J Supercomput (2012) 61:118–140
DOI 10.1007/s11227-011-0656-7

StreamPI: a stream-parallel programming extension for object-oriented programming languages

Jingun Hong · Kirak Hong · Bernd Burgstaller · Johann Blieberger

Published online: 2 August 2011

Abstract Because multicore CPUs have become the standard with all major hardware manufacturers, it becomes increasingly important for programming languages to provide programming abstractions that can be mapped effectively onto parallel architectures. Stream processing is a programming paradigm where computations are expressed as independent actors that communicate via FIFO data-channels. The coarse-grained parallelism exposed in stream programs facilitates such an efficient mapping of actors onto the underlying multicore hardware.

We propose a stream-parallel programming abstraction that extends object-oriented languages with stream-programming facilities. StreamPI consists of a class hierarchy for actor-specification together with a language-independent runtime system that supports the execution of stream programs on multicore architectures. We show that the language-specific part of StreamPI, i.e., the class hierarchy, can be implemented as a library-level programming language extension. A library-level extension has the advantage that an existing programming language implementation need not be touched. Legacy-code can be mixed with a stream-parallel application, and the use of sequential legacy code with actors is supported. Unlike previous approaches, StreamPI allows dynamic creation and subsequent execution of stream programs. StreamPI actors are typed. Type-safety is achieved through type-checks at stream graph creation time.

We have implemented StreamPI's language-independent runtime system and language interfaces for Ada 2005 and C++ for Intel multicore architectures. We have evaluated StreamPI for up to 16 cores on a two CPU 8-core Intel Xeon X7560 server, and we provide a performance comparison with StreamIt (Gordon et al. in Interna-

J. Hong · K. Hong · B. Burgstaller (✉)
Yonsei University, Seoul, Korea
e-mail: bburg@cs.yonsei.ac.kr

J. Blieberger
Vienna University of Technology, Vienna, Austria

tional Conference on Architectural Support for Programming Languages and Operating Systems, 2006), which is the de facto standard for stream-parallel programming. Although our approach provides greater programming flexibility than StreamIt, the performance of StreamPI compares favorably to the static compilation model of StreamIt.

Keywords Programming language support for multicore architectures · Stream-parallel programming abstraction · Synchronous data-flow · Multicore architectures

1 Introduction

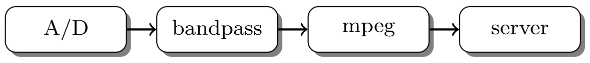
For the past three decades, improvements in semi-conductor fabrication and chip design produced steady increases in the speed at which uniprocessor architectures executed conventional sequential programs. This era is over, because power and thermal issues imposed by laws of physics inhibit further performance gains from uniprocessor architectures. To sustain Moore's Law and double the performance of computers every 18 months, chip designers are therefore shifting to multiple processing cores. The IBM Cell BE [20] processor provides nine processing cores, Microsoft's Xbox CPU [2] has three cores, and recent x86 systems from Intel and AMD already contain eight and 12 cores, respectively. According to a survey conducted by IDC [21], all PCs (desktops, mobile and servers) were predicted to be multicores by the end of 2010, with quad- and octo-cores together constituting more than 30% market share. For programming languages it becomes therefore increasingly important to provide programming abstractions that work efficiently on parallel architectures.

Many imperative and early object-oriented languages such as Fortran, C and C++ were designed for a single instruction stream. Extracting parallelism that is sufficiently coarse-grained for efficient multicore execution is then left to the compiler. However, sequential applications usually contain too many dependencies to make automated parallelization feasible within the static analysis capabilities of compilers. Ada, C# and Java provide thread-level concurrency already as part of the programming language. Thread-level concurrency allows the expression of task-parallelism (performing several distinct operations—tasks—at the same time), data-parallelism (performing the same task to different data items at the same time) and pipeline parallelism (task parallelism where tasks are carried out in a sequence, every task operating on a different instance of the problem) [33] already in the source code.

Because threads execute in a shared address space, it is the programmer's responsibility to synchronize access to data that is shared between threads. Thread-level concurrency plus synchronization through protected objects, monitors, mutexes, barriers or semaphores [17, 22] is commonly referred to as thread and lock-based programming. In addition to the difficulties of writing a correct multi-threaded program, thread and lock-based programming requires the programmer to handle the following issues.

1. Scalability: the number of cores is expected to double every 18 months, and applications should scale with the number of cores of the underlying hardware. Encoding a programming problem using a fixed set of threads limits scalability.

Fig. 1 Example stream program



2. Efficiency: over-use of locks serializes program execution, and the provision of lock-free data structures is difficult enough to be still considered a publishable result. Programs are likely to contain performance bugs:[1] cache coherence among cores is a frequent source of performance bugs with data that is shared between threads. False sharing [33] is a performance bug where data is inadvertently shared between cores through a common cache line.
3. Composability: composing lock-based software may introduce deadlocks and performance bugs.

It is therefore important to identify programming abstractions that avoid the above problems. The data-flow model is so common with parallel programs that it was selected as a distinguished parallel programming pattern [35]. Stream-parallel programs employ the data-flow model; they consist of a set of independent actors that communicate via FIFO data streams. Actors read from their input channels, perform computations and write data on their output channels. Each actor represents an independent thread of execution that encapsulates its own state. Actors are self-contained, without references to global variables or to the state information of other actors. The self-containedness of actors rules out any dependencies except those implied by communication channels: an actor fires if sufficient data is available on its input channels and if the output channels provide enough space to accommodate the data produced by the actor. Because of this absence of dependencies, stream programs provide a vast amount of parallelism, which makes them well-suited to run on multicore architectures.

Figure 1 depicts an example stream program that consists of an A/D converter, a bandpass filter, an mpeg-encoder and a network server that provides an mpeg data-streaming service. The application domain for stream parallelism includes networks, voice, video, audio and multimedia programs. In embedded systems, applications for hand-held computers, smart-phones and digital signal processors operate on streams of voice and video data.

Despite its large application domain, stream-parallelism is not well-matched by general-purpose programming languages; mainly because actors and streams are not provided at the language level. As a consequence, programmers need to devise their own abstractions, which are then prone to lack readability, robustness, portability and performance. A programming language implementation such as a compiler and a runtime system that is not aware of stream parallelism most likely will not be able to take advantage of the abundance of parallelism provided by stream programs.

A domain-specific programming language for stream-programming on the other hand will provide efficient abstractions for stream programming, but generally lack user acceptance. We argue that a middle-ground in the form of a library-level programming extension can combine the superior compiler support and high user-acceptance of contemporary programming languages with novel programming abstractions offered by domain-specific stream programming languages. Because actors

[1] Bugs that prevent an otherwise correct program from executing *efficiently*.

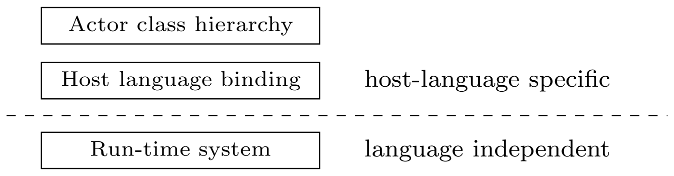


Fig. 2 StreamPI architecture overview

and stream graph abstractions fit nicely with encapsulation and inheritance provided by the object-oriented programming paradigm, we advocate that such a library-level extension should be class-based rather than procedural.

