6.033 Design Project 1

I. Assignment

There are two deliverables for Design Project 1:

1. Two copies of a design proposal not exceeding 800 words, due in Recitation 6.
2. Two copies of a design report not exceeding 2,500 words, due in Recitation 13.

As with real life system designs, 6.033 design projects are under-specified, and it is your job to complete the specification in a sensible way given the overall requirements of the project. As with designs in practice, the specifications often need some adjustment as the design is fleshed out. We strongly recommend that you get started early so that you can iterate your design. A good design is likely to take more than just a few days to put together.

II. The Problem

Data flow graphs are a popular parallel programming model. In a dataflow graph, a series of independent blocks of code, or operators, are connected together in an acyclic directed graph. An edge from an operator A to an operator B indicates that A sends messages to B. This is the only mechanism through which operators may exchange data – in dataflow programs there are no global variables or shared state between operators.

Because of this limited form of sharing of data between operators, dataflow programs are very easy to parallelize. Each operator is an independent processing task that can be run on a separate processor, with queues or other communication primitives connecting the operators. It is not uncommon for such programs to yield nearly linear speedups when they are run on a machine with more processors.

Different data flow systems provide different collections of operators. An example dataflow graph that uses simple data processing operators of the sort used by a database system is shown in Figure 1 below. Data enters the system at an input operator, and flows onwards through "downstream" operators, ending up at an output. The data flowing between the operators can be arbitrary objects (e.g., arrays, tuples, or other data structures), which we call *data elements.* In this case, the dataflow graph reads input elements corresponding to stock quotes and sends those elements through a series of downstream operators that find stocks whose variation in price over the last 10 minutes is greater than some threshold value *k*. Each "Var" operator emits one element every 10 minutes. Also note that operators may have multiple inputs and outputs, although there is at most one edge connecting to any given input or output.

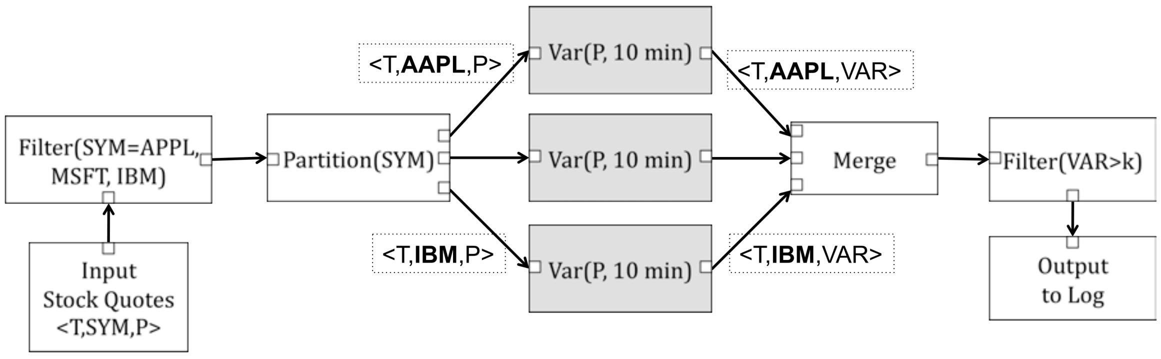

Figure 1: Example data flow graph that reads in elements representing stock trades, finds elements corresponding to certain stocks, computes the variation in price of each stock over the last 10 minutes (e.g., difference between maximum and

minimum value), and then finds stocks whose variation is greater than some threshold *k*. Stateful operators are shown in gray (see Section III.1.d). Small boxes represent input and output ports.

In this design project, your task is to design a system to schedule and execute dataflow programs efficiently on a simplified parallel (multiprocessor) computer. Your system will accept a dataflow graph, prepare it to run on the given machine, and then execute it over a supplied input data set. The objective is to run the graph so as to produce correct outputs in minimal time. Note that different dataflow graphs and hardware configurations will present different workloads, for which different execution strategies may be optimal.

The key challenges you must confront include:

- Ensuring that your design is able to parallelize effectively on a multi-processor computer.
- Mapping operators to operating system threads, and threads to processors, in a way that minimizes memory access costs. This is somewhat tricky as your design must support a multi-processor architecture where processors have substantially faster access to local memory than to memory allocated by other processors.

These two challenges present a tradeoff: running all operators on one processor would minimize time spent waiting for memory, but would leave the other processors idle. Conversely, running each operator on a different processor may spend lots of time waiting for memory loads from other processors' memory.

In addition to managing this basic tradeoff, you may want to consider optimizations that improve parallelism by reorganizing the data flow graph in various ways. For example, if an operator doesn't maintain any state from one element to the next (as in the `Filter(SYM=…)` operator in Figure 1), it can be replicated and run on several processors simultaneously to improve parallelism.

