Phaser Beams: Integrating Stream Parallelism with Task Parallelism

Jun Shirako, David M. Peixotto, Dragoş-Dumitru Sbîrlea, Vivek Sarkar

Computer Science Department, Rice University, Houston, TX 77005

{shirako, dmp, dragos-dumitru.sbirlea, vsarkar}@rice.edu

Abstract

Current streaming languages place significant restrictions on the structure of parallelism that they support, and usually do not allow for dynamic task parallelism. In contrast, there are a number of task-parallel programming models that support dynamic parallelism but lack the ability to set up efficient streaming communications among dynamically varying sets of tasks. We address this gap by introducing *Phaser Beams* as a foundation for integrating stream parallelism with task parallelism. Phaser Beams builds on past work on *accumulators* and *point-to-point synchronization* in Habanero-Java (HJ) *phasers*, which in turn was derived from X10 clocks.

Phaser Beams introduce three key extensions relative to past work: 1) a *bounded phaser* that limits the maximum phase difference between a producer and a consumer synchronizing on that phaser and the buffer size needed to support streaming, 2) an extension to accumulators to work in *non-barrier mode* with bounded phasers for use in streaming, and 3) a *dynamic cycle-detection* algorithm to phaser registration to detect cyclic structures in a streaming program so as to enable efficient batching optimizations for acyclic structures. These extensions could easily be incorporated in a future version of X10.

Our preliminary Java-based implementation of Phaser Beams is restricted to the single node case, and the results obtained on three multicore SMPs are encouraging. As a calibration of the baseline performance of phaser synchronization, the performance of barriers in HJ phasers was found to be significantly faster than Java's CyclicBarrier and the Java and C++ implementations of X10's clocks, when measured using the BarrierBench benchmark. To evaluate the effectiveness of streaming with cyclic structures in Phaser Beams, we measured a performance improvement of up to 6.0× for the thread-ring microbenchmark, relative to the original Java version. For the restricted case of static parallelism and acyclic graphs, we compared the performance of our Phaser Beams implementation with the batched C-based StreamIt v2.1.1 implementation that only supports acyclic graphs. Our results on one microbenchmark (Push-Pop) and three StreamIt benchmarks (FilterBank, FMRadio, BeamFormer) show that the scalability of the Phaser Beams implementation relative to serial Java is better than that of StreamIt v2.1.1 relative to serial C, but the difference in absolute performance varied depending on the relative performance of the serial Java vs. serial C versions. To evaluate the effectiveness of Phaser Beams to integrate stream and dynamic task parallelism, we measured a performance improvement of up to 37.2× for the FacilityLocation benchmark compared with a pure task-parallel implementation using Intel's Concurrent Collections (CnC) language, and up to 40.2× speedup relative to sequential implementation for Sieve of Eratosthenes.

[Copyright notice will appear here once 'preprint' option is removed.]

1. Introduction

Streaming languages are attractive because they allow the programmers to specify parallelism declaratively by describing the computation units and flow of data among the units, as in languages such as StreamIt [11, 26]. However, current streaming languages place significant restrictions on the structure of parallelism that they support, and usually do not allow for dynamic task parallelism. In contrast, there are a number of task-parallel programming models (*e.g.,* Intel Threading Building Blocks [27], Java Concurrency [10], .Net Task Parallel Library [8], OpenMP 3.0 [22], Intel Concurrent Collections (CnC) [6] and X10 [5, 31]) that support dynamic parallelism but lack the ability to set up efficient streaming parallelism among dynamically varying sets of tasks. Extending a programming model to support multiple paradigms for parallelism takes careful thought and places challenges on language design, implementation, and usage; however, if successful, such extensions can have a big impact because they can enable the use of multiple paradigms in a single language environment. In that regard, stream parallelism and dynamic task parallelism are often regarded as separate paradigms, and their integration has not received much attention in past work. We address this gap by introducing *Phaser Beams* as a foundation for integrating stream parallelism with dynamic task parallelism. Phaser Beams builds on past work (summarized in Section 2) on X10 *clocks* [5, 23], *point-to-point synchronization* in Habanero-Java (HJ) *phasers* [24], and *phaser accumulators* [25].

The Phaser Beams primitives (described in Section 3) include three key extensions to past work. First, we introduce the notion of a *bounded phaser* that limits the maximum phase difference between a producer and a consumer synchronizing on that phaser and the buffer size needed to support streaming. Second, we extend accumulators to work in *non-barrier mode* with bounded phasers for use in streaming. Third, we add a *dynamic cycle-detection* algorithm to phaser registration that can be used to detect cyclic structures in a streaming program so as to enable efficient batching optimizations for acyclic structures. These extensions could easily be incorporated in a future version of X10. Section 4 shows how various streaming patterns can be implemented with these extensions in the context of the Habanero-Java (HJ) language [2, 14].

An implementation approach for Phaser Beams with extensions to HJ phasers and accumulators is discussed in Section 5. As in im-

plementations of streaming languages, we can improve the performance of Phaser Beams by using a batching optimization to reduce the synchronization overhead for stream access. This optimization is discussed in Section 6 and has a big impact on the performance results presented later. Although the batching optimization is effective in improving performance, it is not always applicable. If a stream is created with a feedback loop so that an output of one step feeds into a future input of that step, then an illegal batching optimization could cause the computation to deadlock. Section 7 describes a dynamic cycle-detection algorithm that enables Phaser Beams to fall back to a non-batched implementation when necessary.

We have built a Java-based implementation of Phaser Beams, and the results presented in Section 8 are encouraging. Performance results were obtained on a 16-core/16-thread Xeon SMP, a 8-core/64-thread UltraSPARC T2 SMP, and a 32-core/128-thread Power7 SMP to reduce architectural bias in our evaluation. To evaluate the effectiveness of streaming with cyclic structures in Phaser Beams, we measured a performance improvement of up to 6.0× (on UltraSPARC) for the thread-ring microbenchmark, relative to the original Java version. For the push-pop microbenchmark to evaluate synchronization and communication performance, the batched Java-based implementation of Phaser Beams performed up to 9.0× faster (on Power7) than the batched C-based StreamIt v2.1.1 implementation. We also compared the performance of Phaser Beams with StreamIt v2.1.1 for three benchmarks that do not contain dynamic parallelism nor cyclic structures — FilterBank, FMRadio, BeamFormer, but the comparison is complicated by the fact that the serial baseline for the C-based StreamIt version is (with one exception) significantly faster than the serial baseline for the Java-based Phaser Beams version. Specifically, the geometric mean speedup of StreamIt relative to its C-based serial baseline was 3.7×, 9.6× and 0.9× respectively on Xeon, UltraSPARC and Power7 SMP's. For the same machines, the geometric mean speedup of Phaser Beams relative to its Java-based serial baseline was 7.3×, 9.1× and 4.4× respectively. To evaluate the effectiveness of Phaser Beams to support dynamic task parallelism, we measured a performance improvement of up to 37.2× (on Xeon) for the FacilityLocation benchmark compared with a pure task-parallel implementation using Intel's Concurrent Collections (CnC) language [6], and up to 40.2× speedup (on T2) relative to sequential implementation for Sieve of Eratosthenes.

Finally, related work and conclusions are discussed in Sections 9 and 10 respectively.

2. Background

2.1 Async and Finish

We briefly recapitulate the *async* and *finish* constructs for task creation and termination, which were originally defined in X10 [5].

async: The statement, *async* ⟨*stmt*⟩, causes the parent task to create a new child task to execute ⟨*stmt*⟩. Execution of the async statement returns immediately i.e., the parent task can proceed immediately to its next statement.

finish: The statement, *finish* ⟨*stmt*⟩, causes the parent task to execute ⟨*stmt*⟩ and then wait till all sub-tasks created within ⟨*stmt*⟩ have terminated (including transitively spawned tasks). Each dynamic instance of a *finish* statement can be viewed as being bracketed by matching instances of *start-finish* and *end-finish* instructions. Operationally, each dynamic instruction has a unique *Immediately Enclosing Finish* (IEF) dynamic statement instance.

2.2 Phasers

We summarize the *phaser* construct in the Habanero-Java (HJ) language [24], which derives from the X10 clock construct [5, 23].

