pseudo-code:

```
for k ← 1 to n-1 do
  Find l such that
    |A(l,k)| = max(|A(k,k)|, ..., |A(n,k)|)
  PIV(k) ← l {the pivot row}
  A(PIV(k), k) ↔ A(k, k)
  c ← 1/A(k,k)
  for i ← k+1 to n do
    A(i, k) ← A(i, k) × c
  for j ← k+1 to n do
    A(PIV(k), j) ↔ A(k, j)
    for i ← k+1 to n do
      A(i, j) ← A(i, j) - A(i, k) × A(k, j)
```

This algorithm is implemented by the following Lucid program:


```

LU
where
  AA is current A;
  N is current n;
  LU = makeRow(firstCol(D));
  B = AA fby restCols(D);
  D = if i <= k then swapPivRow else swapPivRow/pivRow sby_1
      restCols(swapPivRow - firstCol(D)*pivRow) fi;
  pivRow = B atspace pivPt;
  swapPivRow = cond
    i eq pivPt : B atspace k;
    i eq k     : pivRow;
    default    : B;
  end;
  pivPt = tournamentMaxx(here, N);
  tournamentMaxx(h, n) = b attime N
    where
      H is current h;
      N is current n;
      b = H fby maxx(b, succ b) atspace (2*here);
    end;
  maxx(v,w) = if edgeB atspace v > edgeB atspace w then v else w fi;
  edgeB = abs(firstCol(B));
  firstCol(M) = initial_1 M;
  restCols(M) = succ_1 M;
  makeRow(C) = swap_1 C;
  i = here;
  j = here_1;
  k = now;
end

```

(The use of *maxx* in a tournament is allowed because the associative function *maxx* is pointwise in time and also pointwise in dimension 0.)

It is interesting to compare this Lucid program and the conventional Pascal-like pseudo-code given previously. In the Lucid program there is one 'loop' that is evident, in the definition of *B*. The value of *B* is a (time) sequence of matrices, starting with the original matrix *A*. The next values of *B* are defined in terms of the current values of *B*, via *D*. Variable *D* is defined to be a whole new array, corresponding to the changes made to *A* by one pass through the outer loop of the imperative program. The first column of *D* will be a column of the final result, *LU*, and the rest of *D* is the next value of *B*. (The imperative program builds up the final result a column at a time, also.)

The outer loop of the imperative program is discernible in the indirectly recursive definition of *B*. None of the other loops in the imperative program correspond to recursive definitions in the Lucid program. The variable *swapPivRow* encapsulates all the changes made by the 'interchange' operations of the imperative program. All the other changes made to *A* are expressed in the definition of *D*. All those changes can be made simultaneously and no recursion is needed. There are no control structure in the program (apart from the outer loop) because no control structure is necessary! In Lucid, contexts are implicit, parallelism is implicit, and control is implicit. The parallelism in the Lucid program is fine grain and implicit whereas the task-level parallelism of a parallel implementation of the sequential program given earlier would be coarse grain and would need to be explicitly specified.

Conclusion

This paper has attempted to show that many useful and interesting programs can be found when one programs in the "reduce sequentiality" mode. All the programs used objects that vary in space as well as in time. This seems to be the key to discovering implicit parallelism. Also, a great deal of implicit parallelism comes from serendipity, which is the way that Lucid programs often extend in a regular way to higher dimensioned objects. Getting into the habit of making this extensionality work well seems to be good programming practice: the way one is forced to think makes it easier to think of and control the behavior of multidimensioned objects. That extensionality is helped enormously by the disciplined use of *is current*, both for time and for each space dimension.

The similarity between all the tournament functions given here suggests that they should all be considered as the results of applying a single higher order tournamentizing function to different associative operations. The higher order behavior is very limited. The higher order tournamentizer could be removed from programs by a simple preprocessing phase, basically a sort of macro expansion. It is not clear that full-blown higher order functions should be introduced into Lucid.

Another interesting possibility is suggested by the way that the tournament functions can be specified to work on different dimensions. If there were a tournamentizing function, it would need to be told which dimensions it is to work on as well as which associative operation to apply. This suggests that the definition of the tournamentizer will have operations that are parameterized by the dimension they are to work on. Again, this could just be a property of the macros that are going to be removed by the preprocessing phase, but perhaps it should be more than that. Perhaps such operators with parameterized dimensionality should be added to Lucid itself. This can really only be decided after many programs have been written and the experience gained has been reviewed.

On a final note, the Lucid program for LU Decomposition using pivoting is a Lucid version of an iterative, imperative, sequential, Pascal-like program. It has a lot more potential parallelism than the imperative program, but perhaps it would be even better had it been directly written as a Lucid program. Perhaps pivoting is an invention of sequentiality-fixated programmers and numerical analysts. The whole field of reprogramming in Lucid, as opposed to rewriting in Lucid, may be very fertile.