Fleet Programming Paradigms

Amir Kamil
Computer Science Division, University of California, Berkeley
kamil@cs.berkeley.edu

UCB-AK07

October 17, 2008

1 Introduction

In this memo, I discusses the application of three parallel programming paradigms on Fleet [4]: single program, multiple data (SPMD) parallelism, phased pipeline parallelism, and dynamic task parallelism. This discussion is in the context of the Admiral 2 language[3].

2 The Streaming Model

I intend to use the streaming model as the basis for Admiral 2, particularly that of the StreamIt language [1]. In this model, a program consists of multiple independent *filters*, each of which consists of a steady-state function that computes a fixed number of outputs from a fixed number of inputs. These filters are composed hierarchically into a streaming graph, where edges correspond to the flow of data from one filter to another. Filters only communicate through these data streams[1], and they can be run concurrently.

The streaming model is conceptually similar to using inifinite moves to set up virtual ships in Fleet. The idea in both cases is to perform the same operation on many inputs and therefore amortize the cost of initialization across them. There are other benefits (and some drawbacks) to StreamIt of Fleet that I've already discussed in a previous memo [2].

3 SPMD Parallelism

3.1 Background

The *single program, multiple data (SPMD)* model of parallelism is very common in high performance computing. It is characterized by a fixed number of parallel threads that independently execute the same code. Depending on the language, these threads communicate by message passing or by accessing shared data. Synchronization is generally done through barriers and other collective operations, such as reductions and scans.

There are a number of advantages to the SPMD model. In general, SPMD algorithms operate almost entirely on local data, with communication relegated to specific points in the program. This simplifies the comunication scheduling

[1] Actually, StreamIt also has support for *teleport messaging*. However, I do not expect to support them at this point.

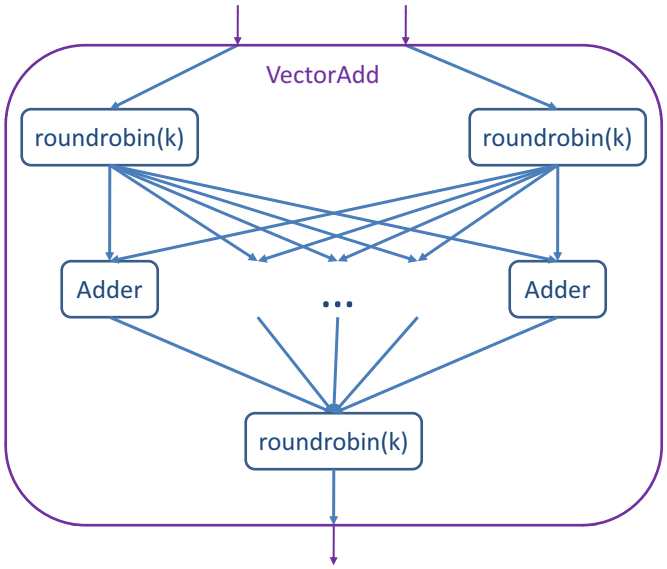

Figure 1: SPMD-style vector addition. Each adder computes the sum of k consecutive pairs of elements.

problem. Compilation is also easier since each thread in a SPMD program executes the same code. SPMD programs are generally written to execute correctly regardless of the number of threads used at runtime, so they can scale from a single thread to many thousands.

3.2 Applications

I examine two applications written in the SPMD style: simple addition of two vectors and matrix-vector multiplication.

3.2.1 Vector Addition

Element-wise vector addition is one of the simplest examples of a SPMD algorithm. The algorithm consists of three phases. In the first phase, the two vectors are distributed to each of the threads. The actual work is done in the second phase, where each thread computes the sum of each pair of elements that it received. In the final phase, the resulting vector is assembled from the partial results of each thread.

The following is the Admiral 2 implementation of vector addition:

```
(int,int)->int parallel VectorAdd(int N, int k) {
  parallel {
    split roundrobin(k);
    split roundrobin(k);
  }
  parallel {
    for (int i = 0; i < N; i++) {
      add Adder;
    }
```

```
  }
  join roundrobin(k);
}

(int,int)->int filter Adder() {
  work push 1 pop 2 {
    push(pop(0) + pop(1));
  }
}
```

The corresponding stream graph is shown in Figure 3.2.1.

The algorithm starts by splitting each input between the N `Adder` "threads." Each `Adder` is given k elements at a time from each input, where the optimal value of k depends on the hardware. The results are combined by taking k values from each `Adder`, so that the output vector is in the correct order.