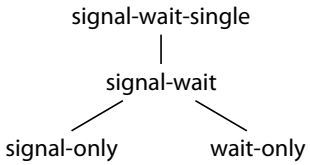


Figure 1. Capability lattice for phasers

Phasers integrate collective and point-to-point synchronization by giving each task the option of registering with a phaser in *signal-only* or *wait-only* mode for producer-consumer synchronization or *signal-wait* mode for barrier synchronization. In addition, a *next* statement for phasers can optionally include a *single* statement which is guaranteed to be executed exactly once during a phase transition [33]. These properties distinguish phasers from synchronization constructs in past work including barriers [13, 22] and X10 clocks [5, 21].

Phasers support the four kinds of operations listed below. At any point in time, a task can be registered in one of four modes with respect to a phaser: *signal-wait-single*, *signal-wait*, *signal-only*, or *wait-only*. The mode defines the capabilities of the task with respect to the phaser. There is a natural lattice ordering of the capabilities as shown in Figure 1. The phaser operations that can be performed by a task, T_i, are defined as follows.

new: When T_i performs a `new phaser(MODE)` operation, it results in the creation of a new phaser, ph, such that T_i is registered with ph according to `MODE`. If `MODE` is omitted, the default mode assumed is *signal-wait-single*. At this point, T_i is the only task registered on ph.

phased async: When task T_i creates an async child task T_j using the "`async phased` $(ph_1\langle mode_1\rangle, \ldots)T_j$" statement, T_i has the option of registering T_j with any subset of phaser capabilities possessed by T_i. This subset is enumerated in the list contained in the `phased` clause. We also support the "`async phased` T_j" syntax to indicate by default that T_i is transmitting all its capabilities on all phasers that it is registered with to T_j.

drop: When T_i executes an end-finish instruction for finish statement F, it completely de-registers from each phaser ph for which F is the IEF for ph's creation. In addition, T_i drops its registration on all phasers when it terminates.

next / signal / wait: The *next* operation has the effect of advancing each phaser on which the task is registered to its next phase, thereby synchronizing with all tasks registered on a common phaser. A *next* operation is equivalent to a *signal* operation followed by a *wait* operation. In the *signal* operation, the task signals all phasers that it is registered on with *signal* capability (*signal-only*, *signal-wait* or *signal-wait-single* mode). A phaser advances to its next phase after receiving all signals. In the *wait* operation, the task is blocked until all phasers that it is registered on with *wait* capability (*wait-only*, *signal-wait* or *signal-wait-single* mode) advance to their next phase. Additionally, phaser-specific operations such as `ph.signal()`, `ph.wait()`, and `ph.drop()` are also supported.

2.3 Accumulators

A *phaser accumulator* [25] is a construct that integrates with phasers to support reductions for dynamic parallelism in a phased (iterative) setting. By separating reduction operations into the parts of sending data, performing the computation itself, retrieving the result, and synchronizing among tasks, we enable asynchronous overlap of communication, computation and synchronization in a manner that extends the overlap in fuzzy [13] or split-phase [16] barriers. The accumulator operations are:

new: When task T_i performs a `new accumulator(ph,op,dataType)`

operation, it results in the creation of a new accumulator, a. `ph` is the host phaser with which the accumulator will be associated, `op` is the reduction operation that the accumulator will perform, such as `SUM`, `PROD`, `MIN`, `MAX` and `ANY`. `ANY` is the default if a reduction operation is not specified, and it selects an arbitrary element as the result for a given phase (among those provided to the accumulator in that phase). Also, `dataType` is the numerical type of the data upon which the accumulator operates.

put: An `a.put()` operation performed by task T_i sends a value for accumulation in accumulator `a` in the current phase. If a task performs multiple `put()` operations on the same accumulator in the same phase, they are treated as separate contributions to the reduction.

get: The `a.get()` operation performed by task T_i receives the accumulated value in accumulator `a` from the *previous* phase. Thus, the barrier synchronization provided by phasers provides an ideal foundation for reductions and there is no data race between `put()` and `get()` operations. Note that tasks that access an accumulator must be registered in *signal-wait* or *signal-wait-single* mode so as to participate in the barrier operation.

2.4 Comparison of X10 Clocks and Phasers

Although phasers are derived from X10 clocks and they share several important properties such as support of dynamic task parallelism and deadlock freedom with `next` operations, some key features distinguish phasers from X10 clocks. An important difference is the registration mode, which enables the creation of different synchronization patterns with point-to-point synchronizations. Until recently, a key difference in semantics lay in X10's `ClockUseException` at the end of a `finish` construct. However, recently the relationship between `finish` and `next` in X10 has been changed [32], and now both X10 clocks and HJ phasers support the IEF rule described earlier.

While past work on HJ phasers obeyed the *deadlock freedom* rule for X10's clocks, this rule does not hold for HJ if *explicit* `wait` operations are used instead of general `next` operations. However, as we will see, explicit `wait` operations play a critical role in streaming programs.

It is interesting future work to introduce these features of phasers in the context of X10 clock. Especially, supports of registration mode and accumulator should be useful extensions for the X10 language. The accumulator not only enables corrective operations such as reduction, but it also plays a fundamental role in supporting pipeline parallelism as described in this paper. The next operation containing single statements is a common synchronization pattern in parallel programming as in the OpenMP.

3. Phaser Extensions for Streaming with Dynamic Parallelism

In this section, we describe extensions to phasers [24] and accumulators [25] needed to support Phaser Beams.

3.1 Bounded Phasers

Figure 2 shows how a streaming computation might be implemented with standard phasers and accumulators. This implementation is unsatisfactory because it greatly constrains the parallelism. Since both the producer and consumer are required to be registered on the phaser in with *signal-wait* capability, the `next` works as a barrier operation and they will progress in lock-step. The producer will have to wait for the consumer at each step before it can work to produce the next item. This implementation obviously loses pipeline parallelism between the producer and consumer.

A first attempt to enable pipeline parallelism is to allow the producer to register on the phaser in *signal-only* mode and the consumer to register in *wait-only* mode. This change solves the parallelism problem, but still leaves open the buffering problem since the accumulator only stores the item from the previous phase. If the producer can create items faster than the consumer consumes them, then some of the items will be overwritten or lost.

```
finish {
    final phaser ph = new phaser(SIG_WAIT);
    final accumulator a = new accumulator(ph)
    async phased (ph<SIG_WAIT>) { //Producer
        while(moreWorkToDo()) {
            a.put(doSomeWork());
            next; // Barrier
        }
        a.put(doneWithWork());
        next; // Last barrier
    }
    async phased (ph<SIG_WAIT>) { //Consumer
        while(true) {
            next; // Barrier
            if(!doSomeOtherWork(a.get()))
                break;
}   }   }
```

Figure 2. Producer/consumer computation using phaser and accumulator without streaming extensions

To avoid overwriting previous items, we add an internal buffer to the accumulator that preserves results across multiple phases. However, we have an additional problem: the size of our internal buffer is unbounded because the producer can be an arbitrary number of phases ahead of the consumer. We can solve this problem by introducing the idea of a bounded phaser. A *bounded phaser* is a phaser that limits the maximum phase difference between a signaler and a waiter on the phaser. With a bounded phaser, it is now possible for a signal on the phaser to block the task performing the signal if the bound is reached.

A natural question that arises is how should we choose the bound size for the phaser? In some cases, the bound size is driven by the desired semantics of the program and in some cases it will be a performance tuning parameter. For example, consider implementing a bounded buffer using bounded phasers. We can do so without any extra synchronization to check the fullness/emptiness of the buffer by using a bounded phaser that has a bound corresponding to the buffer size. Here the bound is dictated by the desire to implement a buffer with a specific bound.

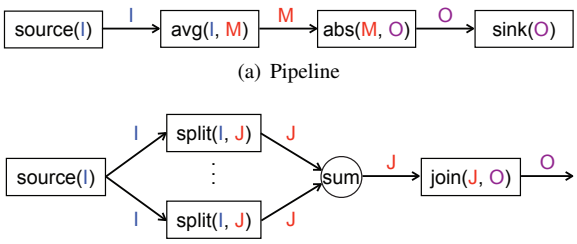
To summarize, we made three changes to phasers and accumulators to support a streaming model. First, we relaxed the restriction that accumulators only be used for phasers registered in `SIG_WAIT` mode. Second, we added an internal buffer to accumulators to store results from tasks in multiple phases. Third, we introduced a bounded phaser that limits the difference in phases between signalers and waiters on a phaser.

4. Expressing Streaming Patterns using Phaser Beams Primitives

In the Phaser Beams model, a stream is represented by a phaser and an accumulator. The accumulator stores the actual values that are in the stream, and the phaser is used to control when the values appear on the stream. In this section we describe how streams are created, used, and connected together to implement common streaming patterns.