Figure 2 shows the overview of the proposed architecture. A class hierarchy containing stream-graph primitives such as actors is embedded into an object-oriented host programming language. Programmers extend actors to create user-specific actor behavior. The language-independent runtime system services are provided through a thin binding in the host language. The runtime system provides the functionality to connect actors into stream-graphs and to execute stream-graphs on a given multicore architecture. The purpose of the language-independent runtime system is to facilitate stream-parallel programming abstractions across different programming languages. At the moment we do not support multi-language stream programs. The contributions of this paper are as follows.

– We present StreamPI, a stream-parallel programming interface for object-oriented programming languages. Our programming interface lifts the abstraction level for the development of stream programs. Actors are represented as objects that are conveniently connected into stream graphs.
– We employed Ada 2005 and C++ as StreamPI host languages to show that the StreamPI abstractions fit well with contemporary object-oriented languages.
– We provide design and implementation of a language-independent runtime system that supports the execution of stream programs. Our runtime system manages the data channels between actors, load-balances and schedules actors among the execution units of a multicore architecture, and provides the complete stream program execution infrastructure.
– Unlike previous approaches, StreamPI allows dynamic creation of stream graphs. Instead of applying static profiling heuristics [13] or off-line profiling [11, 29, 48, 49], we perform dynamic profiling of stream programs to load-balance actors among the execution units of a multicore architecture.
– We provide experimental results that show the validity of our approach.

The remainder of this paper is organized as follows: in Sect. 2 we provide background information and survey related work. In Sect. 3 we introduce the stream-parallel programming interface for object-oriented programming languages. In Sect. 4 we describe the design and implementation of the language-independent runtime system required to support applications that use our programming abstractions for stream parallelism. Section 5 contains our evaluation of StreamPI on the Intel x86-64 architecture. We draw conclusions and outline future work in Sect. 6.

2 Background and related work

Stream programming has turned out to be an effective programming approach with multicore architectures. A survey on programming languages that include a concept of streams can be found in [42]. With synchronous data-flow languages (SDF, [31]), the number of tokens consumed and produced by an actor is already fixed at compile-time.

Lustre [9] is an SDF programming language used for safety-related software in aircrafts, helicopters and nuclear power plants. Lustre supports only a limited number of data types and control statements. Esterel [5] improves on the number of control statements and is well-suited for control-dominated synchronous systems. Both languages require a fixed number of inputs to arrive at the same time before a stream node executes.

StreamIt [47] uses syntax similar to Java to specify the computations of stream program actors. A StreamIt programmer constructs a stream graph consisting of filters which are connected by a fixed number of constructs: pipelines, split-joins and feedback loops. Two types of splitters are supported: Duplicate and RoundRobin. A feedback loop allows one to create cycles in the stream graph. In contrast to its predecessors, StreamIt supports a dynamic messaging system for passing irregular, low-volume control information between filters and streams. StreamIt employs the SDF model except for the following differences: StreamIt has a non-consuming read (`peek`) operation similar to the computation model introduced in [28], and stream graphs in StreamIt are "structured", i.e., they are restricted to composites of filters, pipelines, split-joins, and feedback loops.

StreamIt has become the de facto standard for research in stream programming language implementations. Stream program orchestration denotes the mapping of a stream program onto a parallel architecture. Although stream programs contain an abundance of parallelism, stream program orchestration remains to be a challenging problem, which is reflected by the significant amount of recent research effort on this topic [8, 10, 11, 13, 14, 25, 27, 29, 45, 48–51].

The StreamIt compiler [13, 25] targets the Raw Microprocessor [36], shared-memory multicore architectures and clusters of workstations. It applies heuristic stream graph transformations such as actor fusion and fission to increase the computation to communication ratio of stream programs. To load-balance actors among processors, a greedy partitioning heuristic is applied. This heuristic minimizes the makespan, i.e., the time duration of the longest-running processor. Kudlur and Mahlke's stream graph modulo scheduling [29] employs an integer linear programming (ILP) formulation to distribute StreamIt actors among the synergistic processing elements of the Cell processor [18]. The ILP formulation consists of an integrated unfolding and partitioning step that spreads data-parallel actors and maximally packs actors onto cores. The Flextream approach by Hormati et al. [19] applies an ILP formulation in combination with dynamic adaptation techniques that modify the static solution according to resource availability. Udupa et al. [48] applied an ILP formulation for the orchestration of StreamIt programs on GPGPUs. The optimal execution configuration of a stream program in terms of the number of registers per thread and the number of data-parallel actor instances is determined through profiling. A buffer layout technique optimized for GPGPU memory architectures is presented. Orchestrating the execution of stream programs on heterogeneous platforms consisting of a multicore CPU and a GPGPU accelerators has been examined in [49]. Udupa et al. formulated a communication-aware ILP problem for partitioning computations between CPU cores and GPGPU streaming multiprocessors (SMs). The ILP problem is approximated by a heuristic algorithm, with solutions on average within 9.05% of the optimal solution across the benchmark suite. Wei et al. extend the Brook stream programming environment with an ILP formulation to allocate actors on the Cell processor s.t. communication costs are minimized.

It has been reported in [11] that state-of-the art ILP approaches for stream graph orchestration are intractable or at least impractical for larger stream graphs and a larger number of processors. This becomes an issue with the growing stream program sizes reported in a recent survey [46]. As a result, the approach in [11] proposes a 2-approximation algorithm for deploying stream graphs on multicore computers. A stream graph transformation for bottleneck elimination based on hot regions is introduced. The approach applies a data rate transfer model that optimizes the arrival rate of stream programs rather than the makespan.

Contrary to the above StreamIt-based approaches, this paper proposes a library-level stream-programming extension that allows dynamic stream graph creation, i.e., stream-graph orchestration is conducted dynamically and not inside the compiler. This precludes the use of ILP formulations, due to the large runtime overhead of ILP solvers.

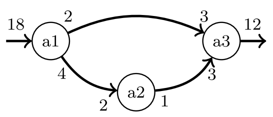
StreamFlex [41] constitutes a data-flow programming model targeting real-time stream-processing with Java. StreamFlex offers a zero-copy guarantee for streamed objects, and applies software transactional memory for communication with non-real-time tasks. The focus of StreamFlex is on real-time systems, esp. on predictability. No provision for stream program orchestration on multicore architectures is made.

SPEX [34] is a parameterized data-flow language extension for C++ that allows programmers to create streaming computation and communication patterns of DSP systems. Data-flow is described with a set of parameters. The static compiler iterates through all combinations of parameters to produce individual SDF graphs which are then scheduled using existing data-flow scheduling techniques. Based on actual parameter values, prescheduled SDF graphs are selected and executed at run-time.

The stream programming paradigm is also applied in Google's systems programming language "Go" [12]. Go provides co-routines that communicate via channels.

We do not consider specification languages like SDL [4] here, because in this paper we are interested more in implementing systems than in designing systems. Special purpose hardware for stream program execution [24], mapping stream programs onto FPGAs [16] and programming support for media processors [52] differ from our approach which targets general-purpose shared-memory multicore architectures.

Our approach is based on SDF and thus differs from Kahn process networks [23] which allow data-dependent communication. Leung et al. show in [32] how Kahn process networks can be mapped onto parallel architectures using MPI for communication. Carpenter et al. [8] present an iterative heuristic partitioning and allocation algorithm that maps Kahn process networks with optional SDF-parts onto heterogeneous multiprocessors.