3.2.2 Matrix-Vector Multiplication

Dense matrix-vector multiplication is another problem that is suitable to a SPMD implementation. I consider three algorithms for this problem: row split, column split, and blocked split[2]. All three algorithms use the following `dotprod` function as each of their P "processors:"

```
(int,int)->int filter dotprod(int K) {
  int vect[K];
  prework pop (K,0) push 0 {
    for (int i = 0; i < K; i++) {
      vect[i] = pop(0);
    }
  }
  work pop (0,K) push 1 {
    int res = 0;
    for (int i = 0; i < K; i++) {
      res += vect[i] * pop(1);
    }
    push(res);
  }
}
```

The `dotprod` functions reads and stores K values from its first input as part of its initialization. In its steady state, the function reads a sequence of K values from its second input, computes their dot product with the stored values, and produces the resulting scalar at its output.

The column-split algorithm divides the $N \times N$ matrix by assigning N/P of its columns to each processor. The N-element input vector is accordingly divided among the processors, each receiving the appropriate N/P elements. Each element in the output vector is simply a sum of the partial results from each processor, so a reduction is performed to compute the final value of each element. Figure 3.2.2 illustrates this algorithm.

The row-split algorithm similarly divides the matrix by assigning N/P of its rows to each processors. Each processor now needs access to the entire input vector, so it is copied to each of them. Each element in the output

[2]The algorithms assume that the matrices are represented in row-major format.

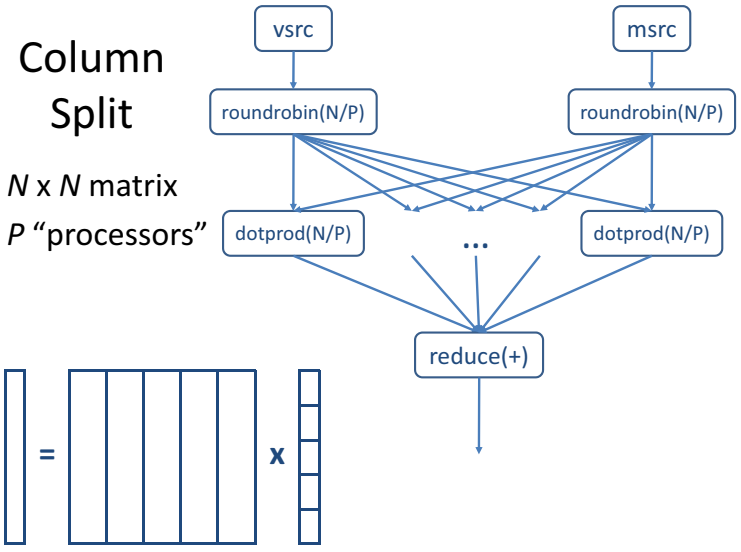

Figure 2: Dense matrix-vector multiplication using a column-split scheme. Each of the P "processors" operates on its own set of N/P columns.

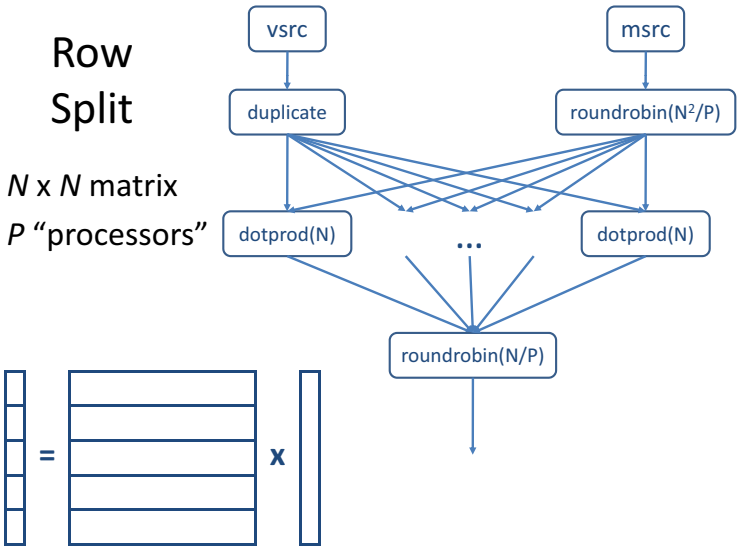

Figure 3: Dense matrix-vector multiplication using a row-split scheme. Each of the P "processors" operates on its own set of N/P rows.

