Parallel Language Extensions for Side-Effects

by Sean Halle

May 16, 2008

Abstract

Side effects make many analyses more difficult or impossible to perform in compilers for parallelism. Yet it is often convenient for programmers to use side-effects when writing their code. Perhaps more important is that algorithms represented in terms of side-effects are often higher performance on shared memory multicomputers. It would be nice to have a way to express side-effects and also make them precise for the compiler to analyse.

We propose a syntax that may be beneficial in this sense. Our main premise is that side-effects are communication events. In a serial program, the programmer knows exactly which writes communicate to which reads. If they don't, the chances of the program being correct are slim. However, the compiler is unable, in general, to deduce the rules of which writes communicate to which reads in the dynamic trace. Our proposed syntax states these communication rules explicitly.

We also propose some mechanisms for implementing the new syntax on distributed memory machines.

1 Description

We propose an extension to imperative languages that adds semantics for side-effects. Our first approach is to place as much of the syntax in comments as is reasonable, the goal being backwards compatibility with the original language. If this proves possible, serial programs can be incrementally enhanced to add parallelism while remaining valid serial programs. Incremental enhancement is valuable from a programmer productivity perspective.

This first analysis addresses only automatically breaking up nested loops that operate on multi-dimensional matrices that are accessed through pointers and indirection. However, the side-effect semantics should generalize to any data-structure if one uses the DKU pattern [cite DKU paper].

1.1 Iteration Space

Before going further, we define a side-effecting read or write as one that calculates its address using the contents of one or more dynamically modifiable memory locations. This happens when using a variable to index an array, or when using indirection through either pointers or indexes. For example, `A[ i ]` may be side-effecting because `A` is treated as a pointer and `i` is added to it. The address accessed is unknown to the compiler because `i` is a variable whose value is, in general, unknown to the compiler. When the address accessed is unknown to the compiler, and multiple processors might be calculating the same address, then the compiler must assume that side-effects are taking place. A more obvious example of side-effecting involves indirection: `*P` is indirection through a pointer, so any read or write stated this way is side-effecting. Likewise `A[ B[ iter ] ]` uses the contents of a memory location to determine the address of another memory location, so reads or writes using this are also side-effecting.

The side effect semantics we propose use the concept of iteration space. When accessing a data structure by iterating through values of an iterator variable, each variable defines one dimension of the iteration space. Thus, one set of iterator values equals one point in the space.

At each point in the iteration space, zero or more side-effecting reads and/or writes occur. To see this, imagine collecting the trace of a run, then marking each instruction with the set of iterator values at the start of the instruction. When one goes back and re-scans the marked trace, one notes the iterator values at each side-effecting instruction and assigns that instruction to the set of iterator values. Because a set of iterator values is one point in iteration space, one has thus collected all side-effecting operations that occur at each point in iteration space.

More usefully, one could relate iteration points to source code. First, make a label for each side-effecting read or write in the source. Then go down the trace and mark each side-effecting instruction with the source-label it represents. As one goes, down the trace, one keeps a list for each source-label. When a side-effecting instruction is encountered, one notes the iteration-values and the source-label, and adds the iteration-values to the source-label's list. One can now attach each list to the label in the static source-code. Now, when looking at the source, for each side-effecting operation, one has the set of all iteration points in which the source-operation sends or receives a side-effect communication.

To make an easier to manipulate notation, the lists are made into lists of tuples. Each operation's list is modified by attaching the operation's source-label to each iteration point. Each iteration point becomes a tuple of iteration values plus source-label. When done, each side-effecting instruction performed in the trace appears in exactly one location in one list of tuples.

Now, structure can be seen among these tuple-lists. Nearly every algorithm representation that contains loops will have structure among its tuple-lists. In the trace, when a read of a particular address comes sometime after a write to the same address, a communication has taken place from the write to the read. If one graphs all such communications and looks for patterns, it will become clear that invariants exist among the values of tuples that communicate. An example of such an invariant might be "Write 5 only communicates to Read 7, and every time, iterator j in write 5's iteration-point is one more than iterator j in Read 7's point".

These invariants were known to the programmer as they wrote the sequential code. They crafted the code such that the sequential semantics enforced those invariants. Our proposed strategy in the context of parallelism, then, is to provide syntax to state these invariants explicitly.

The programmer labels each write operation that can cause a side-effect and labels each read operation that can receive a side-effect. They then state conditions on communication from writes to reads. The conditions are relationships among iteration variable values when the write happens and iteration variable values with the read it communicates to happens. In effect the conditions relate tuple sub-spaces – the iterator sub-spaces of the writer that are allowed to communicate to iterator sub-spaces of the reader. In practice, the programmer states a generic point in iteration space for the write, then states offsets to the point(s) where the read-statement receives the communication. This is similar in concept to stating an induction hypothesis.