Actor	#Steady-State Iterations
a1	3
a2	6
a3	2

Fig. 3 Example: SDF graph and minimal steady-state schedule

Summing up, our approach for StreamPI goes beyond that of StreamIt, because we allow dynamic creation of stream graphs. In contrast to all other stream program orchestration approaches, the whole spectrum of data types available in a host language such as C++ or Ada can be used for streaming. Unlike StreamIt, we do not restrict the user to predefined stream graph constructs like pipelines, split-joins and feedback loops. The filters, splitters and joiners provided by StreamPI are sufficient to generate structured stream-graphs. For example, Fig. 4(d) shows how a feedback loop can be constructed with StreamPI. As explained in [45], it is yet not entirely clear whether structured stream graphs are sufficient for all possible applications. Our plan with StreamPI is to survey the stream graph patterns arising from real-world applications and build higher-level stream graph constructs from commonly occurring patterns.

2.1 SDF semantics

Stream programs expose an abundant amount of explicit parallelism already in the source code. Actors (i.e., stream graph nodes) constitute independent units of execution that interact only through data channels. Actors may be stateless or encapsulate state. Despite this vast amount of parallelism, it is still a challenging task to schedule a stream program on a parallel architecture. The obvious solution of assigning an independent thread of control (e.g., a Pthread) to each filter and to model communication via producer-consumer style bounded buffers induces too much context-switch and synchronization overhead for all but the largest filters. In fact filters often contain only a small amount of computation, which makes it hard to maintain a high computation-to-communication ratio with stream programs. StreamIt and StreamPI apply SDF semantics, which requires the amount of data consumed and produced by an actor to be known a priori. Figure 3 depicts an SDF example stream graph. The numbers associated with each input and output of an actor denote the number of data items consumed and produced during one actor execution. For example, Actor a2 consumes two data items and produces one data item per execution.

Conceptually, an SDF graph repeatedly applies an algorithm to an infinite data stream. An SDF graph is executing in steady-state if the amount of data buffered between actors remains constant across repeated executions. The table in Fig. 3 depicts the number of iterations required for each actor such that the above SDF graph stays in steady state. E.g., Actor a1 has to be executed three times, resulting in 3×2 data items on channel a1$\rightarrow$a3. Actor a3 will consume those 6 data items during its two executions. Computing the steady state for SDF graphs has been studied in [3, 30].[2] StreamIt uses a variant of this algorithm for structured SDF graphs [26]. An SDF

[2]Note that a steady state for a given SDF graph need not exist in general.

graph is scheduled based on its steady state. The scheduler consists of two phases, one bootup-phase to bring the system into steady state, and the steady-state schedule itself.

3 A class-based stream-parallel programming abstraction

To add a stream programming abstraction to an object-oriented host programming language, the following three approaches are conceivable: (1) extend the core language itself through language extensions, (2) provide compiler extensions for streaming constructs, or (3) provide a programming library that the user can link with application code.

StreamPI is strictly a library. Although language extensions are attractive, they create a high barrier to adoption, especially in commercial settings. A library-based extension allows re-use of legacy code, opens up a migration path and does not require programmers to step out of their accustomed programming environment. Moreover, a library lowers the entry barrier for language researchers and enthusiasts who want to work in this area themselves. Contemporary object-oriented programming languages like Ada 2005, C++, C# and Java provide excellent support for packages, types and generic programming, which facilitates library design and implementation. Libraries have been successfully applied to extend programming languages, as demonstrated by the POSIX threads library [7], the message-passing interface (MPI, [37]), and the Intel Thread Building Blocks [38].

Figure 4 shows the three actor programming primitives that StreamPI provides: filters, splitters and joiners. Together, these primitives are sufficient to generate arbitrary stream graph structures. E.g., Fig. 4(d) shows how a loop can be constructed from a joiner, a filter and a splitter.

We chose the class hierarchy depicted in Fig. 5 to represent stream program actors. The abstract `Base_Filter` class at the root of this hierarchy encapsulates the commonalities among actors. Classes `Filter`, `Splitter` and `Joiner` are generic

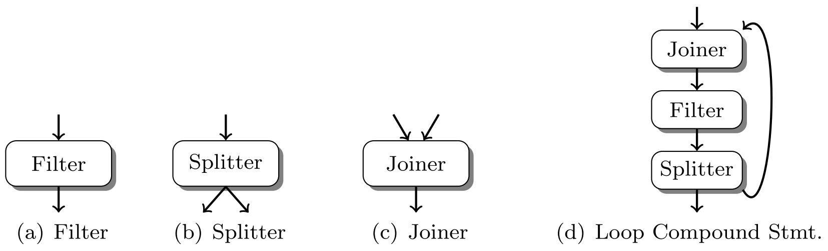
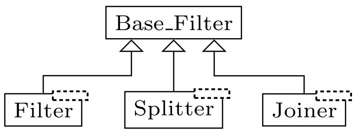


Fig. 4 Three StreamPI stream-graph primitives and one compound construct

Fig. 5 Type hierarchy for StreamPI actors



```
1   package Base_Filter is

2      type Base_ Filter is abstract tagged private;
3      type Base_ Filter_Ptr is access all Base_ Filter'CLASS;

4      procedure Work (f: access Base_Filter) is abstract;

5      procedure Connect(f: access Base_Filter;
6                        b: access Base_Filter'CLASS;
7                        out_weight: positive := 1;
8                        in_weight: positive := 1) is abstract;

9      function Get_In_Type(f: access Base_Filter)
10      return Root_Data_Type.Root_Data_Type'CLASS is abstract ;

11      procedure Set_In_Weight (f: access Base_Filter; in_weight : positive) is abstract;

12  private
13      type Base_ Filter is abstract tagged null record;
14  end Base_Filter;
```

Fig. 6 StreamPI Base_Filter class for Ada 2005

classes (indicated by ⬚) derived from `Base_Filter`. Derived classes are parameterized with type information to specify the type of the data consumed/produced by an actor. Filters are parameterized by input type and output type. Distinguishing input type and output type allows a `Filter` object to type-convert data, which in turn enables the generation of heterogeneous stream graphs. Splitters and joiners are restricted to a single type, i.e., their input and output type is the same. It should be noted that this does not restrict the expressiveness, because type-converting filters can always be inserted before/after a splitter or joiner.

The `Base_Filter` for Ada 2005 is depicted in Fig. 6. It provides an abstract primitive operation[3] named `Work` (line 4). For class `Filter` and classes derived from `Filter`, this work function is overridden by the programmer to encode the actor's actual computation. With splitters, the purpose of the work function is to split the data-elements from the incoming data-stream among the outgoing streams. Joiners merge the data from incoming streams onto a single output-stream (see Fig. 4).

Every actor provides a `Connect` method (lines 5–8 in Fig. 4) to attach its stream graph successor(s). The arguments to the `Connect` method are the downstream successor (line 6) and the number of output data items (line 7) of this actor plus the number of input data items (line 8) of the downstream successor. For example, to connect actors X and Y via edge $X \xrightarrow{1\quad 2} Y$, method `X.Connect(Y,1,2)` would be used by the StreamPI library user. In the case of multiple successors (i.e., with splitters), the `Connect` operation must be invoked for each successor. The successor's `Set_In_Weight` operation is invoked from within `Connect` to communicate the `in_weight` argument value to the successor. `out_weight` and `in_weight` of stream-graph edges are used to compute the steady-state schedule as outlined in Sect. 2.

When two actors are connected at run-time, their input and output types are used to ensure type compatibility, i.e., the output type of an upstream actor must match

[3] The equivalent of a C++ pure virtual function.