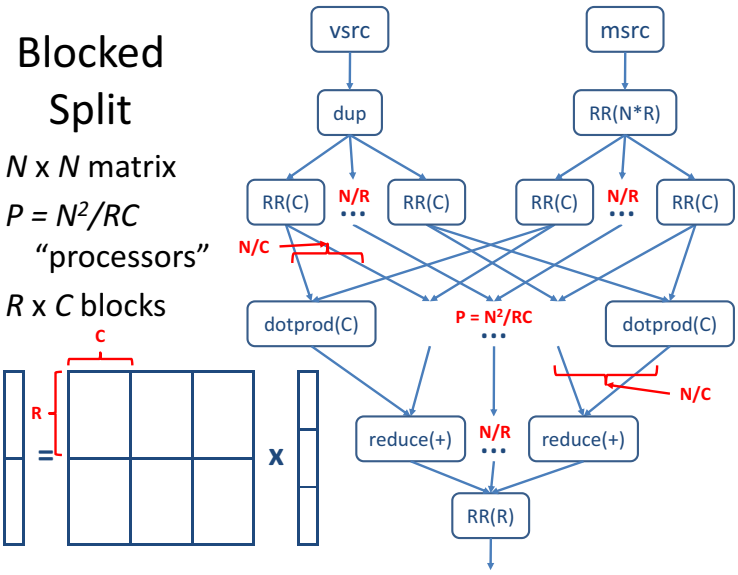

Figure 4: Dense matrix-vector multiplication using a block-split scheme. Each of the P "processors" operates on its own block of elements.

vector is computed entirely on a single processor, so the output vector is generated simply by taking the output of each processor. Figure 3.2.2 illustrates this algorithm.

The blocked-split algorithm is the most complex. It divides the matrix into blocks of size $R \times C$, where $P = N^2/RC$. Each processor now needs access to C elements from the input vector, so the appropriate elements are copied to each processor. Each element in the output vector is partially computed on C processors, so a reduction is performed across each set of C processors to obtain the final value. The output vector is then assembled from each set of processors. Figure 3.2.2 illustrates this algorithm.

3.3 Implementation

I only intend to compile for a single Fleet processor, and I assume that it is reasonable to require recompilation in order to run on a different Fleet implementation. As such, the number of processors P will be a compile-time constant, rather than a run-time constant as in other SPMD implementations. This should allow the compiler to perform all computation and communication scheduling statically.

4 Phased Pipeline Parallelism

4.1 Background

Pipelining is the main mode of parallelism in the streaming paradigm. This model of parallelism consists of a sequence of concurrently executing stages that perform the same operation on many pieces of data. Each stage consumes data from its predecessor, operates on it, and passes it on to its successor. The stages operate on different pieces of data at any particular time, allowing them to be run in parallel.

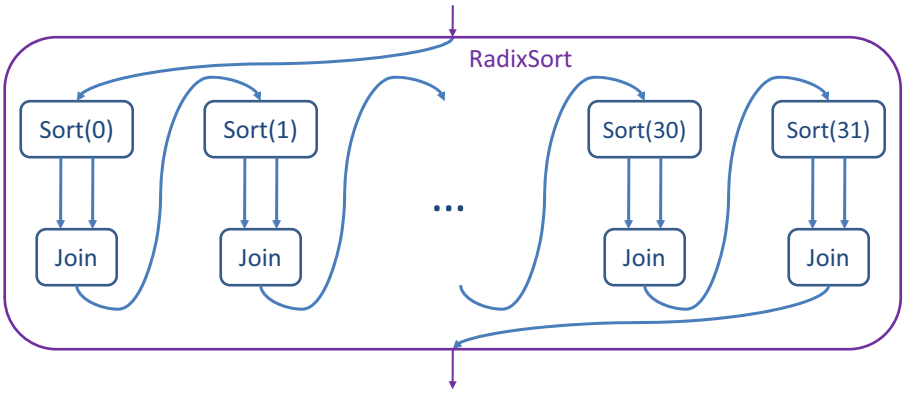

Figure 5: Pipelined, least significant digit radix sort.

StreamIt operates on the assumption that the input to each pipeline stage is infinite[3]. This assumption does not allow what I call *phased pipeline parallelism*. In this model, the program executes in phases, with each phase consisting of one or more adjacent pipeline stages. The input data to a phase is finite, and when the data is exhausted, execution continues to the next phase. Phases can be run concurrently to increase the level of parallelism.