The remainder of this document explains the constraints and assumptions of this project in more detail. Section III explains the APIs you must implement and hardware model you should assume. Section IV describes some workloads that you should consider in your evaluation. Section V explains what we expect and how we will evaluate your work. Section VI describes the format of the design proposal. Section VII describes the format of the design report.

III. Dataflow Engine APIs and System Model

In this section, we describe the APIs you must implement and the hardware your system will run on.

III.1. Programming Model

The programming model is defined by a graph of operators connected by edges. Each operator defines a set of input and output links. An operator receives data elements from an upstream operator via one of its input links, and emits data elements to downstream operators via a specified output link.

The programming model you will implement is known as an asynchronous dataflow model, meaning that operators may produce data at irregular rates, and two data elements that take different paths through the dataflow graph are not guaranteed to traverse those paths in lock step. For operators that have more than one input, the system provides no guarantees about the relative arrival time of elements on those inputs.

Your system must take as input a program expressed as a dataflow graph, and then run the program. Your system is responsible for scheduling operators to run on specific processors and for passing data elements from operator to operator. Your system must also use synchronization primitives to ensure that connected operators running in different threads properly exchange data (you may wish to refer to Chapter 5.6 of the course notes for more information about such coordination primitives.)

III.1.a. Graph Data Structure

As part of your design you must design a data structure to represent the dataflow graph. You can assume that you are given access to an array of operators, with each element in the array representing one element in the graph as the programmer described it. You have access to the following information for each operator:

- A list of upstream input operators it reads data from (numbered from 1…n)
- A list of downstream output operators it sends data to (numbered from 1…m)
- An `init()` function, that initializes the state of the operator
- A `body()` function, that runs the operator
- A flag that indicates whether the operator is stateless (see Section III.1.d)
- A flag that indicates whether the operator is an input or an output terminal – that is, whether the node is a source that reads data from disk or an output that writes results to a file.

As described in Section III.3, you also may assume you are given

- For each input or output, the average size (in bytes) of each element passed on that stream, as well as the access frequency observed when `body()` processes an element of the stream
- For each operator, the average size (in bytes) of that operator's internal state, as well as the access frequency for that state.

This collection of information about each operator is shown schematically in Figure 2.

You will need to specify the data structure you use to keep track of the graph your program is currently running, the state of running operators, and the links between nodes in the graph.

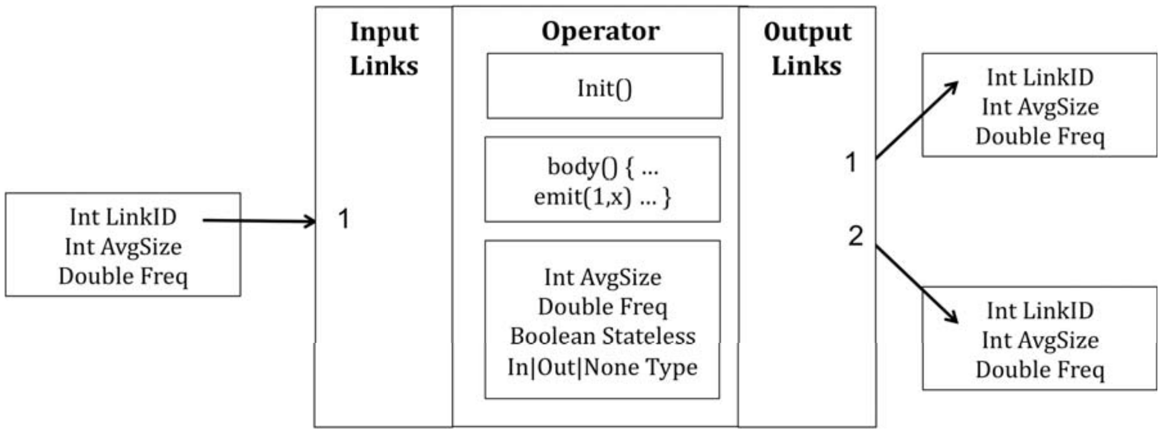

Figure 2: Dataflow graph input specification.

III.1.b. `init`, `body`, and `emit` methods

Before the dataflow graph starts running, your system must call `init()` once for each operator to initialize the operator and its local state. `init()` may also specify configuration parameters needed to instantiate the operator (you don't need to worry about the details of those parameters, as they may vary from operator to operator). `init()` returns a reference to the state the operator will use as it runs; this includes both the mutable state updated by stateful operators (see Section III.1.d), as well as any immutable state the operator references (but does not write) when it runs.