Stream Creation

A stream is created by allocating a phaser/accumulator pair. The phaser is allocated with a *bound* that specifies the number of val-



(a) Pipeline

(b) Splitjoin

Figure 3. Common Streaming Graphs

ues that can be simultaneously live on the stream. The phaser must be allocated in SIG_WAIT mode by the parent task so that it can be passed to child tasks in both SIG mode and WAIT mode. The accumulator is allocated using the default ANY operation for accumulation. The ANY operation will select one of the values put to the accumulator in the current phase as the result of the accumulation. The example below shows how a new stream can be created.

```
int         bnd = 42;
phaser      ph = new phaser(SIG_WAIT, bnd);
accumulator s  = new accumulator(ph);
```

Stream Producers

Stream producers are tasks that are registered in SIG mode on the phaser controlling the stream. A value is written to the stream by calling the put() method on the accumulator and then executing a next operation to signal that a value is available. Note that with a bounded phaser, the next operation (signal) may block the producer, depending on how far its phase is relative to the consumers. An example of a simple stream producer is shown below.

```
async phased (ph<SIGNAL>) {
    while(...) {
        s.put(...); // write data to stream
        next;       // signal data is available
}   }
```

Stream Consumers

Stream consumers are tasks that are registered in WAIT mode on the phaser controlling the stream. A value is read from the stream by first performing a next operation (wait) and then calling the get() method on the accumulator to read the result from the previous phase. An example of a simple stream consumer is shown below.

```
async phased (ph<WAIT>) {
    while(...) {
        next;           // wait for data on stream
        v = s.get();    // read data from stream
}   }
```

We can also pass an integer offset to the get method to read results from earlier phases – analogous to a peek operation. The default get() is equal to get(0), and it returns the result of previous phase. The offset must be ≤ 0 and is constrained by the bound of the phaser. Passing an offset that is out of bounds will generate an exception. Reading results from previous phases is useful when implementing a task that reads multiple values from a stream.

4.1 Common Streaming Patterns

We now discuss how to map basic streaming constructs to Phaser Beams. These constructs are common to the streaming paradigm. We follow the terminology from StreamIt [29] for these constructs.

Filter

A filter is the most basic streaming construct. It processes elements from an input stream and writes them to an output stream. We represent filters as phased tasks that use accumulators for their input and output streams. The examples below all use filters as the building blocks for more complicated stream constructs.

Pipeline

A pipeline is a series of computations operating on the same stream of data. We can achieve a pipeline stream by registering each step in the pipeline as WAIT on its predecessor's phaser and SIG on its successor's phaser. In this example, we show a pipeline composed of two filters. The first filter takes two elements from the input stream, computes their average, sends them to the next filter which takes the absolute value and writes the result to the output stream. Figure 3(a) shows the streaming graph of this pipeline example.

```
void Pipeline() {
    phaser phI    = new phaser(SIG_WAIT, bnd);
    accumulator I = new accumulator(phI);
    phaser phM    = new phaser(SIG_WAIT, bnd);
    accumulator M = new accumulator(phM);
    phaser phO    = new phaser(SIG_WAIT, bnd);
    accumulator O = new accumulator(phO);

    async phased (phI<SIG>)            source(I);
    async phased (phI<WAIT>, phM<SIG>) avg(I,M);
    async phased (phM<WAIT>, phO<SIG>) abs(M,O);
    async phased (phO<WAIT>)           sink(O);
}
avg(I, M) { // e.g.
    while(...) {
        wait; wait;       // wait for two elements on I
        v1 = I.get(0);    // read first  element
        v2 = I.get(-1);   // read second element
        M.put((v1+v2)/2); // put result on M
        signal;
} }
```

Splitjoin

The splitjoin stream construct has a single source node that splits its input to N compute nodes. Each compute node feeds its output to a single join node that collects the results. We can implement this construct using Phaser Beams by allocating a phaser/accumulator pair for the input stream and another pair for the join stream. Each compute node registers as WAIT on the input phaser and SIG on the join phaser. A compute node will read the input from the shared input accumulator and put its output to the join accumulator. At a join node, it is a common process to accumulate its received values. The reduction operators available to accumulators support a natural way for such accumulations. In the next example, all N nodes put the value to an accumulator with a SUM reduction so that the join node receives the summed results (Figure 3(b)). When each compute node needs an individual input/join stream, we can replace the input/join accumulator by an array of accumulators to contain different data sequences for each node.

```
void Splitjoin() {
    phaser phI       = new phaser(SIG_WAIT, bnd);
    accumulator I    = new accumulator(phI);
    phaser phJ       = new phaser(SIG_WAIT, bnd);
    accumulator J    = new accumulator(SUM,phJ);
    phaser phO       = new phaser(SIG_WAIT, bnd);
    accumulator O    = new accumulator(phO);

    async phased (phI<SIG>)                 source(I);
    for(int s = 0; s < N; s++)
        async phased (phI<WAIT>, phJ<SIG>) split(I, J);
    async phased (phJ<WAIT>, phO<SIG>)     join(J, O);
```

```
}
source(I) {
    while(...) {
        I.put(...);
        signal;
} }
join(J, O) {
    while(...) {
        wait;
        O.put(J.get());
} }
```

```
split(I, J) {
    while(...) {
        wait;
        v = foo(I.get());
        J.put(v);
        signal;
} }
```

4.2 General streaming graphs with dynamic parallelism

In addition to the above typical streaming patterns, Phaser Beams support any shape of streaming graphs. Each node of a graph can create new nodes or terminate its execution dynamically so as to reshape the graph and adapt to the runtime fluctuation of input/output streams. Furthermore, streaming graphs and task parallel structures can be arbitrarily nested. A simple and useful extension is to support dynamic pipeline/splitjoin, whose pipeline-stages/compute-nodes are increased or decreased dynamically. As an example, we extend the splitjoin example to support dynamic parallelism as follows, where each compute node can spawn a new node and/or terminate its execution. In Section 8, we show actual implementations and experimental results for these dynamic configurations (FacilityLocation for dynamic splitjoin and Sieve of Eratosthenes for dynamic pipeline).

```
   ...
   async phased (phI<SIG>)             source(I);
   async phased (phI<WAIT>, phJ<SIG>) dynamicSplit(I, J);
   async phased (phJ<WAIT>, phO<SIG>) join(J, O);
}
dynamicSplit(I, J) {
   while(...) {
      if (spawnNewNode()) async phased dynamicSplit(I, J);
      if (terminate())    break;
      wait; ...
} }
```

5. Semantics and Implementation of Phaser Beams

As described in Section 3, an accumulator has an internal buffer to store results, and the phaser runtime manages the bounded phaser restriction. Implementation details for phasers and accumulators are described in existing literature [24, 25].

In phaser semantics, each task has two kinds of phase numbers: `sigP` and `waitP` to be increased by `signal` operation and `wait` operation, respectively, for each phaser object. `sigP` for all signaling tasks must be equal or larger than `waitP` for all waiting tasks, and `wait` operation is blocked when its `waitP` is equal to the smallest `sigP`. For bounded phaser implementation of bound size = `bnd`, we newly added the following constraint between `sigP` and `waitP`. `waitP` + `bnd` for all waiting tasks must be equal or larger than `sigP` for all signaling tasks, and `signal` operation is blocked when its `sigP` is equal to the smallest `waitP` + `bnd`. Both constraints are handled by a master task, which is the first waiting task to reach each barrier point [24].

For accumulation, each accumulator object can select one of two implementation policies, *eager* and *lazy* [25] as shown in Figure 4. In *eager* policy, all signaling tasks share an atomic buffer and `put` operation updates each element of the atomic buffer. In *lazy* policy, each signaling task has a local array and the master task performs accumulation sequentially within `wait` operation. The master task stores accumulated results to the result buffer, which contains `bnd` $\times 2$ results so that `get` operation can access

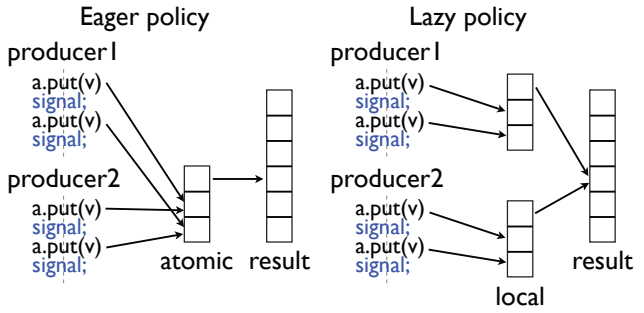


Figure 4. Implementation for eager and lazy policies

results with offset up to `bnd`. The optimal implementation depends on platforms and number of tasks.