4.2 Radix Sort

Radix sorts are naturally expressed in a phased pipeline manner. The following is a radix 2, least signifcant digit, integer radix sort in Admiral 2:

```
int->int pipeline RadixSort() {
  for (int i = 0; i < 32; i++) {
    add Sort(i);
    add Join();
  }
}

int->(int,int) Sort(int k) {
  int mask = 1 << k;
  work pop 1 push [(0,1), (1,0)] {
    int in = pop();
    if ((in & mask) == 0) {
      push(0, in);
    } else {
      push(1, in);
    }
  }
}
```

[3]StreamIt does allow dynamic reconfiguration of the stream graph, which can be used to remove or replace a filter if its input is finite. Doing so, however, requires a significant amount of work from the programmer.

```
(int,int)->int Join() {
  boolean left = true;
  work pop [(0,1), (1,0)] push 1 {
    if (left) {
      int in = pop(0);
      if (in == EOS)
        left = false;
      else push(in);
    } if (!left) {
      push(pop(1));
    }
  }
}
```

The `Sort` stages each act on a particular digit, steering values for which that digit is not set to the first output and the rest to the second. Each `Sort` stage is followed by a `Join` stage, which simply takes values from the first input until it is exhausted and then reads from the second input[4]. For 32-bit integers, 32 `Sort` and `Join` stages are required.

A major issue in this algorithm is that a `Join` stage must exhaust its first input before proceeding to its second. Thus, support for finite inputs is necessary, which is not present in the pure streaming model. Here, an out-of-band value `EOS` is used to mark then end of a stream. The program above assumes that this marker is automatically propagated to the outputs of a stream one all its inputs have been exhausted.

4.3 Implementation

4.3.1 End of Stream

As alluded to above, the Admiral 2 compiler will automatically generate the code to propagate end of stream markers. Since the Fleet hardware has 37-bit words, the upper 5 bits (or 1 bit, in the case of 36-bit values) can be used to signifiy the end of a stream. It should be possible to detect the `EOS` value in a dock and react to it as required.

4.3.2 Scheduling

Each radix sort stage can be expected to operate at half the throughput of its predecessor, so it would be inefficient to execute all stages simultaneously. Instead, a fixed number of stages should be scheduled to execute concurrently, starting at the beginning. Then as each stage exhausts its input, it should be torn down, and the next idle stage should take its place. It should be possible to implement this scheme using only static scheduling.

5 Dynamic Task Parallelism

5.1 Background

The most common model of parallelism is *task parallelism*. In this model, multiple threads concurrently execute different pieces of code. In *dynamic task parallelism*, these threads are created and destroyed as the program executes.

[4]Note that the program as written assumes unbounded buffering of inputs. If this is not the case, the code can be altered to explicitly buffer in main memory.

The StreamIt language only supports static task parallelism: the structure of the stream graph must be known at compile-time[5]. Unfortunately, many algorithms do not fit into the static task parallelism model, especially recursive algorithms such as sorting. For example, the StreamIt implementation of bitonic sort requires that the input size be specified at compile-time, so that the stream graph can be statically constructed.

5.2 Recursive Sorts

I intend to examine multiple recursive sorting algorithms using task parallelism. The following is a recursive merge sort in (pseudo-)Admiral 2:

```
int->int parallel mergesort() {
  boolean end = false;
  init pop [0,1] peek 2 push [0,1] {
    int i1 = peek(0);
    int i2 = peek(1);
    if (i2 == EOS) {
      end = true;
      if (i1 != EOS) {
        push(pop());
      }
    }
  }
  if (!end) {
    dynamic {
      split roundrobin;
      parallel {
        add mergesort();
        add mergesort();
      }
      add mergejoin();
    }
  }
}

[int,int]->int filter mergejoin() {
  work pop [(1,0),(0,1)] peek (1,1) push 1 {
    int i1 = peek(0, 0);
    int i2 = peek(1, 0);
    if (i1 > i2) {
      push(pop(0));
    } else {
      push(pop(1));
    }
  }
}
```

[5]StreamIt does allow dynamic reconfiguration of the stream graph, but all possible configurations must be known at compile-time.