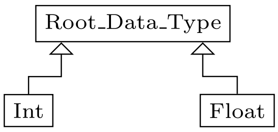


Fig. 7 Root data-type hierarchy for OO-languages that lack a single root superclass

```
generic
    type In_ Type is new Root_Data_Type.Root_Data_Type with private;
    type Out_ Type is new Root_Data_Type.Root_Data_Type with private;
package Filter is
    type Filter is abstract new Base_Filter.Base_Filter with private;
    procedure Work(F: access Filter) is abstract;
    procedure Push(F: access Filter; Item: Out_Type);
    function Pop(F: access Filter) return In_Type;
    ...
private
    type Filter is abstract new Base_Filter.Base_Filter with record
        In_Var : aliased In_Type;
        Out_Var : aliased Out_Type;
    end record;
end Filter;
```

Fig. 8 Generic package to embed the StreamPI filter class in Ada 2005

the input type of the adjacent downstream actor(s). We use method `Get_In_Type` (lines 9 and 10 in Fig. 6) to retrieve the input type of the downstream actor. Unlike C# and Java, C++ and Ada do no provide a single root superclass `Object` from which all other classes are derived. Due to this lack of a single root superclass, we employ a user-extensible data-type class hierarchy with C++ and Ada. This hierarchy is depicted in Fig. 7. By subtype polymorphism, the root of the class hierarchy (`Root_ Data_Type`) is used as the return-type of method `Get_In_Type` in Fig. 6. The calling upstream actor of method `Get_In_Type` then attempts a dynamic downcast to its own output type to ensure type-equivalence. In Ada 2005, this is achieved by directly comparing tags of the types ([22, 3.9(22)]), in C++ we employ a `dynamic_cast<>()`.

If the data types differ, exception `Stream_Type_Error` is raised. This exception is provided by the StreamPI runtime system, as explained in Sect. 4. Hence we combine a type secure approach with dynamic creation of stream graphs. Users may define arbitrary types by extending the data-type class hierarchy from Fig. 7. Filters, splitters, and joiners (see Fig. 4) specific to a chosen root data type can be instantiated.

The generic Ada 2005 package for filters is depicted in Fig. 8. StreamPI filters provide a `Pop` method to retrieve a single data item from a filter's input stream. Likewise, `Push` allows a filter to write a data item onto the output stream. Methods `Push` and `Pop` are to be used within a filter's `Work` method. As already mentioned, by overriding the `Work` method of a filter, the user implements the actual behavior of the filter. The `Work`-methods of splitters and joiners are provided by our implementation: splitters partition the incoming data stream into sub-streams, joiners merge several incoming data streams of the same type into a single stream. The design and implementation for C++ follows along the same lines, the C++ template for filters is depicted in Fig. 9.

Figure 10 shows a stream-parallel version of the Mergesort algorithm for $N = 8$ data items. During each steady-state execution (aka iteration) of this stream program,

```
template <class In_Type, class Out_Type = In_Type>
class Filter : public Base_Filter {
public:

...

virtual void Work (void) = 0;

void Push (Out_Type item);

In_Type Pop(void);

void Connect (Base_Filter * C, int Out_Weight=1,
               int In_Weight=1);
...
};
```

Fig. 9 Template to embed the StreamPI filter class in C++

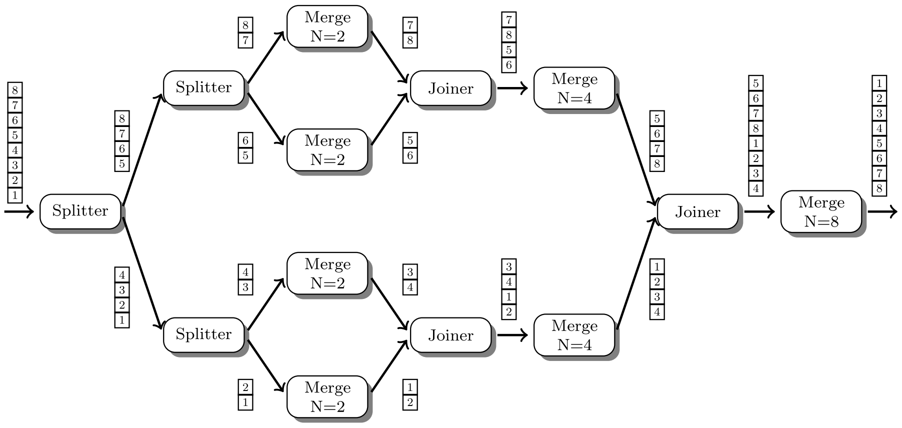


Fig. 10 Mergesort for $N = 8$ data items

8 data items are popped from the input stream, sorted, and pushed onto the output stream. We chose this example because it showcases the dynamic creation of stream-graphs depending on user input data (parameter N). The Mergesort example is implemented as follows (for space considerations, the full implementation is given in the Appendix).

1. First the stream data type is declared by extending the `Root_Data_Type`. In our case it is an integer type (see Fig. 11). The implementation of the operations for this type are not shown since they are straightforward.
2. Next the filters needed for Mergesort are defined by extending the standard filter class. We need a filter for the source of the stream to be sorted. The source generates data via a random number generator. In addition we need a Merger for doing the actual work and a Printer to display the final result. Splitters and joiners are also defined as shown in Fig. 12. Note that for space-considerations we had to move the implementations of the above filters' `Work`-operations to the Appendix.

Fig. 11 Root_Data_Type.Int

```
package Root_Data_Type.Int is

   type int is new Root_Data_Type with record
      i: integer;
   end record;

   function "+" (Left, Right : Int) return Int;

   function "<=" (Left, Right : Int) return boolean;

end Root_Data_Type.Int;
```

```
with Root_Data_Type.Int;
with Base_Filter;
with Filter;
with Splitter;
with Joiner;

package UserFilters is

   package Int_Filter is new Filter (Root_Data_Type.Int.int, Root_Data_Type.Int.int);

   type Merger(aValue : integer) is new Int_Filter.Filter with record
      N : integer := aValue;
   end record;
   procedure Work(f: access Merger);

   type Source is new Int_Filter.Filter with null record;
   procedure Work(f: access Source);

   type Printer is new Int_Filter.Filter with null record;
   procedure Work(f: access Printer);

   package Int_Splitter is new Splitter (Root_Data_ Type.Int.int);

   package Int_Joiner is new Joiner (Root_Data_ Type.Int.int);

end UserFilters;
```

Fig. 12 Mergesort_Filters

3. Procedure Main[4] uses the recursive function `SetUp_MergeSort` to setup the stream graph needed by Mergesort. This is done in a standard way. A reference to this can be found in almost any book on algorithms and data structures. An example of the stream graph for $N = 8$ items to be sorted is shown in Fig. 10. Runtime arguments of Main are the number of CPU cores to use and the number of iterations of the stream graph.

4 The language-independent runtime system

We implemented the StreamPI runtime system (RTS) as an Ada package that must be compiled and linked with applications that wish to use the StreamPI functionality. For C++ we implemented a binding to call Ada 2005 code from C++. Our RTS package contains several child packages as shown in Fig. 13. It exports only

[4]Shown in the Appendix.