6. Batching Optimization for Acyclic Structures

A standard optimization for streaming languages is to amortize the overhead of stream operations by batching the reads and writes to the stream. Instead of writing each individual item to the stream, the writes are batched into a local buffer. When the buffer is full or the task that owns the buffer is terminated, it is put to the stream. Similarly, a read from the stream reads a buffer of items instead of individual items. When all items are read from the buffer a new buffer will be obtained from the stream.

For Phaser Beams, the batching optimization provides performance improvement from two sources: reduced synchronization and reduced method invocation overhead. The reduced synchronization overhead in the `wait` operation is the largest source of improvement. With the batching optimization, a `wait` operation will be blocked to obtain a new buffer only when there are no more elements available in current buffer. Most of the `wait` operations will simply increment a local index variable pointing to the current element in the buffer. Section 6.1 provides details on how a `wait` is translated for the batching optimization.

6.1 Compile Time Transformation

For convenience, we initially implemented the transformation to support the batching optimization as a set of macros. In the future, we plan to implement this optimization as an intermediate representation transformation. Our transformation converts the code to use an adaptive policy that chooses (at runtime) between a batched and non-batched stream implementation based on whether or not the stream contains a cycle. We can also completely disable the adaptive switching by defining the macros to always execute the non-batched version since the non-batched version is the safe variant (we use that approach for the non-batched results presented in Section 8). We implement the transformation using the five macros described below.

`INIT_ACCUM` The initialization macro handles two tasks. First it waits until the cycle detection algorithm described in Section 7 has computed whether or not the given phaser is part of a cycle. Then it allocates the variables necessary to perform the batching optimization. The variables needed for batching are an array for the buffer and an index for indicating the current position in the buffer.

```
#define INIT_ACCUM(ph, a, type) {
  while (!ph.isDetectionDone()) ; //wait
  boolean isAcyclic_##ph = ph.isAcyclic();
  int   cnt_##ph = (ph.mode == SIG) ?
```

```
      0 : ph.batchSize - 1;
    type[] arr_##a  = (ph.mode == SIG) ?
      new type[ph.batchSize] : null;
  }
```

`SIGNAL` The signal macro checks to see if the phaser is acyclic. If not, then the phaser is signaled directly. If so, the batching optimization is implemented by checking to see if the buffer is full. If the buffer is full then it is put to the stream. If it is not full then the index is advanced so that `PUT` will write to the next available slot in the buffer. To support multiple accumulators on a phaser, the signal macro can contain more than one accumulator as parameters.

```
#define SIGNAL(ph, a, type)
if (isAcyclic_##ph) {
  if (cnt_##ph == ph.batchSize-1) {
    a.send(arr_##a);
    ph.signal();
    arr_##a = new type[ph.batchSize];
    cnt_##ph = 0;
  } else { cnt_##ph++; }
} else { ph.signal(); }
```

`WAIT` The implementation of the wait macro is similar to the signal macro. If the phaser is cyclic, then we perform a direct wait on the phaser. Otherwise, we check our counter to see if the buffer will be full. If so, we execute the wait on the phaser and retrieve the buffer so the `GET` will pull the value from the correct buffer. If the buffer is not full, we simply increment our counter that keeps track of the current size of the buffer. The wait macro can also contain multiple accumulators.

```
#define WAIT(ph, a)
if (isAcyclic_##ph) {
  if (cnt_##ph == ph.batchSize-1) {
    ph.doWait();
    arr_##a = a.resultArr();
    cnt_##ph = 0;
  } else { cnt_##ph++; }
} else { ph.doWait(); }
```

`PUT` The get macro is fairly simple. If the batching optimization is enabled, then the value is written to the buffer. Otherwise, the value is written directly to the accumulator.

```
#define SEND(ph, a, val)
if (isAcyclic_##ph) arr_##a[cnt_##ph] = val;
else a.send(val)
```

`GET` The get macro expands to an expression that checks whether we are performing the batching optimization or not. If batching is enabled, the result is read from the buffer retrieved from the latest `WAIT`, otherwise the value is read directly from the accumulator.

```
#define RESULT(ph, a)
isAcyclic_##ph ? arr_##a[cnt_##ph] : a.result()
```

6.2 Runtime Implementation

In the non-batched implementation, each accumulator contains a result buffer that stores the stream results. As described in Section 5, the size of the result buffer is derived from the phaser bound. Each element of the result buffer contains a single element from the stream. To support the batching optimization, we change the result buffer to contain not just a single element, but an entire batch array.

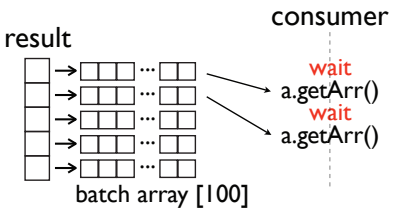


Figure 5. 2-D buffer for batch optimization

The size of the batch array is specified when the phaser is created, and is independent of the size chosen for the phaser bound.

Figure 5 shows the runtime structures needed to support batching. With bounded phasers and the batching optimization, the accumulator will have an internal data structure that looks like a two dimensional array. The first dimension is set according to the bound on the phaser controlling the accumulator. The second dimension is set according to the batch size for the batching optimization. So a declaration like

```
phaser ph     = new phaser(SIG_WAIT, 5, 100);
accumulator a = new accumulator(ANY, ph);
```

allocates an array of batch arrays that has 5 batch arrays each of length 100. Note that the accumulator also has a one dimensional internal result buffer of size 5 to support both batched and non-batched versions for adaptive switching. The `getArr()` method was added to accumulators that returns the batch array for the current phase.

The runtime was also extended to support new methods on phasers for controlling the batching optimization. The `isDetectionDone` method is used to contain the execution of a task until we can definitively say whether or not we have detected a cycle. The `isAcyclic` method is used to indicate whether or not the phaser is part of a cycle and thus whether it is safe to use the batching optimization. The method `isDetectionDone` returns true when either: the parent task terminates OR the parent task executes a `signal/wait` operation. The method `isAcyclic` returns true if both: the parent task does not detect a cycle with phaser registration AND the parent task does not execute a `signal/wait` before dropping registration on the phaser.

7. Cycle Detection

As we will see later in Section 8, the batching optimization described in the previous section can lead to significant performance improvements. However, it can only be applied to phasers that do not belong to a producer-consumer cycle. Some implementations of streaming languages restrict the data flow structure to be acyclic, thus ensuring that the batching optimization is always applicable. However, we decided against imposing this restriction since phaser registrations naturally lend themselves to creating both acyclic and cyclic structures. Instead, we implement an adaptive approach to the batching optimization by performing *cycle detection* at runtime.

For convenience, we call a phaser *cyclic* if it belongs to a producer-consumer cycle, and *acyclic* otherwise. In our approach, each phaser is initially assumed to be acyclic by default, thereby making it a candidate for the batching optimization. However, if a newly created registration causes the phaser to become cyclic, we change the implementation of the phaser to be non-batched. This detection of cycles at runtime builds on past algorithms for incremental computation of the transitive closure relation [17, 34]. Although we use past work for cycle detection, the following adaptive algorithm is ascribable to registration properties that are unique to

```
finish { // Parent task performs finish-enter
  P1 = new phaser(SIG_WAIT);
  P2 = new phaser(SIG_WAIT);
  async phased (P1<WAIT>, P2<SIG>) { // Child T1 ... }
  async phased (P2<WAIT>, P1<SIG>) { // Child T2 ... }
} // Parent task performs finish-exit
```

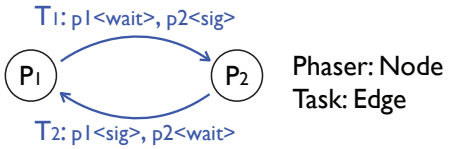


Figure 6. Example of child tasks creating a cycle

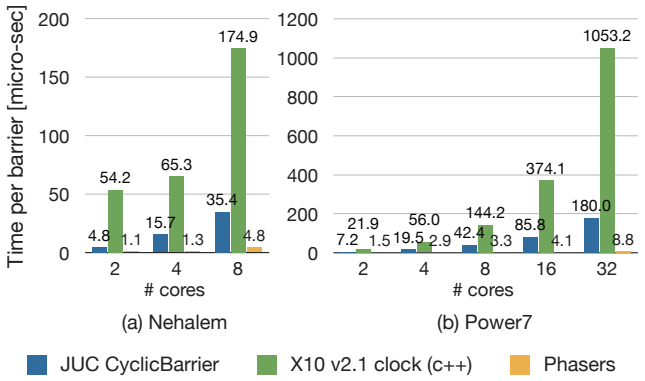


Figure 7. Barrier microbenchmark on Nehalem and Power7

phasers. We are unaware of any implementation of a streaming language that performs this cycle detection and adaptive batching on the fly while still ensuring that the overheads are minimal as in our approach.