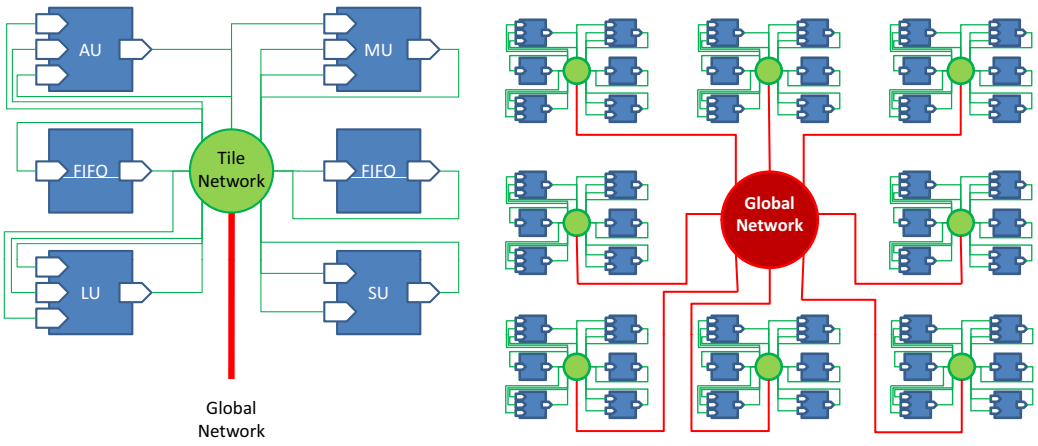

Figure 6: A Fleet processor composed of tiles.

The `mergejoin` function simply reads from its two inputs and produces as output the greater input. The `mergesort` function initially examines its input to determine if it contains more than one element; if not, it simply copies its input to its output. If there is more than one element, then it recursively spawns two `mergesort` instantiations and splits the input between them. It also spawns a `mergejoin` to combine their results.

5.3 Implementation

A runtime scheduler is required in order to support dynamic task parallelism. This scheduler may run on a separate general-purpose processor or on a subset of the Fleet processor itself. When a new "thread" is spawned, the scheduler must perform the following actions:

1. Allocate a set of ships to run the thread.
2. Rewrite the code to run on the target ships.
3. Schedule communication.
4. Dispatch the code.

I discuss each of these steps in more detail.

5.3.1 Ship Allocation

The scheduler must allocate a set of ships for a new "thread" to run on. In order to simplify the problem, I intend to divide the Fleet processor into *tiles*, where each tile contains at least the fundamental set of ships*CITATION?*. As a result, the scheduler only needs to allocate tiles to threads instead of individual ships. The compiler will cooperate by producing code that runs on a single or a set of multiple tiles.

It is unclear what to do if not enough tiles are available, since Fleet does not support preemption. The simple solution is to wait until tiles are available, but this may result in deadlock for some programs. Another possibility is to arrange for the code to cooperatively multitask, so that each "thread" checks in with the scheduler at regular intervals. However, it is not clear that this will solve the deadlock problem, since the interval cannot depend on a clock.

5.3.2 Code Rewriting

Since many Fleet instructions include destination addresses, they may need to be rewritten depending on the ships allocated to run a particular "thread." In order to minimize the amount of rewriting required, I assume the following about the Fleet hardware:

- The switch fabric uses relative addressing.
- All tiles have the same relative layout for their ships.

If the above conditions are satisfied, then communication internal to a tile need not be rewritten, as the relative addresses would be the same on any tile.

5.3.3 Communication Scheduling

The scheduler must set up flow control in the newly spawned "thread" to prevent deadlock from occurring in the switch fabric. I assume that in addition to the addressing and layout conditions mentioned in §5.3.2, the switch fabric uses a hierarchical structure such that each tile has an internal network connecting its ships and an external link to other tiles. If this is the case, then internal communication scheduling can be done at compile-time, and only external communication need be scheduled dynamically. The scheduler must consider the external communication requirements when spawning a thread, and if there is insufficient capacity in the network, it will likely have to react analogously to the situation where there aren't enough tiles to run the thread.

5.3.4 Code Dispatch

The final dock instructions must eventually be dispatched to their destinations. This should be done by the memory unit in a tile, so that the instruction destinations need not be rewritten and so as to not clog up the inter-tile switch fabric.

6 Conclusion

In this memo, I have argued that the streaming model can be extended to support single program, multiple data parallelism, phased pipeline parallelism, and dynamic task parallelism. I intend to further explore the details of what is required to support them and to begin implementing them in a compiler.

References

[1] Streamit language specification, version 2.1, September 2006. http://cag.csail.mit.edu/streamit/papers/streamit-lang-spec.pdf.

[2] A. Kamil. StreamIt on Fleet, July 2008.

[3] A. Kamil. The Admiral 2 Language, 2008. Work in progress.

[4] I. E. Sutherland. FLEET - A One-Instruction Computer, August 2005. http://research.cs.berkeley.edu/class/fleet/docs/people/ivan.e.sutherland/ies02-FLEET-A.Once.Instruction.Computer.pdf.