Your system should call the `body(state, …)` function to cause an operator to run. When an operator runs, it will consume data from its inputs and possibly produce outputs (possibly using the `state` pointer created by `init()`). To avoid overly constraining your design, we have been intentionally vague about the exact signature of the `body()` function, when your system must call the function, or how the function consumes data from its inputs. Many different designs are possible; for example:

- You could provide an input queue for every input of an operator, and arrange that the `body()` function takes a queue as an argument. Then, whenever a data element arrives on an input, your system could call the `body()` function from an idle thread, passing in the queue associated with that input. In this case, the `body()`function would de-queue the next element from the specified queue and process it, or could choose to leave the element on queue while it waits for additional inputs.
- You could also design your system so that every operator runs in a separate thread, and that there is again an input queue for every input of an operator. Each operator thread would then run in an infinite loop, repeatedly calling the `body()` function. This function would scan input queues, consuming data and producing outputs. If no data is available on any input, the thread might sleep, waiting for input, or it might explicitly signal one of its upstream parents to cause it to produce data.

When the `body()`function runs, it may produce zero or more output elements in response to every input (i.e., not all inputs produce an output, and some inputs may produce multiple outputs). To signal that it has produced data, body will call a function:

```
emit(int output, Object element);
```

once per output element, indicating which output the element should travel along (e.g., the partition operator in Figure 1 has 3 outputs). The implementation of `emit()`is up to you—you must only ensure that output elements are eventually passed to the body of the next downstream operator or operators. You can assume `emit()`has access to internal state of your system, for example, so that it can determine which operator to run next. You can also assume that the language runtime will garbage collect any data elements that are no longer in use—you do not need to worry about the details of data element de-allocation.

As an example, if we define `body()` to accept the active input queue as an argument, then the `body()` function for a filter operator is given below. Filter is a generic version of the operator used to determine if `var>k` in Figure 1–it applies a specified predicate to a particular field of each input element. Here we assume that the `body()`function takes as input an array of input queues that it can read data from (your implementation need not work this way), and that `body()` is called once per input.

```
Structure PredicateState {
      Predicate p;  // the predicate we apply to each input
                    // data element – specified in init(…)
      int inputFieldId;  //the field to apply p to
}
Body (PredicateState ps, InputQueue[] qs) { //inputs number from 1…n
      try {
            DataElement de = qs[1].pop();
            if (ps.predicate(de.getField(ps.inputFieldId)))
                emit(de,1);
        } catch Empty {}
}
```

III.1.c. Running the Dataflow Program

There are many possible strategies for executing a dataflow program, and it is up to you to choose which to use. For example, activation of the dataflow program might begin at an input, recursively calling down to the next operator's `body()`function, and returning after a `body()`function returns without calling `emit()`. Alternatively, all operators might run from their own thread. To explain how your system works, your design project should include psuedocode for a function

```
      run(…) !
```

which takes as input a dataflow graph structure and executes it. You will probably want to decompose `run()`into several subroutines that prepare a graph for execution, schedule it onto one or more threads, and actually execute it. Note that the `run()`function is not required to run the operators exactly as logically specified; for example, it may replicate stateless operators onto several processors, or execute several successive operators via direct function calls in the same thread.

III.1.d. Stateful vs. Stateless Operators

A **stateful** operator maintains state from on element execution to the next (e.g., the "Var" operator in Figure 1.) Figure 3 indicates the basic difference. The distinction between **stateful** and **stateless** affects how an operator may be scheduled. The same operator may be scheduled concurrently on two processors only if the output of the operator is the same as it would be if it was run serially. While this is true of stateless operators, it is not generally true of stateful operators.

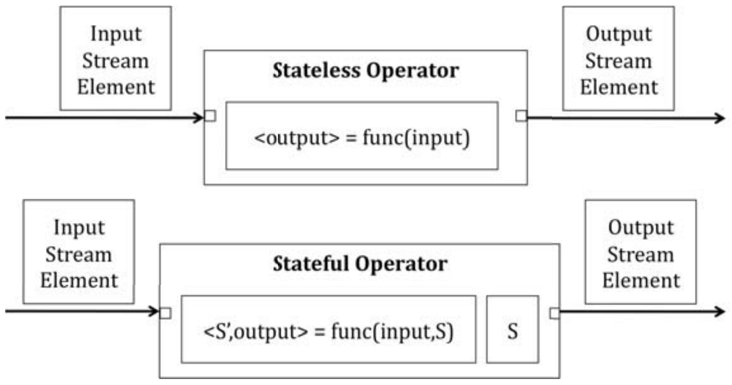

Figure 3: Stateless and Stateful operators; stateful operators manipulate some state S as elements are processed.