7.1 Runtime Cycle Detection and Adaptation

In our approach, cycle detection is performed for all tasks and phasers belonging to the same *finish scope*. Note that finish scopes can be dynamic and nested in general, but each dynamically created task and phaser has a unique *immediately enclosing finish* (IEF) operation. Since the lifetime of a phaser is limited to the scope of its IEF construct, cycle detection can be performed independently for different finish scopes. The Phaser Beams approach of combining task parallelism and streaming parallelism allows for different sets of tasks to dynamically create their own streaming networks. We refer to the task performing the finish-enter and finish-exit operations for a finish scope as the *parent task* for the finish scope. In our implementation, the parent task assumes responsibility for tracking all phasers and phaser registrations that occur within the finish scope, and for using that information to detect cycles.

The graph maintained by the parent task includes a node for each phaser allocated in the finish scope and one or more edges for each child task that includes at least one signal and one wait restriction. Consider a new child task (async) A registered in `WAIT` mode for all phasers in set W and in `SIG` mode for all phasers in set S. (Note that S and W need not be disjoint, since a task can be registered on the same phaser in both `SIG` and `WAIT` modes.) Then, an edge is added with label A from node P_i to node P_j for each pair (P_i, P_j) such that $P_i \in W$ and $P_j \in S$.

Figure 6 contains a simple example of child tasks creating a cycle. In general, the parent task adds new edges to the graph for each child task (async) that it creates. Cycle detection is performed after each edge insertion, and each phaser that participates in a cycle is converted to a cyclic phaser. Insertion of edges for the parent's own phaser registrations are deferred until the parent task performs either an *end-finish* operation or its first `next/signal/wait` operation. In the former case, no new edges need to be added to the graph since all of the parent's phaser registrations are dropped upon reaching the end-finish, whereas edges for the parent's registrations will be added in the latter case. This is an important distinction because a common case is that the parent task is registered on all phasers in both `SIG` and `WAIT` modes, which would induce a self-loop on each phaser. If (as in Figure 6) the parent task proceeds directly to end-finish after creating the child tasks, then its phaser registrations won't impact the graph. As a result, the identification of each phaser as acyclic or cyclic can be made when the parent performs an end-finish or `next/signal/wait` operation. There is a possible optimization which we've deferred for future work *viz.,* when a child task terminates it may allow for the possibility of converting a phaser from cyclic to acyclic because it would result in removal of edges from the graph when its phaser registrations are dropped. It is worth noting that "acyclicity" will be a stable property in that case *i.e.,* when a phaser becomes acyclic in such a case, it cannot become cyclic again.

Our cycle detection scheme only needs to track phaser registrations for child tasks of the parent task performing the finish-enter and finish-exit operations. Though these child tasks can spawn additional tasks with phaser registrations, the *capability rule* [24] ensures that no new edges will be added to the graph.

Finally, as described in Section 6, it is necessary to generate dual-version code for each phaser operation so that the operation can be performed either in batched or non-batched mode depending on whether the phaser is detected to be acyclic or cyclic. The experimental results for Phaser Beams in Section 8 include the overhead of cycle detection and dual-version code. If needed, the overhead of dual-version code can be reduced by performing code specialization or dynamic code generation.

8. Experimental Results

In this section, we present experimental results for a HJ-based implementation of Phaser Beams. First, we briefly report the barrier microbenchmarking results of phasers, X10 clocks and Java's CyclicBarrier in order to demonstrate phaser's synchronization performance, on which the effectiveness of Phaser Beam implementations relies.

Figure 7 shows the barrier performance of JUC CyclicBarrier, C++ based X10 v2.1 clocks and phasers on an Intel Core i7 2.4GHz 2 quad-core processor (*Nehalem*) and IBM Power 7 3.55GHz 4 eight-core processor (*Power7*), SMT is turned off. The major performance advantage of phaser is due to "local spinning" technique introduced in the MCS algorithm [19], which gives significant synchronization efficiency compared with simple counting semaphore implementation used in CyclicBarrier and clocks.

8.1 Experimental Setup

We used the following benchmark programs for performance experiments.

1. **Push/pop micro-benchmark** is a simple producer-consumer microbenchmark to evaluate point-to-point communication and synchronization between two threads. The original code is from the `Minimal` benchmark, which is included in the StreamIt examples [26]. For benchmarking purposes, we removed the print statements, and ported it to our implementation of Phaser Beams with a batch macro. In the StreamIt version, the total iteration count N is given at runtime and both `IntSource` and

`IntSink` repeat N times. Likewise, the iteration count in the phaser version is given at runtime.

2. **Threadring micro-benchmark** is from the Computer Language Benchmarks Game [28]. It has threads linked in a ring and a token. Each thread waits for the token to be passed from previous thread, and then also passes to next thread. We ported the original Java code (threadring.java-3.java) to our implementation of Phaser Beams. This benchmark is used as a feedback example to evaluate the cycle detection mechanism.
3. **StreamIt benchmarks** we ported three streaming applications, `FilterBank`, `FMRadio` and `BeamFormer`, from the C-based StreamIt implementation [26] to our implementation of Phaser Beams.
4. The **facility location application** is a clustering application that solves the problem of optimum placement of production and supply facilities depending on an input stream of customer locations. This benchmark is used as a dynamic splitjoin example discussed in Section 4.2. Detailed descriptions are provided in Section 8.4.
5. The **Sieve of Eratosthenes** is an implementation of the standard algorithm that finds prime numbers using as input the stream of consecutive increasing integers. We provided two implementations of this benchmark, the dynamic splitjoin implementation similar to FacilityLocation and dynamic pipeline implementation to extend pipeline stages dynamically. A detailed description is available in Section 8.5.

For all Java runs, the main program was extended with a 30-iteration loop within the same process, and the best of the 30 times was reported in each case. This configuration was chosen to reduce the impact of JIT compilation time in the performance comparisons. Our HJ compiler and runtime are derived from IBM X10 version 1.5 [5], and we use the following runtime options: `-NUMBER_OF_LOCAL_PLACES=1 -INIT_THREADS_PER_PLACE=$nthreads`. Here, `nthreads` is the number of threads for which the measurement is being performed.

We present results for two different modes of the batching macro. The **batch** results are reported for the full macro expansion (as described in Section 6) to adaptively enable the batching optimization for acyclic streams. The **non-batch** results are reported for a macro version that expands directly to the non-batched version. In the non-batch version there is no overhead for cycle detection and runtime batching policy selection.

The batch size used for the batch optimization is 10000, which is same as the StreamIt runtime. The bound size used is 8 for the batched versions (i.e. 8 batches with 10000 entries in each batch), and 20000 for the non-batched version.

We use the StreamIt compiler and runtime 2.1.1 [26], with the compile options '`-O2`' for serial runs and '`-O2 --cluster $nthreads`' for parallel runs. This version of the StreamIt runtime always applies batching optimizations. It is based on the techniques described by Gordon *et al.* [11], but does not implement their later improvements [12]. The batch size in the StreamIt runtime is 10000, and the batched arrays are stored into a C++ standard library queue. This implementation does not support cyclic structures, so it always performs the batching optimization.