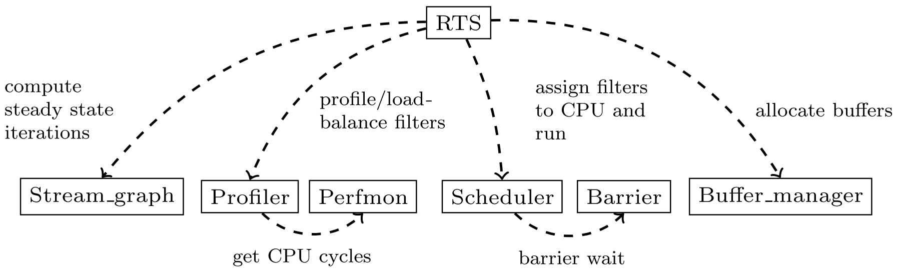


Fig. 13 Component diagram for the StreamPI runtime system

```
with Base_Filter;

package RTS is

   Stream_Type_Error : exception;
-- Raised with connections of type-incompatible filters.

   Invalid_Stream_ Graph : exception;
-- Raised for a stream that has no steady state.

   procedure Connect(From : Base_Filter.Base_Filter_Ptr;
                     To : Base_ Filter.Base_Filter_Ptr;
                     out_weight : positive := 1;
                      in_weight : positive := 1);

    procedure Run (NrCPUs : Positive; NrIterations : Natural);

    procedure Stop;

 end RTS;
```

Fig. 14 RTS runtime system package specification

three procedures, as depicted in Fig. 14. Procedure `Connect` is used by our generic implementations of filters, splitters and joiners. The `Connect` operations from the `Base_Filter` type hierarchy invoke `RTS.Connect` to inform the runtime system about connections between actors. Component RTS.Stream_Graph maintains the stream-graph topology from calls to `RTS.Connect`.

The C++ binding to the Ada 2005 runtime system is a thin binding that maps C++ procedures to their Ada equivalent. We do not propagate exceptions from RTS to C++ client code. Instead, exceptions are trapped in Ada wrapper functions that pass an error code back to the C++ binding. Based on the error code the C++ binding will raise the corresponding C++ exception.

Once a stream graph has been created, the RTS client calls `RTS.Run` to execute the stream graph on `NrCPUs` for `NrIterations`. (The steady-state schedule of the stream graph is executed `NrIterations` times; for every iteration, all actors are executed the required number of times for the system to remain in steady state (see also Fig. 3).) At this stage RTS executes the following steps:

1. The steady state for the given stream graph is calculated as outlined in Sect. 2. We use the algorithm from [3] which has time-complexity linear in the number of actors and edges in the input stream graph. For a stream graph that has no steady state, RTS raises exception Invalid_Stream_Graph. This is not a limitation of our

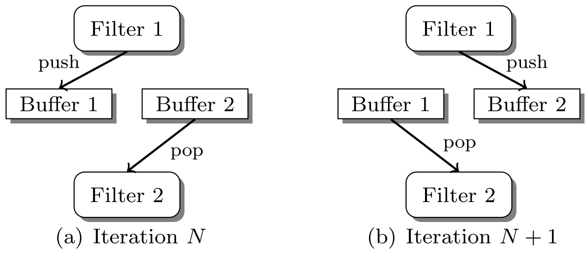


Fig. 15 Double-buffering of a data channel between actors of iteration distance 1

framework but a manifestation of a defective stream graph structure: due to sample rate inconsistencies any schedule for such a graph will result either in deadlock or unbounded buffer sizes [30].

2. Buffers representing data channels are allocated between adjacent actors. The size of a buffer is computed as the data type size times the input or output rate times the number of steady-state iterations of the respective actor times the iteration distance plus one of the adjacent actors. (See also Fig. 3 and Fig. 15.)
3. A boot schedule to bring the stream graph into steady state is computed and executed on the actors. During this bootup-phase the actors are profiled to determine the execution times of their work functions. It should be noted that because of side-effects (e.g., print functions or state that is maintained inside of actors) profiling can only be done as part of the stream-program execution itself (i.e., we cannot profile and then start execution from scratch). Profiling uses the x86-64's hardware cycle counters exported by the clock library from [6]. Based on the execution times the actors are allocated to CPUs using a simple but fast greedy algorithm: actors are sorted from largest to smallest work function execution time. CPU allocation happens then in a round-robin fashion from the sorted list of actors. At each allocation step, the actor with the so far least amount of assigned work is selected.
4. For every CPU (core) a scheduler from package RTS.Schedulers is created and the corresponding actors are registered with the scheduler. A scheduler is an Ada task that maintains a list of registered actors together with the corresponding numbers of steady-state iterations. Invocation of a scheduler's `Run` entry initiates execution of the registered actors' work functions.
5. Stream graph execution is initiated with the schedulers.

There is no need for synchronization of actor execution within a single CPU, because schedulers invoke actor work functions sequentially. However, across CPUs, schedulers need to be synchronized. We require only a single barrier for scheduler synchronization: per stream graph iteration, a scheduler will invoke all registered actors and then wait at the barrier for the other schedulers to complete the stream graph iteration.

This is possible because actors with a producer–consumer relationship are advanced to different iterations of the steady-state schedule during program boot time. Thereby the consumer's dependency is shifted from the data of the current steady-state iteration to the data of the previous steady-state iteration. This technique is called coarse-grained software pipelining; it was introduced in [13], in conjunction with the communication-exposed Raw 4×4 tiled multicore architecture [36]. We adapted coarse-grained software-pipelining for shared-memory multiprocessors by using a barrier.

To keep the barrier synchronization overhead low, we employ a variant of the shared-memory barrier synchronization algorithm from [15]. We use multiple synchronization variables aligned to separate cache lines to avoid false sharing between cores. This is an improvement over StreamIt, where synchronization on shared-memory multiprocessors is done via POSIX mutexes and condition variables. It should be noted that with our approach no context switch is required, because schedulers perform busy waiting inside of the barrier. This saves OS overhead associated with context switching. StreamPI also supports a blocking barrier implementation based on Ada protected objects, if relinquishing the CPU during barrier synchronization is preferred.

As outlined in Fig. 15, we employ multiple buffers between adjacent actors, which is due to coarse-grained software pipelining. Multiple buffers ensure that both the reader and the writer have their own buffer and need not synchronize on every buffer access. After every steady-state iteration, schedulers synchronize on the barrier and advance their actors' read and write buffer pointers before continuing with the next iteration. Barriers in conjunction with multi-buffering reduce synchronization overhead among schedulers and keep the computation-to-communication ratio of stream programs high. Again, this improves over the StreamIt solution on shared-memory multiprocessors where synchronization is required on every individual buffer access.

Furthermore, we employ stream graph unrolling for an additional reduction of synchronization overhead. With stream graph unrolling, multiple iterations of the steady-state schedule are executed before synchronization at the barrier.

5 Experimental results

The StreamIt benchmark collection [44] is commonly used for the evaluation of approaches to stream program orchestration [11, 13, 27, 29, 40, 48, 49]. We chose eight benchmarks from this collection (benchmarks 1–7 and 13 in the below description), which enabled us to compare our approach with the StreamIt compiler framework. We chose five additional programming problems for which the stream programming paradigm is a natural approach.

1. MatMult: multiplies two square matrices by transposing one of them and multiplying in parallel.
2. Mergesort: this benchmark uses a stream of random integers, reads N elements from the stream and outputs the N elements in sorted order (as outlined in Sect. 3).
3. BitonicSort: sorts sequences of integers using a sorting network. It recursively divides the given sequence into two halves and merges each half into monotonically increasing or monotonically decreasing order until all the elements of the sequence are sorted.
4. RadixSort: a binary implementation of the radix-sort algorithm [39].

Table 1 Characteristics of benchmark programs implemented with StreamPI

Benchmark	Filters	Splitters	Joiners
Matrix Multiply	15	2	2
MergeSort	65	31	31
Bitonic Sort	500	241	241
RadixSort	16	0	0
Block Matrix Multiply	38	8	8
Comparison Counting	20	2	2
DCT	18	2	2
Distance	160	40	37
Markov	480	269	269
DFT	133	19	17
Leontif	385	199	180
Reachability	160	40	37
JPEG	65	10	10

5. BlockMatMult: Block matrix multiplication splits each matrix in the stream into blocks and multiplies blocks with small communication overhead; blocks are then added and combined.
6. CompCount: sorting by comparison counting.
7. DCT: implementation of the discrete cosine transformation.
8. Distance: computation of the geodesic distance of two nodes of a graph.
9. Markov: computation of a Markov chain.
10. DFT: computation of the discrete Fourier transform (DFT) of a series of signals.
11. Leontief: Computation of a Leontief input-output model to predict the performance of economies.
12. Reachability: computation of the reachability matrix of a graph.
13. JPEGCompr: JPEG compression of images.