III.2. Platform Model: NUMA Multiprocessor

Many modern multiprocessor architectures implement Non-Uniform Memory Access (NUMA) where memory access time for a given memory location may vary depending on the processor that is accessing the data memory.

Your system will run on the simplified NUMA architecture shown in Figure 4. This system has N processors, each of which has a local 1 GB memory bank. Each processor can access its own memory at a rate of 10 GB/sec, as well the memory banks of other processors at a rate of 1GB/sec.[1] A given memory address refers to the same memory cell regardless of which processor uses the address, but the access speed varies. You can assume memory latency is the inverse of memory throughput.

You can assume that 1 GB is more than enough to store all operator state, but that an unlimited amount of data will flow through the dataflow system when it is running, so it is not possible to buffer an entire sequence of inputs.

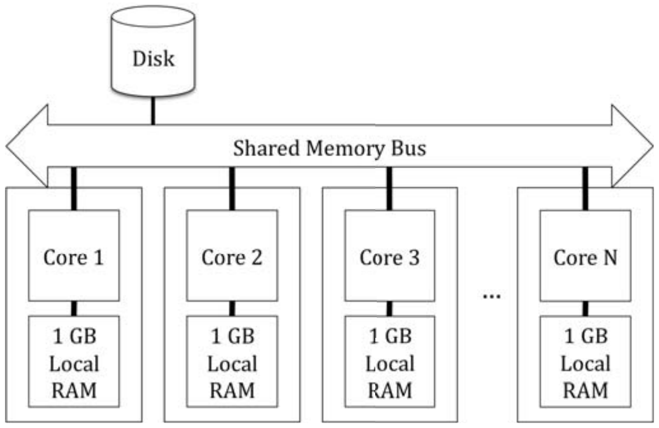

Figure 4: Multiprocessor NUMA Architecture

III.3. Operator Execution Model

Your goal in this project is to design a system that can schedule and run a dataflow graph on this architecture as quickly as possible. To understand and evaluate design choices, you will need to be able to estimate the performance that a particular design will achieve. This performance is a function of the time taken to run each operator, plus any overhead incurred by your system in executing the operators and in transferring data through the system.

The time to execute a single operator is a function of the CPU cycles required to perform the computation and the time to access memory (you can ignore costs to read input data from a network or disk in evaluating the performance of your operators.)

Since we do not know the details of each operator's implementation, we assume a fixed CPU cost to process a single data element. However, the costs of memory access by the operator will depend on how your system manages the memory in our NUMA architecture. Data access speeds vary depending on whether the data resides locally (in the bank local to the computation) or remotely (in a different bank). We can characterize the memory use by an operator into two categories:

1. **Data Elements** represent the data input to and output by operators. Data elements are initially created in the local memory bank of the processor that produced the element.

[1] This memory model is much simpler than what modern machines provide in reality; though modern machines are NUMA and do have several banks of RAM, they have a complex cache hierarchy that cause memory latencies to be much less predictable than this model. A detailed studied of different memory and cache architectures is outside the scope of 6.033, however.

2. **Operator State** represents the data associated with the operator, including the program itself and any data required to perform the computation. For stateless operators, this memory is read-only.

To help you understand the time an operator spends accessing memory, (as specified above) you can assume you are given data about each operator that specifies the quantity in bytes of each class of memory allocated for the operator, and provides an *access frequency* representing the expected number of times each memory cell is accessed in executing the operator. For example, an image classification operator might process a series of 40 KB images, with access frequency 10, meaning it accesses each image 10 times on average. Given these parameters, the memory access time is a function of the quantity of memory, the access frequency, and the access speed. The access speed in turn depends on whether the data is resident in the bank local to the processor or in another processor's memory bank. For example, if the images are resident in local RAM with an access speed of 10 GB/sec, then the time for that operator to access an image is (data size * access frequency / access speed) = 40 KB * 10 / (10 GB/sec) = 0.04 ms. Thus, the cost model for executing a single operator in our NUMA architecture is:

$$\text{Cost} = C + D_E F_E / S_E + D_O F_O / S_O,$$

where C is the CPU time required to complete the computation, and the last two terms are the access costs for memory (data size * access frequency / access speed), where D_E represents the size of the data elements and D_O represents the size of the operator data.

In addition to the time to execute each individual operator, we must also include the time required for your system to forward data from one operator to another and to choose which operator to run next. Depending on your design, your system may incur any or all of the following additional costs:

Table 1: Performance numbers that may be useful in your analysis

Thread to thread context switch	0.01 ms
Function call	0.0001 ms
Memory transfer within bank	10 GB/sec
Memory transfer between banks	1 GB/sec
Allocating new memory	0.001 ms
Acquire/release lock	0.005 ms

Note, that we do not expect you to define precise mathematical formulas that capture the performance of your design. Rather, these cost figures should guide you in your design choices (e.g. to make the trade-off between a context switch and a function call) and enable an approximate cost analysis.