To make this more concrete, we introduce the syntax. First consider this example of labelling writes and reads in the source, where an array is pointed to by a pointer `P`. The line:

```
*P[i] = func( *P[j] );
```

would be broken into two pieces, and labels added, to become:

```
a = *P[j];          //: read1
*P[i] = func( a ); //: write1
```

Where the labels are added as comments. A "`//:` " indicates that something in the proposed syntax follows. Here it is a label definition. Note that only one memory access is allowed in a labelled line, which makes it clear which memory access the label refers to.

The programmer can then make statements about communications between labels. They can either assert knowledge that they happen to know about communications, or they can state conditions that the communications are forced to obey.

Here is an example of an assertion:

```
//: assert write1 doesNotCommTo read1;
```

which states that the assignment, `*P[i]` cannot communicate to the read `*P[j]`, no matter what. The programmer wrote this because they know something about the logic in the code. For example, the domains of `i` and `j` may be disjoint. The assertion tells the compiler it is free to perform any transformation it likes without worrying about a side-effect between these two memory operations.

1.2 Parallelism

So far the discussion has been in the context of sequential code. For parallelism, the iteration space is broken into pieces. Each piece is iterated by one processor, known as a "kernel procesor". The statements about communications from writes to reads determine the communication between the pieces of iteration space, and thus between kernel processors.

As a result, the syntax must identify a kernel of code, many copies of which will execute in parallel at run-time. A kernel processor is created to run the kernel code on one piece of iteration space, then it dies. The size of the iteration sub-space given to each kernel processor, and how many kernel processors exist simultaneously is decided by the execution-model implementation (run-time).

In addition to specifying side-effect communications, the programmer must also specify how to break up the data-structure that the kernel operates on and how to put the individual results back together. The programmer does not, however, specify distribution of the pieces. It is expected that the compiler will be able to distribute pieces of data and deduce an iteration schedule to execute on each piece.

In special cases, the programmer doesn't have to specify how to break up the data-structure. For example, when the data structure is a matrix, the programmer only has to state the kernel. The compiler can automatically figure out how to perform division and re-construction. This is the case we consider in the rest of this paper.

In general, every side-effecting read stated in the kernel might receive from every side-effecting write, so every possible communication must be made definite for the compiler. All possible side-effecting communications in the kernel must be made definite for the compiler because when multiple kernels are running simultaneously, the time-ordering of side-effecting communications between them is non-deterministic. A subtlety at work is that one of the writes may be communicating to a given read at one combination of iterator values, then a different write at another point. Thus, the compiler must know the communication to each read from every write. All possibilities for such communications must be either known to never occur or else the communication must be stated. Any possible communications not explicitly stated are asserted to never occur (and the execution-model-implementation might check these assertions at run-time, at least in debug mode).

To define the kernel code, the proposed notation defines the start and end of a kernel via `startKernel` and `endKernel` statements. Communication statements come after the kernel ends.

1.3 Syntax for Commands

Now we propose syntax for commands that the run-time is expected to enforce. They start by stating the label of either a read or a write then a command then iterator value relationships.

Here is an example, which should be equivalent to a Sieve block⟨cite|Lokhmotov08⟩:

```
int P[][];
P = new int[ dataDepVar ][ dataDepVar ];  (just saying that cannot predict size)

//: startKernel K1, takesPiece (P[Xb, Xe][Yb, Ye])
for( int i = Xb; i < Xe; i++)
 { for( int j = Yb; j < Ye; j++)
    { a = P[i][j];    //:read1
      P[j][i] = f(a); //:write1
    }
 }
//: endKernel K1
//: write1 mustNotCommunicateTo read*;
```

The last comment is a command to the run-time that tells it to ensure that write1 does not communicate to any read in a different kernel processor. One possible way to implement this is by queueing up all writes and performing them after all kernel processors have completed, as was introduced by Lokhmotov ⟨cite|Lokhmotov08⟩.

An example of a more general side-effect command is:

```
//: write1 communicatesTo read1 at (remote(i) == local(i), remote(j) == local(j) + 1);
```

This side-effect command tells the EMI (execution-model-implementation) to make sure that write1s that take place in one of the kernel processors must communicate to the read1s in the appropriate other kernel processors. Notice that `i` in a given kernel processor runs from `Xb` to `Xe`. Thus the constraint that `remote(i) == local(i)` means that only those processors with an over-lapping `X` range will communicate. The additional constraint from the statement `remote(j) == local(j) + 1` means that only processors that also have over-lapping `Y` range will communicate. In practice, the compiler will probably choose pieces with disjoint `X` values, so only pieces with adjacent `Y` values will communicate.