All results in this paper were obtained on three platforms. The first platform is a 16-thread (4 Quad-Core) Intel Xeon E7330 2.4 GHz SMP with 32 GB main memory running Fedora Linux release 8. For Java runs, we use the Java SE Runtime Environment (build 1.7.0-ea-b72) with Java HotSpot Server VM (build 17.0-b01, mixed mode). We use the gcc compiler version 4.1.2 (Red Hat 4.1.2-33) with '`-O3`' option to compile C++ source codes generated by the StreamIt compiler. The second is a 64-thread (8 cores $\times$ 8 threads/core) 1.2 GHz Sun UltraSPARC T2 (Niagara 2) with 32 GB main memory running Solaris 10. We conducted all Java runs in the Java 2 Runtime Environment (build 1.5.0_12-b04) with Java HotSpot Server VM (build 1.5.0_12-b04, mixed mode). The gcc compiler version 3.4.3 with '`-O3`' option is used as the binary code generator for StreamIt. The third platform is 128-thread (32-cores $\times$ 4 threads/core) 3.55 GHz IBM Power7 with 256 GB main memory running Red Hat Enterprise Linux release 5.4. We use the IBM J9 VM (build 2.4, JRE 1.6.0) for all Java runs, and the gcc compiler version 4.1.2 for StreamIt.

Benchmark version	Xeon	T2	Power7
StreamIt	114.0 $\times 10^6$	21.7 $\times 10^6$	33.1 $\times 10^6$
Phaser non-batch	11.0 $\times 10^6$	2.7 $\times 10^6$	8.4 $\times 10^6$
Phaser batch	148.2 $\times 10^6$	24.5 $\times 10^6$	299.4 $\times 10^6$

Table 1. Push/pop microbenchmark throughput (operations/second)

Benchmark version	Xeon	T2	Power7
Java (original code)	9.4 μs	16.3 μs	11.9 μs
StreamIt	Deadlock	Deadlock	Deadlock
Phaser non-batch	2.2 μs	2.7 μs	2.9 μs
Phaser batch	2.2 μs	2.7 μs	3.0 μs

Table 2. Threadring benchmark average time per hop

8.2 Performance for microbenchmarks

This section presents synchronization and communication performance using the push/pop and `threadring` microbenchmarks on a 16-core Intel Xeon SMP, an 8-core $\times$ 8-multithreading Sun UltraSPARC T2 SMP, and an 32-core $\times$ 4-multithreading IBM Power7 SMP, and discuss the performance breakdown of Phaser Beams. The number of threads in `threadring` is set to match the number of hardware threads, 16 for Xeon and 64 for T2 and 128 for Power7. The StreamIt runtime and Phaser Beams use the same batch size.

Table 1 shows the throughput (the number of push & pop operations per second) in the push/pop microbenchmark. Phaser Beams with batched optimization is 13.5$\times$, 9.1$\times$ and 35.6$\times$ faster than non-batched version and 1.3$\times$, 1.1$\times$ and 9.0$\times$ faster than StreamIt on the Xeon, UltraSPARC T2 and Power7 respectively. Confirming the lightweight nature of phaser synchronization, the throughput for batched version of Phaser Beams shows better performance than StreamIt on all platforms even though performance results for batched Phaser Beams include the overhead of cycle detection.

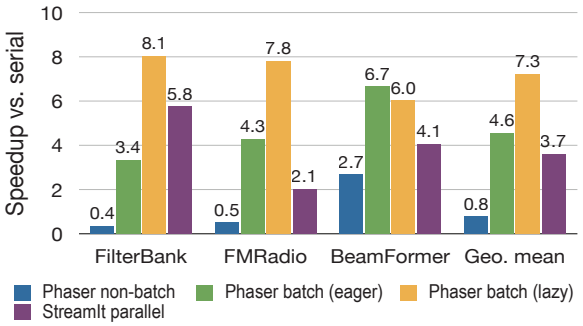
Table 2 shows the execution time per hop in `threadring` on the three platforms. The original code for `threadring` is written in Java. We used the default program size of `N` = 10^7, where `N` is the number of hops that will be executed. `Threadring` has a feedback cycle and each node in the ring needs to wait for the feedback signal (token) from the previous node. Therefore, the batch optimization must detect the cycle and disable batching so as to avoid deadlock. The batched version of Phaser Beams is 4.3$\times$ faster than Java version on Xeon, 6.0$\times$ faster on UltraSPARC T2 and 4.0$\times$ faster than on Power7, and shows almost the same performance as non-batched version despite cycle detection and adaptation overhead. The StreamIt v2.1.1 runtime resulted in deadlock because it doesn't support streaming programs with cycles.

8.3 Performance for Streaming Applications

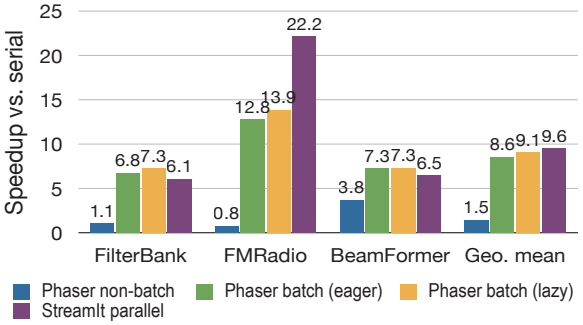
This section presents the performance of Phaser Beams and StreamIt using three streaming applications, `FilterBank`, `FMRadio` and `BeamFormer`, originally provided with the StreamIt benchmarks.

Benchmark	version	Xeon	T2	Power7
FilterBank (time in seconds)	Java serial	11.4 s	175.6 s	15.1 s
	Phaser (non-batch)	29.2 s	159.2 s	56.0 s
	Phaser (batch eager)	3.4 s	25.8 s	4.4 s
	Phaser (batch lazy)	1.4 s	23.9 s	3.4 s
	StreamIt serial	8.9 s	41.2 s	1.9 s
	StreamIt	1.5 s	6.7 s	5.4 s
FMRadio (time in seconds)	Java serial	25.3 s	288.1 s	26.6 s
	Phaser (non-batch)	48.9 s	349.8 s	97.3 s
	Phaser (batch eager)	5.9 s	22.4 s	5.8 s
	Phaser (batch lazy)	3.2 s	20.7 s	4.8 s
	StreamIt serial	7.6 s	470.3 s	5.9 s
	StreamIt	3.7 s	21.2 s	8.0 s
BeamFormer (time in seconds)	Java serial	19.1 s	258.7 s	20.7 s
	Phaser (non-batch)	7.1 s	68.9 s	13.0 s
	Phaser (batch eager)	2.9 s	35.4 s	5.7 s
	Phaser (batch lazy)	3.2 s	35.2 s	6.0 s
	StreamIt serial	6.4 s	86.8 s	8.9 s
	StreamIt	1.6 s	13.4 s	3.5 s
Geometric mean (speedup related to Java serial)	Phaser (non-batch)	0.8×	1.5×	0.5×
	Phaser (batch eager)	4.6×	8.6×	3.9×
	Phaser (batch lazy)	7.3×	9.1×	4.4×
	StreamIt serial	2.3×	2.0×	4.4×
	StreamIt	8.5×	19.0×	3.8×

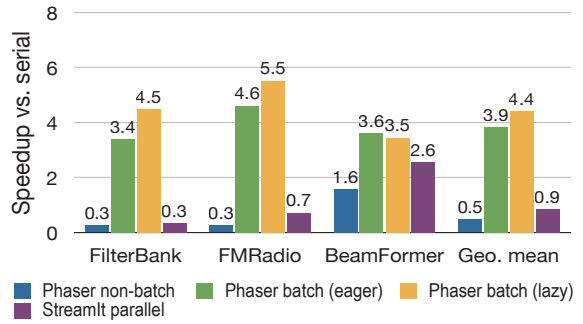
Table 3. Absolute performance for StreamIt Benchmarks



(a) Scalability on Xeon



(b) Scalability on UltraSPARC T2



(c) Scalability on Power7

Figure 8. Scalability related to each sequential base language

The StreamIt compiler divides an application into the given number of partitions, and each partition is executed by a thread. As the suitable number of threads, we selected 18-thread for `FilterBank`, 12-thread for `FMRadio` and 9-thread for `BeamFormer` according to the balance and granularity of streaming graphs. For example, `FilterBank` has one source filter, one sink filter, and one splitjoin containing eight parallel pipelines with two partitions of stages. Therefore, $1 + 1 + 8 \times 2 = 18$ should be the best for this application. The parallel HJ versions with Phaser Beams have the same number of threads and partitioning as the StreamIt versions. Static compiler optimizations of Phaser Beams related to partitioning is a subject for future work. We chose 20 million as stream size so as to extract sufficient streaming parallelism.

Table 3 shows the execution time and geometric mean speedup relative to serial Java versions on Xeon, UltraSPARC T2 and Power7. Since Phaser Beam uses Java and StreamIt uses C++, we also present relative speedup compared to the sequential version in each base language so as to evaluate parallel performance not skewed by the difference between base languages in Figure 8.