Table 1 shows the characteristic features of our benchmark programs, with up to 1018 actors for the Markov benchmark. All benchmarks were implemented with StreamPI and compiled with the 64-bit version of GNAT GPL 2009 (20090511).

To determine the scalability of the StreamPI implementation with respect to the number of CPU cores, we executed all benchmarks on a 2 CPU Intel Xeon X7560 server for $P = 2, 4, 8$ and 16 cores. The results are depicted in Fig. 16.

Matrix multiplication did not scale well because of a join node preceding the sink node. This join node had two incoming stream graph edges with high data-rates, which caused a very high CPU load and made this node a bottleneck. Block matrix multiplication (BlockMatMult) showed a much better speedup of 1.86, 3.52, 5.69 and 9.44, because it did not exhibit a bottleneck actor and contained more coarse-grained parallelism.

The work functions of Mergesort showed very fine-grained parallelism; under those circumstances scalability was reasonable. All other benchmarks showed good scalability, up to a factor of 12.52 for 16 cores with DCT.

To evaluate the synchronization overhead and the load-balance achieved with StreamPI, we instrumented the runtime system to record the execution times for actor

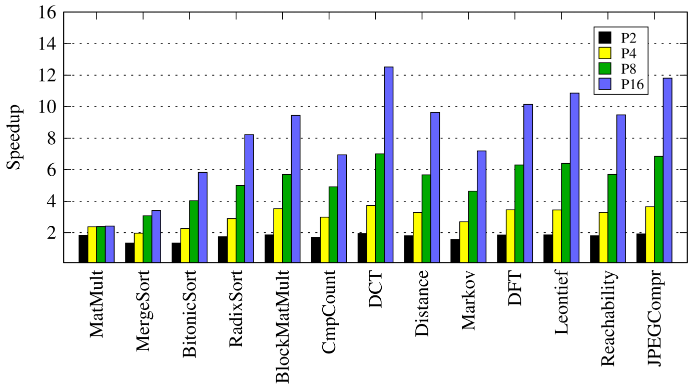


Fig. 16 Speedups obtained for 2, 4, 8 and 16 cores

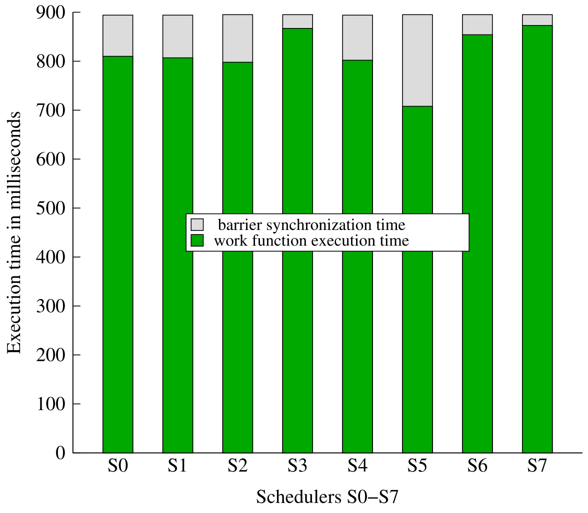


Fig. 17 Workload distribution for the Leontief benchmark for $P = 8$ cores running schedulers S0–S7

execution and barrier synchronization. Figure 17 shows the workload distribution for $P = 8$ cores for the Leontief benchmark for 1000 stream graph iterations. The lower bars show the overall time that a given scheduler was executing actor work functions, while the upper bars depict the overall time spent for barrier synchronization. For the example in Fig. 17, scheduler S7 had the highest workload and arrived latest at the barrier, resulting in 22 ms or 2.45% of the overall execution time spent for barrier synchronization. For 1000 stream graph iterations, the time per barrier synchronization was thus 22 μs on the Xeon X7560 CPU. Scheduler S5 had the smallest workload and spent 708 ms or 79.1% for actor execution. Notwithstanding barrier synchronization times, this means that the load-balance showed less than 19% deviation between the most and least loaded schedulers. Note that the load-balance depends on the accuracy of the actor execution times determined at program boot time (see Sect. 4), the actual distribution of those execution times, and the quality achieved by the greedy actor allocation algorithm.

Table 2 Performance comparison between StreamIt and StreamPI

Benchmark	StreamPI Speedup (%)
BitonicSort	31.2
Mergesort	150.0
Matrix Multiply	−42.6
Block Matrix Multiply	−39.9
Comparison Counting	315.4
RadixSort	−21.3
DCT	52.9

We compared the performance of StreamPI with the StreamIt language and compiler framework [1, 13, 43, 44, 47]. For those benchmarks of our benchmark selection where a version for StreamIt exists, we used the StreamIt compiler [44, v. 2.1.1] to compile benchmarks for an 8-core shared-memory multicore architecture. For benchmarks which the StreamIt compiler could not scale up to eight cores, we reverted to four cores. Because of StreamIt's compile-time optimizations and its static compilation model, we expected StreamIt to be generally faster than StreamPI. However, StreamPI was faster for several benchmarks (see Table 2), with more than 300% for ComparisonCounting. We suspect the following StreamIt attributes to contribute to this result.

1. StreamIt applies a static actor profiling heuristic. StreamPI profiles actors during stream graph boot time, which gives much more accurate execution times and thus better load-balance.
2. StreamIt uses POSIX mutexes and condition variables to synchronize buffer accesses between adjacent actors. In contrast, StreamPI requires only a single in-memory barrier synchronization per iteration of the stream graph.

6 Conclusions and future work

In this paper we have proposed a stream-parallel programming abstraction to extend object-orient programming languages with stream-programming facilities. StreamPI provides a class hierarchy for actor-specification that is embedded in an object-oriented host language. Actors use the underlying StreamPI runtime system through a language-specific binding. StreamPI's runtime system itself is language independent. It manages actor allocation, stream graph creation and stream program execution on multicore architectures.

To show the feasibility of our approach, we have embedded the StreamPI programming abstractions in Ada 2005 and C++. StreamPI is non-intrusive in the sense that no modifications of the host language and compiler are required. Legacy-code can be mixed with a stream-parallel application, and the use of sequential legacy code with actors is supported. Unlike previous approaches for stream program orchestration on multicore architectures, StreamPI allows dynamic creation of stream programs, and the full spectrum of data types available in the host language can be used for streaming. Each StreamPI filter has an input and output type. That way filters are allowed to type-convert data, which allows the generation of heterogeneous stream graphs. Splitters and joiners are restricted to a single type. StreamPI provides type-safety through runtime typechecking during stream graph creation time.

We have implemented StreamPI on Intel multicore architectures for C++ and Ada 2005, and provided experimental results that show the effectiveness of our approach on an Intel Xeon X7560 server with two CPUs and eight cores per CPU. StreamPI profiles actors at stream graph boot time to derive actor execution times and an allocation of actors onto available cores. These activities are carried out by the StreamPI runtime system transparently, i.e., without user intervention. Compared to previous orchestration approaches on shared-memory architectures, StreamPI minimizes the synchronization overhead between adjacent actors through multi-buffering and a single in-memory barrier. Surveyed benchmarks contained up to 1018 actors, with speedups up to a factor of 12.52 for 16 cores.

Despite the added flexibility of StreamPI, performance compares favorably to StreamIt, although StreamIt uses a static compilation model and a set of program transformations to increase parallelism and the computation to communication ratio of the underlying stream program.

Further research is required to determine the trade-offs between the high-level stream program abstraction and the conventional thread-and-lock-based programming approach. This comparison will have to take into account that stream programs “naturally” contain pipeline-parallelism between tasks with producer-consumer relationships. Thread-and-lock-based programs only contain pipeline parallelism if explicitly encoded by the programmer.

As for further future work, we plan to survey the stream graph patterns arising from real-world applications and extend StreamPI with higher-level stream graph constructs for commonly occurring patterns.

Acknowledgements Research partially supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (No. 2010-0005234), and the OKAWA Foundation Research Grant (2009).

Appendix: Mergesort example

A.1 Filters

```
with Gnat.Io; use Gnat.Io;
with Ada.Numerics.Discrete_ Random;
with Root_Data_Type.Int;
use Root_Data_Type.Int;
with Filter;

package body UserFilters is

    procedure Work(F: access Merger) is
        type Integer_Array is array (Positive range 1..F.N) of Root_Data_Type.Int.Int;
        Index1 : Integer := 1;
         Index2 : Integer := 2;
         Value1 : Root_ Data_Type.Int.Int;
         Value2 : Root_ Data_Type.Int.Int;
         Tmp : Integer_ Array;
         Leftover : Integer;
    begin
        for I in 1..F.N loop
            Value1 := F.Pop;
            Tmp(I) := Value1;
        end loop;

        while Index1 <= F.N and Index2 <= F.N loop
            Value1 := Tmp(Index1);
            Value2 := Tmp(Index2);
            if Value1 <= Value2 then
                F.Push(Value1);
                Index1 := Index1 + 2;
            else
                F.Push(Value2);
                Index2 := Index2 + 2;
            end if;
        end loop;

        if Index1 <= F.N then
            Leftover := Index1;
        else
            Leftover := Index2;
        end if;

        while Leftover <= F.N loop
            Value1 := Tmp(Leftover);
            F.Push(Value1);
            Leftover := Leftover + 2;
        end loop;
    end Work;

    type Rand_ Integer is range 1.. 100;
    package Random_Integer is new Ada.Numerics.Discrete_ Random(Rand_Integer);
    use Random_ Integer;
    G : Generator;

    procedure Work(F: access Source) is
        Item : Root_Data_ Type.Int.Int;
        I : Rand_ Integer;
    begin
        I := Random(G);
        Item.I := Positive(I);
        F.Push(Item);
    end Work;


    procedure Work(F: access Printer) is
        Item1 : Root_Data_ Type.Int.Int;
    begin
        Item1 := F.Pop;
        Put_Line( “sink  :”&Integer’Image(Item1.I));
    end Work;

end UserFilters;
```

A.2 Driver program

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Command_Line; use Ada.Command_Line;
with Gnat.Os_Lib;
with Base_Filter;
with UserFilters;
use UserFilters;
with RTS;

procedure Main is

    NrCPUs : Positive := 1;
    NrIterations : Natural := 1;

    procedure Program_Exit is
    begin
       Put_Line ("Usage:  " & Command_Name
                 & " <NrCPUs> <NrSteadyStateIterations>");
       Gnat.Os_Lib.OS_ Exit (1);
    end Program_Exit;

    function SetUp_MergeSort(In_Filter : Base_Filter.Base_Filter_Ptr;
                             N : Natural; First:Boolean)
                            return Base_Filter.Base_Filter_Ptr
    is
       Merge : Base_ Filter.Base_Filter_Ptr;
       L : Base_ Filter.Base_Filter_Ptr;
       R : Base_ Filter.Base_Filter_Ptr;
       Split : Base_ Filter.Base_Filter_Ptr;
       Join : Base_ Filter.Base_Filter_Ptr;
    begin
       if N = 2 then
          Merge := new  Userfilters.Merger(2);
          In_ Filter.Connect(Merge, 2, 2);
          return Merge;
       end if;

       Split := new  UserFilters.Int_Splitter.Splitter;
       if First then
          In_ Filter.Connect(Split, 1, N);
       else
          In_ Filter.Connect(Split, N, N);
       end if;
       L := SetUp_ MergeSort(Split, N/2, False);
       R := SetUp_ MergeSort(Split, N/2, False);
       Join := new Userfilters.Int_Joiner.Joiner;
       L.Connect(Join, N/2, 1);
       R.Connect(Join, N/2, 1);
       Merge := new  Userfilters.Merger(N);
       Join.Connect(Merge, 2, N);
       return Merge;
    end SetUp_ Mergesort;

    M : Base_ Filter.Base_Filter_Ptr;
    S : Base_ Filter.Base_Filter_Ptr := new UserFilters.Source;
    P : Base_ Filter.Base_Filter_Ptr := new UserFilters.Printer;
begin
    if Ada.Command_Line.Argument_Count /= 0
      and then Ada.Command_Line.Argument_Count /= 2
    then
       Program_Exit;
    end if;
    if Ada.Command_Line.Argument_Count = 2 then
       declare
       begin
          NrCPUs := Integer'Value(Ada.Command_Line.Argument (1));
          NrIterations := Integer'Value(Ada.Command_Line.Argument (2));
       exception
          when others =>
             Program_ Exit;
       end;
    end if;

    M := SetUp_ MergeSort(S, 16, True);
    M.Connect(P,16,1);
    RTS.Run(NrCPUs, NrIterations);

end Main;
```

References

1. Amarasinghe S, Gordon MI, Karczmarek M, Lin J, Maze D, Rabbah RM, Thies W (2005) Language and compiler design for streaming applications. Int J Parallel Program 33(2):261–278
2. Andrews J, Baker N (2006) Xbox 360 system architecture. IEEE MICRO 26(2):25–37
3. Battacharyya SS, Lee EA, Murthy PK (1996) Software synthesis from dataflow graphs. Kluwer Academic, Norwell
4. Belina F, Hogrefe D (1989) The CCITT-specification and description language SDL. Comput Netw 16:311–341
5. Berry G, Gonthier G (1992) The Esterel synchronous programming language: design, semantics, implementation. Sci Comput Program 19(2):87–152
6. Bryant RE, O'Halloran DR (2003) Computer systems: a programmer's perspective. Prentice-Hall, New York
7. Buttlar D, Farrell J, Nichols B (1996) PThreads programming. O'Reilly, Sebastopol
8. Carpenter PM, Ramirez A, Ayguade E (2009) Mapping stream programs onto heterogeneous multiprocessor systems. In: CASES '09: proceedings of the 2009 international conference on compilers, architecture, and synthesis for embedded systems. ACM Press, New York, pp 57–66
9. Caspi P, Pilaud D, Halbwachs N, Plaice J (1987) Lustre: a declarative language for programming synchronous systems. In: Proceedings of the 14th ACM conference on principles of programming languages, pp 178–188
10. Chen MK, Li XF, Lian R, Lin JH, Liu L, Liu T, Ju R (2005) Shangri-la: Achieving high performance from compiled network applications while enabling ease of programming. In: PLDI '05: proceedings of the 2005 ACM SIGPLAN conference on programming language design and implementation. ACM Press, New York
11. Farhad SM, Ko Y, Burgstaller B, Scholz B (2011) Orchestration by approximation: mapping stream programs onto multicore architectures. In: Proceedings of the sixteenth international conference on architectural support for programming languages and operating systems, ASPLOS '11, New York, NY, USA. ACM Press, New York, pp 357–368
12. Google (2009) The Go programming language specification, retrieved Nov 2009. http://golang.org
13. Gordon MI, Thies W, Amarasinghe S (2006) Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In: International conference on architectural support for programming languages and operating systems, San Jose, CA
14. Gummaraju J, Rosenblum M (2005) Stream programming on general-purpose processors. In: MICRO 38: proceedings of the 38th annual IEEE/ACM international symposium on microarchitecture. IEEE Computer Society Press, Los Alamitos, pp 343–354
15. Gupta R, Hill CR (1990) A scalable implementation of barrier synchronization using an adaptive combining tree. Int J Parallel Program 18:161–180
16. Hagiescu A, Wong W, Bacon DF, Rabbah R (2009) A computing origami: folding streams in FPGAs. In: DAC '09: proceedings of the 2009 design automation conference. ACM Press, New York
17. Herlihy M, Shavit N (2008) The art of multiprocessor programming. Morgan Kaufmann, San Mateo
18. Hofstee HP (2005) Power efficient processor architecture and the Cell processor. In: HPCA '05: proceedings of the 2005 international symposium on high-performance computer architecture. IEEE Computer Society Press, Los Alamitos, pp 258–262
19. Hormati AH, Choi Y, Kudlur M, Rabbah R, Mudge T, Mahlke S (2009) Flextream: Adaptive compilation of streaming applications for heterogeneous architectures. In: PACT '09: proceedings of the 2009 18th international conference on parallel architectures and compilation techniques, Washington, DC, USA. IEEE Computer Society Press, Los Alamitos, pp 214–223
20. IBM Redbooks (2008) Programming the cell broadband engine architecture: examples and best practices. http://www.redbooks.ibm.com
21. IDC (2008) PC semiconductor market briefing: re-architecting the PC and the migration of value, June 2008, http://www.idc.com
22. ISO/IEC 8652:2007 (2006) Ada reference manual, 3rd edn
23. Kahn G (1974) The semantics of a simple language for parallel programming. In: Rosenfeld JL (ed) Information processing, Stockholm, Sweden, Aug. North Holland, Amsterdam, pp 471–475
24. Kapasi UJ, Dally WJ, Rixner S, Owens JD, Khailany B (2002) The imagine stream processor. In: Computer design, international conference on, p 282
25. Karczmarek M (2002) Constrained and phased scheduling of synchronous data flow graphs for the StreamIt language. Master's thesis, Massachusetts Institute of Technology

26. Karczmarek M, Thies W, Amarasinghe S (2003) Phased scheduling of stream programs. ACM SIGPLAN Not 38(7):103–112
27. Karczmarek M, Thies W, Amarasinghe S (2003) Phased scheduling of stream programs. In: LCTES ’03: proceedings of the 2003 ACM SIGPLAN/SIGBED conference on languages, compilers, and tools for embedded systems, vol 38, pp 1235–1245
28. Karp RM, Miller RE (1966) Properties of a model for parallel computations: determinacy, termination, queueing. SIAM J Appl Math 14(6):1390–1411
29. Kudlur M, Mahlke S (2008) Orchestrating the execution of stream programs on multicore platforms. In: PLDI ’08: proceedings of the 2008 ACM SIGPLAN conference on programming language design and implementation. ACM Press, New York
30. Lee EA, Messerschmitt DG (1987) Static scheduling of synchronous data flow programs for digital signal processing. IEEE Trans Comput 36(1):24–35
31. Lee EA, Messerschmitt DG (1987) Synchronous data flow. Proc IEEE 75(9):1235–1245
32. Leung M-K, Liu I, Zou J (2008) Code generation for process network models onto parallel architectures. Technical Report UCB/EECS-2008-139, EECS Department, University of California, Berkeley
33. Lin C, Snyder L (2008) Principles of parallel programming. Addison-Wesley, Reading
34. Lin Y, Choi Y, Mahlke SA, Mudge TN, Chakrabarti C (2008) A parameterized dataflow language extension for embedded streaming systems. In: SAMOS ’08: proceedings of the 2008 international conference on embedded computer systems: architectures, modeling, and simulation, pp 10–17
35. Mattson TG, Sanders BA, Massingill BL (2007) Patterns for parallel programming, 3rd edn. Addison-Wesley, Reading
36. Michael EW, Taylor M, Sarkar V, Lee W, Lee V, Kim J, Frank M, Finch P, Devabhaktuni S, Barua R, Babb J, Amarasinghe S, Agarwal A (1997) Baring it all to software: the Raw machine. Computer 30:86–93
37. Pacheco PS (1996) Parallel programming with MPI. Morgan Kaufmann, San Francisco
38. Reinders J (2007) Intel threading building blocks. O’Reilly, Sebastopol
39. Sedgewick R (2002) Algorithms in C++, 3rd edn. Addison-Wesley-Longman, Reading
40. Sermulins J, Thies W, Rabbah R, Amarasinghe S (2005) Cache aware optimization of stream programs. In: LCTES ’05: proceedings of the 2005 ACM SIGPLAN/SIGBED conference on languages, compilers, and tools for embedded systems. ACM Press, New York, pp 115–126
41. Spring JH, Privat J, Guerraoui R, Vitek J (2007) StreamFlex: High-throughput stream programming in Java. In: OOPSLA ’07: proceedings of the 2007 ACM SIGPLAN conference on object-oriented programming systems and applications
42. Stephens R (1997) A survey of stream processing. Acta Inform 34:491–541
43. StreamIt research group (2006) StreamIt Cookbook. Online reference manual. Massachusetts Institute of Technology
44. StreamIt Web Site (2010) http://groups.csail.mit.edu/cag/streamit/, retrieved Dec 2010
45. Thies W (2009) Language and compiler support for stream programs. PhD thesis, Massachusetts Institute of Technology
46. Thies W, Amarasinghe S (2010) An empirical characterization of stream programs and its implications for language and compiler design. In: PACT ’10 proceedings of the 2010 conference on parallel architectures and compilation techniques. ACM Press, New York
47. Thies W, Karczmarek M, Amarasinghe SP (2002) StreamIt: A Language for Streaming Applications. In: CC ’02: proceedings of the 11th international conference on compiler construction, London, UK, LNCS. Springer, Berlin, pp 179–196
48. Udupa A, Govindarajan R, Thazhuthaveetil MJ (2009) Software pipelined execution of stream programs on GPUs. In: CGO ’09: proceedings of the 7th Annual IEEE/ACM international symposium on code generation and optimization. IEEE Computer Society Press, Los Alamitos
49. Udupa A, Govindarajan R, Thazhuthaveetil MJ (2009) Synergistic execution of stream programs on multicores with accelerators. In: LCTES ’09: proceedings of the 2009 ACM SIGPLAN/SIGBED conference on languages, compilers, and tools for embedded systems
50. Wei H, Yu J, Yu H, Gao GR (2010) Minimizing communication in rate-optimal software pipelining for stream programs. In: CGO ’10: proceedings of the 8th annual IEEE/ACM international symposium on code generation and optimization. ACM Press, New York, pp 210–217
51. Zhang D, Li QJ, Rabbah R, Amarasinghe S (2008) A lightweight streaming layer for multicore execution. SIGARCH Comput Archit News 36(2):18–27
52. Zhang D, Li Z, Song H, Liu L (2005) A programming model for an embedded media processing architecture. In: SAMOS ’05: proceedings of the 2005 international conference on embedded computer systems: architectures, modeling, and simulation, LNCS. Springer, Berlin