III.3.a. Example: Taking Advantage of NUMA

Recall that in our NUMA architecture, local RAM accesses are 10 times faster than accesses to remote memory banks. At a high level, this means that operators that have a large amount of internal data will generally operate 10x faster if that data is in the local RAM, rather than accessing it from a remote bank.

Continuing the example of the image classification operator, suppose that it processes 40 KB video frames and the operator includes a 10 MB database of images to perform the classification. When the operator runs, it uses 10 ms of CPU time, and has an access frequency of 10x for both input and state.

$$C = 10\text{ ms}; D_E = 40\text{ KB}; D_O = 10\text{ MB}; A_E = A_O = 10$$

If all of the data is located in a remote bank, the cost is

> 10 ms + 40 KB * 10 / (1 GB/s) + 10 MB * 10 / (1 GB/s) =
>
> 10 ms + 0.4 ms + 100 ms ≈ 100 ms

If instead the operator data is stored locally, the runtime would be 5x faster:

> 10 ms + 40 KB * 10 / (1 GB/s) + 10 MB * 10 / (10 GB/s) =
>
> 10 ms + 0.4 ms + 10 ms ≈ 20 ms

Using local RAM is still faster even if we need to relocate the operator data:

> D_O / (1 GB/s) = 10 MB / (1 GB/s) = 10 ms.

Thus, the benefits of local access depend on the workload. For operators with large but sparsely used datasets (e.g., an access frequency less than 1) local access will not yield as large a performance gain.

III.4. Thread and NUMA API

In designing your dataflow engine, you may assume a threads package with synchronization primitives, as described in Chapters 5.5 and 5.6 of the course notes. Although you need not detail the implementation of primitives described in the notes, it is important to explain how your system uses these primitives and why they are suitable.

You should assume that threads run only on a single processor (e.g., once created on a particular processor, the operating system will not move the thread to another processor.)

You may also assume some additional API functions specific to our architecture:

> `Int num_processors()` This function returns the number of processors.
>
> `Void create_thread(Int processor, Code function, Pointer state)` This function creates a new thread on the specified processor and begins executing the specified function. When the function is running, all memory allocation is allocated in the memory bank local to that processor. The `state` provides a way to pass in some shared memory to a new thread (for example, an array of input queues for an operator running on the thread to read from); `Pointer` signifies that state is a pointer to an arbitrary piece of memory that can be "cast" to any type of state.

If you feel that your design needs an API function that is not discussed here, ask your TA for guidance. Changes to the API are ok, so long as you clearly explain what they do.

IV. Workloads

In this section, we give you some basic workloads on which you should evaluate the performance of your system, as well as some guidelines about what we are looking for in your design project.

IV.1. Operator Weights to Consider

Each workload is composed of a collection of operators, each having individual properties. When thinking about performance optimizations and trade-offs, there are several types of operators to consider. It is not important to handle all of these different cases equally efficiently – rather, you should choose which cases are important to optimize:

- Very lightweight operators for which the total run-time is comparable to the time that the body takes to schedule the thread (loading and unloading registers, etc.), e.g. partitioning a series of quotes based on stock symbol.
- Computationally intensive operators with little internal data, e.g. a video encoder.
- Operators that depend on a large internal dataset with high access frequency, e.g. matching an image to a large database of images.
- Mixtures of the above.
- More operators than processors.
- Fewer operators than processors.

You should consider how these different types of operators affect the way you run the workloads described in the next two sections.

IV.2. Image Classifier Workload

Safety applications and surveillance systems often require on-line image analysis that scales to large numbers of video streams. For example, a surveillance system might be designed to analyze frames to identify people and vehicles moving through a scene. For vehicles, the system would identify license plates, while for people the system would extract the face from the image and match it against a database. Dataflow systems are a promising approach to building these scalable image analyzers, because the problem is computationally intensive and highly parallelizable.

Figure 5 shows a diagram of an image classification application. The dataflow program processes a sequence of video frames by applying different classifiers, similar to a decision tree. An early classifier in the tree may determine that the frame is not relevant and drop the frame, or it may pass the frame along for further classification. Frames that pass through the tree are annotated along the way, before being merged and sent to an output log.

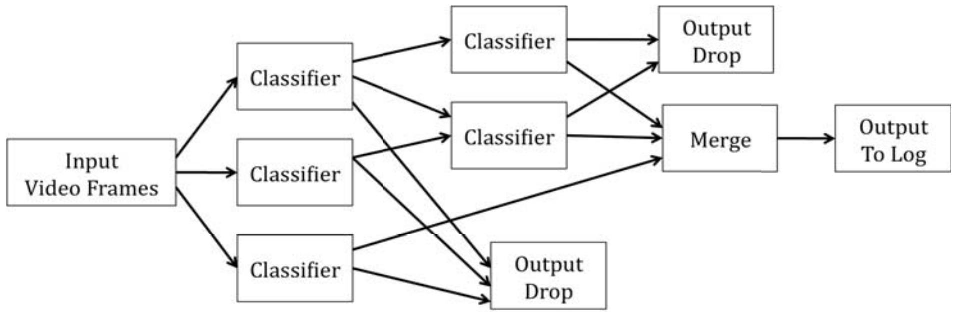

Figure 5: A dataflow graph for image classification. Each separate classifier operator implements a different classification operation, e.g., one might be for face recognition while another identifies vehicles. Some classifiers are run on all frames, while others are run conditionally. Some classes of output data are not important to the application and are dropped.

Different image classification applications might have very differently structured workflow diagrams (e.g, with more layers of classifiers, or more branches). Ideally, your system would perform well on most structures. For your analysis, you should consider the following image analysis workloads:

1. Run each frame against N classifiers, but log output only if ALL classifiers report a detection. Each frame is 40 KB and each classifier is 10 MB with an access frequency of 10 and a CPU time of 1 ms. Assume your system has N/2 processors.
2. Same workload as above, only on a system with 2N processors.

3. Run high resolution frames against N classifiers, and report all of the positive detections. Each frame is 10 MB with an access frequency of 10; each classifier is 1 MB with an access frequency of 10 and CPU time of 10 ms. Consider a platform with N processors.

IV.3 Stock Split-Adjust Workload

A stock market feed is composed of price quotes and split notifications. A company splits its stock to keep the share price in a "reasonable" range, typically between $1 and $100. When a N-way split is issued, each share of stock is converted to N shares, and the market price is divided by N. To properly analyze historical changes in price, it is important that the prices be "split-adjusted", so that pre-split and post-split prices correspond to the same fraction of the company.

Figure 6 shows a dataflow graph that split-adjusts a series of stock quotes. The split-adjust operator maintains a table of multipliers, which it updates as splits arrive.

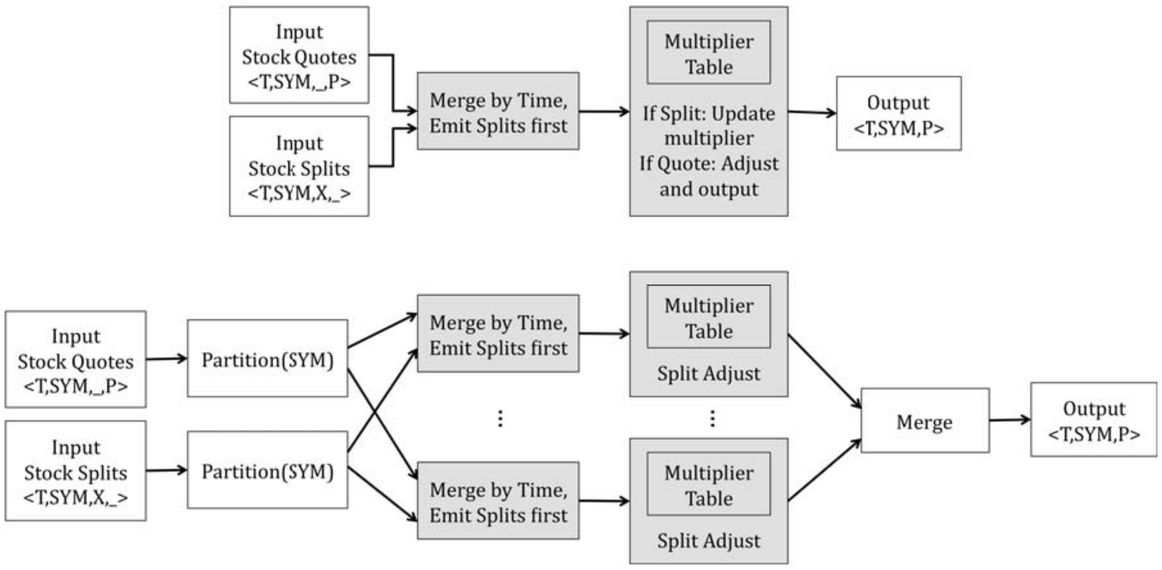

Figure 6: Two versions of a dataflow graph for split-adjusting stock quotes. Operators in gray are stateful. The lower version has been manually parallelized by partitioning the data on Symbol.

Figure 6 also shows how this dataflow can be **manually** parallelized. In the parallel version, the input elements are partitioned by ticker symbol, and the stateful adjust operator is split into multiple instances. This is possible because the mutable state for different stocks is independent. Since this type of transformation is very difficult to automatically recognize, you may assume that the developer of the dataflow figured this out and specified an explicit N-way partitioning before running the system.

The "Merge by Time" operator also deserves special mention. This operator processes two sorted inputs and merges them in sorted order, while always emitting splits before quotes. Depending on your design, implementing this correctly may require additional mechanism. If your implementation waits until one data item is available on both inputs, it can simply process quotes until the timestamp of the next quote is before that of the next split. However, if your system processes inputs independently, you may need to avoid pushing quotes far ahead of splits, because otherwise those quotes will need to be queued somewhere while waiting for the splits to "catch up".

As a simplification, you may assume that all input streams are read from disk files, meaning that your system can always read ahead on one or the other stream if needed.

For your analysis, consider the following workloads:

1. Assume the parallelized version of the split-adjust dataflow. The partition operator creates N sub-streams and there are N processors. The size of each element (either quote or split) is 100 bytes. The CPU time for each operator is 0.001 ms. The split-adjust operator is 1 MB, with an access frequency of 0.001. The other operators have negligible size.
2. Assume the same workload as above, but add in 10 additional operators after split adjust (e.g. doing simple statistical calculations). Each operator has a CPU time of 0.001 ms and negligible state.

IV.4. What we expect from your analysis

Your goal is to execute each of these workloads so as to maximize the number of output elements produced per second (the output throughput). Your system should make effective use of multiple processors (i.e., you should be able to argue that for a given fixed workload, the output throughput of your system will generally be better when the workload is run on more processors.)

There are several ways that your system may obtain parallelism, including *pipelined parallelism*, where multiple operators in a sequence run in parallel on different processors, as well as *partitioned parallelism*, where a given operator is replicated several times (as in the case of partitioning by symbol in the stock quotes workload), with each copy running over a different part of the input data. For pipelined parallelism, you may wish to identify some way to balance the load of each operator in a sequence, since output rate of a processing pipeline is determined by the rate of the slowest processor in that pipeline.

Though we do not expect you to develop a detailed numerical model of how your system would perform on these workloads, you should take into account the relative performance numbers given in Table 1. In particular, you may want to discuss the following issues in your analysis:

- Overhead of access to remote memory, as described in Section III.2.
- Depending on data element size, the relative benefit of pipelining across processors vs. building parallel pipelines on each processor.
- Thread switching overhead: As indicated in Table 1, switching from one thread to another on a single processor is relatively expensive, since it requires saving and copying thread state as well as flushing caches and virtual memory mapping structures. For graphs with many lightweight operators, optimizations that avoid this overhead may be possible.

V. Design Strategy and Evaluation Criteria

In this section, we give tips on your design and some suggestions for how we plan to evaluate it. It is important that your design be both correct and efficient. A good strategy for approaching this problem is to begin with the simplest possible ***correct*** design, and then assess which optimizations garner the most improvement on the workloads described in Section IV. Then you may choose which optimizations are worth adding to the design.

Key points to consider when ensuring a correct design:

1. A stateful operator, if scheduled concurrently in two threads, must be properly synchronized.
2. A stateful operator must process input elements in order.
3. Stateless operators may process input elements out of order or concurrently, but they must restore the original stream order before reaching an output or a stateful operator.

4. Queuing inside the dataflow network is constrained to fit in memory. Think about what happens if some operators or inputs are faster than others.
5. It must be possible to correctly implement sorting operators that merge ordered streams.

Additionally, we will be evaluating your design based on how you address the following questions and issues:

- How does your system load balance to keep all processors busy, while controlling inter-processor memory costs?
- How do two operators that share an edge synchronize the exchange of data? Do they use queues or some other data structure? We suggest providing pseudocode or finite state machines (FSMs) that illustrate this process.
- What data structure do you use to internally represent dataflow graphs?
- How are operators in your system scheduled? Do you use operating system threads? How many threads do you use? Do you have some way of determining which thread to run next, or to trigger the execution of a downstream operator when an upstream operator produces data?
- Does every operator run in its own thread, or do you have a pool of threads that run operators that have available input data? How do you map operators to threads?
- Does your design do anything special to exploit the fact that you are running on a NUMA architecture? For example, do you schedule adjacent operators in the dataflow on the same processor when possible?
- How do your ensure that your system performance scales as the number of operators increases?
- Do you treat stateless operators differently from stateful operators?
- How does your system deal with multi-input operators, as well as operators like Merge By Time in Figure 6?
- How does your system run on the workloads in Section IV?
- Are there particular workloads your design does not handle well?

It's okay, and sometimes desirable, to say, "no, my design doesn't do that" as long as you can identify what your design can and can't do, and explain why you made the trade-offs you did. Remember: a simple, correct design that meets the requirements laid out above is more important than a complex one that is hard to understand and does a flaky job. When in doubt, leave it out!

VI. Your Design Proposal

The design proposal should be a concise summary (at most 800 words) of your overall system design. It must be written as a memo, addressed to your boss (who you can assume has read the project description.) Your memo should start with a short, precise description before describing any details. It should also include a graphic or diagram describing your system.

You do not have to present a detailed rationale or analysis. However, if any of your design decisions is unusual (particularly creative, experimental, risky, or specifically warned against in the assignment), or you want to change the API substantially, it would be wise to describe such changes in your proposal.

You will receive feedback from both your TA and the Writing Program in time to adjust your final report.

VII. Your Design Report

The following are some tips on writing your design report.

Audience: You may assume that the reader has read the DP1 assignment but has not thought carefully about this particular design problem. Give enough detail that your project can be turned over successfully to an implementation team.

Rough outline:

1. Title Page: Give your design report a title that reflects the subject and scope of your project. Include your name, recitation time and section, and the date on the title page.
2. No Table of Contents.
3. Introduction: Summarize the salient aspects of your design, the trade-offs you made, a brief rationale for the design you have chosen, and the results from your analysis.
4. Design Description. Explain and elaborate your solution. Show how your solution satisfies the design constraints and solves the design problem (or how it fails to do so). Be sure to identify trade-offs you made, and justify them. You will want to sub-divide this section into subsections and use figures or code, where needed, to support your design choices.

   More specifically, in this section you should describe your general execution strategy for dataflow programs, including

   - What interface you chose for the `body()`function, and how it is called.
   - How you schedule operators (probably using multiple threads) and how you map threads to processors.
   - You should provide pseudocode for the `run()`function, taking care to show how upstream and downstream operators synchronize the exchange of data and how reads from input streams are scheduled.
   - If you treat different classes (e.g., stateless vs stateful) of operators differently, be sure to describe this clearly.

   If you have designed performance optimizations, you may wish to use a modular approach: start by explaining the unoptimized case before describing how optimizations fit in.
5. Analysis. Clearly describe your analysis and the conclusions you have drawn from your analysis. Be sure to describe how your system performs on the workloads given in Section IV.
6. Conclusion: Provide a short conclusion that provides recommendations for further actions and a list of issues that must be resolved before the design can be implemented.
7. Acknowledgments and References: Give credit to individuals whom you consulted in developing your design. Reference any sources cited in your report. The style of your references should be in the format described in the Mayfield Handbook's explanation of the IEEE style.
8. Word count. Please indicate the word count of your document at the end of the project.

VIII. How do we evaluate your report?

When evaluating your report, your recitation instructor will look at both content and writing and assign a letter grade. The writing program will separately grade the proposal and report for writing and assign a letter grade.

The most important aspect of your design is that we can understand how it works and that you have clearly answered the questions listed in Section V. Complicated designs that we cannot understand will not be graded favorably. A clear presentation with effective use of diagrams, pseudocode and / or FSMs is essential. The primary description of your design should be in the text of your document, with pseudocode used only to support this description, not in place of it. Be sure to break up any pseudocode into short logical components.

On the other hand, excessively simple designs are unlikely to perform well and will also be penalized. For example, simply running every operator in its own thread and placing a queue between operators, without discussing how your system achieves parallel speedups or exploits the NUMA nature of memory access would be too simplistic.

Some overall content considerations:

- Does your solution actually address the stated problem?
- How complex is your solution? Simple is better, yet sometimes simple will not do the job. On the other hand, unnecessary complexity is bad.
- Is your analysis correct and clear?
- Are your assumptions and decisions reasonable?

Some writing considerations:

- Is the report well-organized? Does it follow standard organizational conventions for technical reports? Are the grammar and language sound?
- Do you use diagrams and/or figures appropriately? Are diagrams or figures appropriately labeled, referenced, and discussed in the text?
- Does the report use the concepts, models, and terminology introduced in 6.033? If not, does it have a good reason for using different vocabulary?
- Does the report address the intended audience?
- Are references cited and used correctly?

IX. Collaboration

This project is an individual effort. You are welcome to discuss the problem and ideas for solutions with your friends, but if you include any of their ideas in your solution you should explicitly give them credit. You must be the sole author of your report.

X. Tasks and Due Dates

- Two copies of design proposal (800 words or fewer), due in Recitation 6.
- Two copies of detailed design report (2500 words or fewer), due in Recitation 13.

Please choose a font size and line spacing that will make it easy for us to write comments on your report (e.g., 11 pt font or larger; 1-inch margins.)