The geometric mean speedups in Table 3 show the batched version of Phaser Beams with *Lazy* accumulation policy obtained the best performance in Phaser Beams variants, and comparable performance to the parallel StreamIt on Xeon and Power7. However, the batched Phaser Beams with *Lazy* policy is 2.1× slower than parallel StreamIt on UltraSPARC T2, which is mainly due to the gap of sequential performance. The serial version of StreamIt is 2.0× to 4.4× faster than serial Java in geometric mean because of the base language difference, garbage collection in the JVM and sequential optimization by the StreamIt compiler. Regarding accumulation policy in Phaser Beams, *Lazy* is better than *Eager* except for `BeamFormer`. Dynamic selection of *eager* or *lazy* is a subject for future work. The batch optimization brings large performance improvement even though it includes runtime cycle detection.

Figure 8(a) and 8(c) show the batched version of Phaser Beams with *lazy* policy has better scalability than parallel StreamIt for all benchmarks on both platforms. As described in section 8.2, the synchronization performance of StreamIt is close to Phaser Beams on UltraSPARC T2. Therefore, the scalability of Phaser Beams and StreamIt are similar on UltraSPARC T2 (Figure 8(b)). Although the number of threads used in `FMRadio` is 10, the speedup of batched Phaser Beams and StreamIt is larger than 10 on UltraSPARC T2. We suspect this speedup is due to the data partitioning done by the parallel execution which allows the per thread data to fit within cache memory.

8.4 Dynamic Parallelism using CnC and Phaser Beams

This section shows initial experimental results for the integration of task and streaming parallelism using FacilityLocation. Formally [20], we are given a *metric space* and a *facility cost* for each node as well as a stream of *demand points*. The problem is finding the minimum number and positioning of nodes such that it minimizes a metric space expression dependent on which demand points are assigned to a node. This problem occurs in several fields such as strategic placement of production facilities, networking and communication, document classification. For example, consider the creation of a network, where servers have to be purchased and clients assigned to the servers in the order the come, as the come, by purchasing cables. Once the demand gets too high, new servers have to be purchased and at the same time the costs should be kept

Benchmark version	Xeon	T2	Power7
CnC	171.2 s	542.0 s	204.7 s
Phaser non-batch	7.0 s	35.0 s	8.9 s
Phaser batch	4.6 s	31.4 s	8.3 s

Table 4. Execution time of FacilityLocation

as close to the minimum as possible. A similar example is the webpage cluster problem: pages can have to be assigned to clusters according to some attributes. As the web grows rapidly, new pages have to be classified and server brought in. Both these situations are in fact streaming problems because of their dynamic nature in which new data arrives constantly (online) a working solution is expected at every point in time (incremental). As the problem is relevant in many fields, different formulations and approaches for solving it exist: two level [35], various hierarchical approaches [7], online and incremental [9]. There is an offline formulation of the FacilityLocation problem [4], The online, incremental (streaming) approaches of this problem do not find an optimal solution, but instead offer at any point in time a solution that is at most a constant time worse than the best one when points come in in random order; against an adversarial opponent, no online solution can be O(1) away from the optimum [20]. We have implemented the randomized algorithm in [20] for its simplicity.

Our implementation takes advantage of the dynamic parallelism feature of Phaser Beams in the sense that each place represents a cluster and the async corresponding to a place updates the metrics for points assigned to that cluster only. After assigning a *demand point* to an existing cluster or establishing a new facility (cluster center) to the new point, the application updates statistical information for the cluster. In our implementation, a `split` node handles the clustering and distributes *points* to *centers*, and a `ssq` node at each *center* processes the assigned *points* by updating the statistical information of the cluster (number of centers, sum of square distances to the center). Therefore, a new `ssq` node must be created when the split node opens a new *center*. This application relies on the dynamic task and streaming parallelism feature of Phaser Beams.

We also provided an alternative implementation using Intel Concurrent Collections (CnC) language [6] to compare with the Phaser Beams version. The execution time on the Xeon, UltraSPARC T2 and Power7 systems is reported in Table 4. The table clearly shows Phaser Beams versions obtain much better performance than the CnC version on all three platforms. Because the CnC doesn't support streaming feature, it cannot represent the order constraint of the input stream, which must be preserved to select and open *centers* correctly. Therefore, the CnC version initially launches all `split` tasks to process *points*, and the order constraint is forced as data dependence between the concurrent `split` tasks. The lack of streaming supports incurs not only unnecessary synchronization and task creation overhead, but also huge memory pressures for the CnC implementation, e.g., the garbage collection of CnC version on Xeon comprises 62.8% of the whole execution time. On the other hand, Phaser Beams spends less than 5% for GC on each platform, and the overall execution time is drastically small compared with the CnC version. The table also shows batch optimization brings further improvement.

8.5 Sieve of Eratosthenes with dynamic pipeline

This section describes the performance analysis of Phaser Beams that we performed using the Sieve of Eratosthenes application. The Sieve of Eratosthenes showcases the dynamic parallelism by finding all prime numbers up to N. If FacilityLocation models

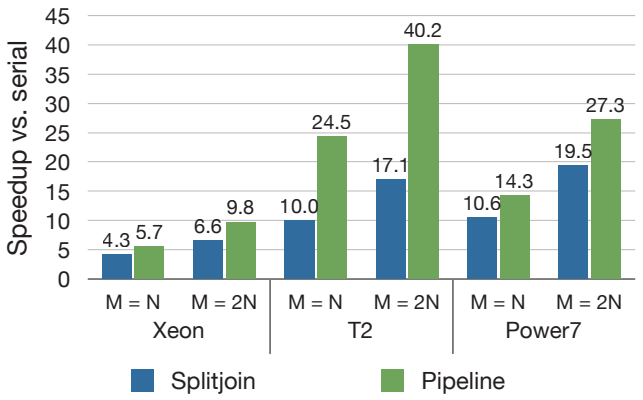


Figure 9. Speedup of Sieve benchmark on Xeon, UltraSPARC T2 and Power7

a dynamic split join pattern, the Sieve can benefit from either a pipeline or split-join implementation.

The first implementation we built is the dynamic pipeline. Each filter in the pipeline filters out the input numbers that are divisible by one of the prime numbers associated with the filter, and outputs indivisible numbers to the next filter. If a filter is the end of pipeline and an incoming number is indivisible, it means the number is prime. Such new prime numbers are associated with the last filter and stored in a shared vector to keep all prime numbers. When the number of primes associated with the current last filter reaches a certain maximum, a new filter is created and appended to the end of the pipeline to work as the new last filter. This implementation creates a structure such as the one in Figure 3a, but with dynamic number of filters. The communication between two neighbor filters is implemented using a phaser/accumulator pair.

We also implemented a split-join version. The parallel filters test for divisibility with the prime numbers associated with them just like in the pipeline version. Here, each filter produces 0 if the input number is indivisible or 1 otherwise. The results from all filters are reduced through a `SUM` operation and if the result is 0, the number is indeed a prime. The join node that tests the result also must inform the filters, so that one of them adds the new prime to its store of prime numbers (cyclic distribution in our implementation). This feedback loop is done through a phaser and accumulator pair, and each filter, once it checks for divisibility and sends the result, must wait for a signal and test to see if it should add the number to its prime store. Therefore, while each filter in the dynamic pipeline implementation performs one phaser wait call per round, it needs two wait calls in the split-join implementation and leads to a larger latency per filter. On the other hand, the complete latency of the pipeline version is larger, as numbers pass through a chain of filters.

In addition to the standard configuration to find prime numbers up to N, we provided an extension to show the speedup that can potentially be achieved with streaming phasers. The extended version finds all the prime numbers up to N , but also computes the number of integers from N to another constant M that are not divisible with any number up to N. We evaluated two classes of program sizes, (N = M = 500,000) for the standard configuration and (N = 500,000, M = 2 $\times$ N = 1,000,000) for the extended version. The maximum number of primes that a filter can accumulate is 3,000 on Xeon, 1,350 on Power7 and 680 on T2, which were selected to fit with the number of hardware threads on each platform. Here, SMT was turned off on Power7 in this experiment.

For the M = 2 $\times$ N configuration, the dynamic pipeline results in Figure 9 show very good speedup: 9.8$\times$ on 16-core Xeon, 40.2$\times$

on 64-thread T2 and 27.3× on 32-core on Power7. The split-join version gets speedups too, but poor compared to the pipeline implementation. The increased latency per filter leads to a decrease in throughput, that is not compensated by the exploitation of all cores immediately from the beginning of the program (the pipeline starts with a single filter and the parallelism increases as time goes on and filters are added).