As for what happens at run-time, the explanation bifurcates, depending on whether the scheduling onto processors is static or dynamic, and whether the target machine is shared memory or distributed memory:

- If scheduling is dynamic, then the compiler must prepare functions for use by the run-time that divide up the data and tell the run-time which pieces communicate to which others.
- If compiling for a run-time to dynamically divide the data, then the compiler will have to generate three things: a function that performs the division; a function that determines which pieces communicate to which other pieces; and communication and synchronization commands that go inside the kernel code and take the results of the "whichPieceToWhichPiece" function.

For the first implementation of such a side-effect command, it may be easier to restrict access through pointers to go to only arrays, and to only compile for shared-memory. In this case no communications need be generated by the compiler, only synchronizations, and no explicit division need be stated by the programmer. In the more general approach, the programmer writes a divider and an undivider.

During the kernel run, the iterator values must be turned into a destination for each communication. To do this, one idea might be for the compiler to make a function that is used by each kernel processor. The function takes the current iterator values and the ID of the calling kernel and returns an ordered set of IDs of the kernel processors that must be communicated to for that set of iterator values.

Notice that the command does not have to use only spatial considerations. The code could have an iterator that is not directly used to calculate a position in the array. The communication pattern can still use the value of this iterator, especially if it causes repeated passes through the array.

For example:

```
int P[][];
P = new int[ dataDepVar ][ dataDepVar ];  (just saying that cannot predict size)

//: startKernel K2, takesPiece (P[Xb, Xe][Yb, Ye])
for( int k = 1; k < 10; k++ )
 { for( int j = Yb; j < Ye; j++)
    { for( int i = Xb; i < Xe; i++)
       { a = P[i][j];        //:read1
         P[j][i + 1] = f(a); //:write1
       }
    }
 }
//: endKernel K2
//: write1 commsTo read1 at ( remote(i) == local(i) + Xb, remote(k) + 1 == local(k) );
```

Here `j` does not appear in the side-effect command. Thus, for a pair of `local (i,k)` with a `remote(i,k)` that satisfies, the communication takes place for every `j`. Notice also that `Xb` appears in the side-effect command. Finally, notice that although `k` appears in the side-effect command, it is not used to calculate position in the array. The statement `remote(k) + 1== local(k)` is saying that the result from one iteration of `k` is communicated to the next iteration.

The implementation of this command might use a barrier that blocks until all kernel-processors have completed a given value of `k`. Such an implementation will require a double-buffer. One buffer receives the result from the current iteration in a different kernel-processor, while the other buffer is used in the current iteration. A barrier at the end of each iteration of `k` ensures that all kernel-processors are calculating the same iteration. The buffers are switched right after the barrier is passed.

There are four equivalent ways of stating any communication command. They are illustrated here with a randomly made-up example of a communication that uses inequalities.

The four forms of syntax are:

```
//: write1 communicatesTo read1 at   ( remote(i) >= local(j),  remote(j) == * );
//: read1 receivesFrom write1 when   ( local(i)  >= remote(j), local(j)  == * );
//: write1 communicatesTo read1 when ( local(i)  == *, local(j)  <= remote(i) );
//: read1 receivesFrom write1 at     ( remote(i) == *, remote(j) <= local(i)  );
```

All four statements are equivalent. The first label is always the 'local' entity and the second label is the 'remote' entity. Thus in the first command, `write1` is the local entity, and `read1` is remote. Thus in the boolen expressions at the end of the command, the keywords `local` refers to the iteration-point of the kernel processor executing a `write1`. Likewise `remote` refers to any other kernel processor executing a `read1`.

The meaning of the command is that the write communicates its value to the read whenever the j value in the writing kernel-processor is less than the i value in the reading kernel-processor. Even though all four forms are equivalent for this example, there are many cases when it is much more convenient for the programmer to state with one of these forms vs the others. The write received for a given remote j is re-used for every local i and local j. This communication pattern is non-deterministic. It is very likely a bug (this was just a randomly made-up example).

The syntax rules are fairly intuitive: one has the choice of `communicatesTo` or `receivesFrom`. If `communicatesTo` is used then the local entity must be a write. If `receivesFrom` is used the local entity must be a read. Further, one may use `at` or `when`. `at` means to state the conditions for the remote entity to perform the communication action, while `when` means to state the conditions for the local entity to perform the communication action.

If one uses `at` then one gives conditions of the form `( remote(` *iteratorName* `)` *comparisonOperator* `local(` *iteratorName*`),` *repeat for other remote iterator names*. Whereas if one uses `when` then one gives conditions of the form `( local(` *iteratorName* `)` *comparisonOperator* `remote(` *iteratorName*`),` *repeat for other local iterator names* `)`. One may also combine comparisons using the syntax of `if` statements.

To use the extension for non-matrices, the programmer would add a divider and un-divider, along with a counter in each while-loop. The counter creates an iterator that can be used in communication statements.