Both implementations offer lower performance for low N. The pipeline Sieve implementation starts from a single filter and builds up from there, which limits the performance; speedup analysis is imprecise too because the parallelism is not constant during execution. The split-join implementation suffers from low granularity in the beginning, when few primes are assigned to each filter.

The results show that, with M = 2 × N, the speedup compared to M = N is doubled on most machines and in both implementations. This suggests that the synchronization overhead of our approach is small and the non-ideal scaling is caused by the application/implementation parallelism and not by the streaming system.

9. Related Work

Streaming languages have a long history. Here we survey a few past contributions.

Lucid is an early example of a streaming language that focuses on allowing formal proofs of correctness of the programs by having a mathematical notation that can be used with formal inference techniques [1]. The streaming nature comes from its use of recursive equations for specifying the computations.

StreamIt is a programming language for writing streaming programs [11, 29, 30]. Gordon *et al.* [12] discuss a compiler for the StreamIt programming language that targets the Raw architecture and attempts to exploit the various types of parallelism present in a StreamIt program: coarse-grained task, data, and pipeline parallelism. They show how their compiler can take advantage of each type of parallelism and include a suite of benchmark programs classified by they type(s) of parallelism it contains.

Karczmarke *et al.* describe a phased scheduling algorithm for the StreamIt language [15]. Their phased scheduling algorithm seeks to find a balance between code size and buffer size and works by executing the hierarchical units of the program as a set of phases. They also identify the deadlock problem with the feedback nodes and give an algorithm for calculating the minimum number of phases needed to schedule a feedback loop.

Brook adds data parallel operations to C and provides a compiler to execute in a streaming fashion on GPU hardware [3]. Liao *et al.* describe a transformation framework for data and computation transformations for Brook [18].

10. Conclusion

In this paper we introduced Phaser Beams for implementing stream parallelism in task-based parallel languages. The Phaser Beams primitives can be used directly at the level of imperative task-parallel languages, and we presented several coding patterns using Habanero-Java to demonstrate how to integrate task and stream parallelism by Phaser Beams.

Our implementation includes a batching optimization that greatly improves performance by reducing the synchronization overhead of stream access. Since we do not restrict the stream graphs that a programmer can write, we must provide a method for disabling the batching optimization for cyclic graphs. We describe our implementation of a dynamic cycle detection algorithm and show how it is used to adaptively switch between a batched and non-batched stream implementation. For the benchmarks studied, our implementation results in equal or better scaling than the C-based implementation of the StreamIt language despite additional overhead due to runtime cycle detection and adaptation.

We found that using phasers and accumulators to implement a streaming programming model provided some unexpected benefits. For example, we were able to use the semantics of phaser registration to detect cycles in the streaming graph and provide the means to avoid deadlock. Additionally, accumulators offer a natural way to perform reductions that would be written using a split-join in StreamIt. We gave an example in Section 4 on how to support a reduction split-join using accumulators, and we use this technique in the Phaser Beams implementation of the FMRadio and FilterBank benchmarks.

Opportunities for future research include extensions for dynamic selection of *eager* or *lazy* policy in accumulation, static compiler optimizations for streaming graph partitioning in the presence of dynamic parallelism, and support of Phaser Beams functionality in the X10 programming language.

References

[1] E. A. Ashcroft and W. W. Wadge. Lucid, a nonprocedural language with iteration. *Commun. ACM*, 20(7):519–526, 1977.

[2] Rajkishore Barik, Zoran Budimlic, Vincent Cavè, Sanjay Chatterjee, Yi Guo, David Peixotto, Raghavan Raman, Jun Shirako, Sagnak Tasirlar, Yonghong Yan, Yisheng Zhao, and Vivek Sarkar. The habanero multicore software research project. In *Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, OOPSLA '09, pages 735–736, New York, NY, USA, 2009. ACM.

[3] I. Buck et al. Brook for gpus: stream computing on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 777–786, New York, NY, USA, 2004. ACM.

[4] Moses Charikar, Chandra Chekuri, Tomás Feder, and Rajeev Motwani. Incremental clustering and dynamic information retrieval. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, STOC '97, pages 626–635, New York, NY, USA, 1997. ACM.

[5] P. Charles et al. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA, 2005. ACM.

[6] CnC. http://software.intel.com/en-us/articles/intel-concurrent-collections-for-cc, 2010.

[7] Güvenç Şahin and Haldun Süral. A review of hierarchical facility location models. *Comput. Oper. Res.*, 34:2310–2331, August 2007.

[8] DotNet. http://msdn.microsoft.com/en-us/library/dd460717.aspx.

[9] Dimitris Fotakis. Online and incremental algorithms for facility location. *SIGACT News*, 42:97–131, March 2011.

[10] B. Goetz. *Java Concurrency In Practice*. Addison-Wesley, 2007.

[11] M. I. Gordon et al. A stream compiler for communication-exposed architectures. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 291–303, New York, NY, USA, 2002. ACM.

[12] M. I. Gordon et al. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 151–162, New York, NY, USA, 2006. ACM.

[13] R. Gupta. The fuzzy barrier: a mechanism for high speed synchronization of processors. In *Proceedings of the third international conference on Architectural support for programming languages and operating systems*, pages 54–63, New York, USA, 1989. ACM.

[14] Habanero Java (HJ) Project. http://habanero.rice.edu/hj, 2009.

[15] M. Karczmarek et al. Phased scheduling of stream programs. In *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on*

Language, compiler, and tool for embedded systems, pages 103–112, New York, NY, USA, 2003. ACM.

[16] Arvind Krishnamurthy et al. Parallel programming in Split-C. In *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, pages 262 – 273, 1993.

[17] J. A. La Poutré and J. van Leeuwen. Maintenance of transitive closures and transitive reductions of graphs. In *Proceedings of the International Workshop WG '87 on Graph-theoretic concepts in computer science*, pages 106–120, New York, NY, USA, 1988. Springer-Verlag New York, Inc.

[18] S. Liao et al. Data and computation transformations for brook streaming applications on multiprocessors. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 196–207, Washington, DC, USA, 2006. IEEE Computer Society.

[19] John Mellor-Crummey and Michael Scott. Algorithms for Scalable Synchronization on Shared Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[20] A. Meyerson. Online facility location. In *Proceedings of the 42nd IEEE symposium on Foundations of Computer Science*, FOCS '01, pages 426–, Washington, DC, USA, 2001. IEEE Computer Society.

[21] Nalini et al. Compile-time analysis and specialization of clocks in concurrent programs. In *Proceedings of the 2009 International Conference on Compiler Construction (CC 2009)*, 2009.

[22] OpenMP Application Program Interface, version 3.0, May 2008. http://www.openmp.org/mp-documents/spec30.pdf.

[23] Vijay Saraswat and Radha Jagadeesan. Concurrent clustered programming. In *Proceedings of the International Conference on Concurrency Theory (CONCUR'05)*, August 2005.

[24] J. Shirako et al. Phasers: a unified deadlock-free construct for collective and point-to-point synchronization. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pages 277–288, New York, NY, USA, 2008. ACM.

[25] J. Shirako et al. Phaser accumulators: A new reduction construct for dynamic parallelism. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

[26] StreamIt. http://groups.csail.mit.edu/cag/streamit/, 2009.

[27] TBB. http://www.threadingbuildingblocks.org.

[28] The Computer Language Benchmarks Game. http://shootout.alioth.debian.org, 2009.

[29] W. Thies et al. Streamit: A language for streaming applications. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 179–196, London, UK, 2002. Springer-Verlag.

[30] W. Thies et al. Language and compiler design for streaming applications. pages 201–, 2004.

[31] X10 v2.1 language specification. http://dist.codehaus.org/x10/documentation/languagespec/x10-211.pdf.

[32] Clocks: New semantics for interaction between clocks and finish. http://jira.codehaus.org/browse/XTENLANG-806.

[33] K. Yelick et al. Productivity and performance using partitioned global address space languages. In *Proceedings of the international workshop on Parallel symbolic computation*, pages 24–32, New York, NY, USA, 2007. ACM.

[34] D. Yellin. Speeding up dynamic transitive closure for bounded degree graphs. *Acta Informatica*, 30:369–384, 1993.

[35] Jiawei Zhang. Approximating the two-level facility location problem via a quasi-greedy approach. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '04, pages 808–817, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics.