AdaStreams: A Type-Based Programming Extension for Stream-Parallelism with Ada 2005

Gingun Hong[1], Kirak Hong[1], Bernd Burgstaller[1], and Johann Blieberger[2]

[1] Yonsei University, Korea
[2] Vienna University of Technology, Austria

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 data streams. The coarse-grained parallelism exposed in stream programs facilitates such an efficient mapping of actors onto the underlying hardware.

In this paper we propose a type-based stream programming extension to Ada 2005. AdaStreams is a type-hierarchy for actor-specification together with a run-time system that supports the execution of stream programs on multicore architectures. AdaStreams is non-intrusive in the sense that no change of an Ada 2005 programming language implementation is 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, AdaStreams allows creation and subsequent execution of stream programs at run-time.

We have implemented AdaStreams for Intel multicore architectures. We provide initial experimental results that show the effectiveness of our approach on an Intel X86-64 quadcore processor. The initial release of our work is available for download at [1].

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 [12] processor provides 9 processing cores, Microsoft's Xbox CPU [2] has 3 cores, and more than 90% of all PCs shipped today have at least 2 cores. According to a recent survey conducted by IDC [13], all PCs (desktops, mobile and servers) will be multi-cores in 2010, with quad and octal cores together already constituting more than 30% market share. For programming languages it becomes therefore increasingly important to provide programming abstractions that work efficiently on parallel architectures.

J. Real and T. Vardanega (Eds.): Ada-Europe 2010, LNCS 6106, pp. 208–221, 2010.

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 itself. 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) [19] 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 [14,11] 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: applications should scale with the number of cores of the underlying hardware. Encoding a programming problem using a fixed set of threads limits scalability.
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 [19] 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. Pipeline parallelism is so common in parallel programs that it was selected as a distinguished parallel programming pattern [20]. Because pipeline parallelism operates on a conceptually infinite data stream, it is often called stream parallelism.

Stream-parallel programs consist of a set of independent actors that communicate via 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 can execute if sufficient data is available on its input channels and if the output channels provide enough

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

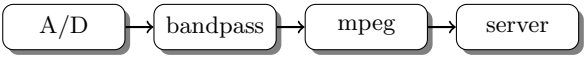


Fig. 1. Example stream program

space to accommodate the data produced by the actor. Because of this lack of dependencies stream programs provide a vast amount of parallelism, which makes them well-suited to run on multi-core architectures.

Fig. 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 and performance. A programming language implementation 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.

The contributions of this paper are as follows.

- We present a type-based stream programming extension for Ada 2005. Our extension lifts the abstraction level for the development of stream programs. Actors are expressed as tagged types and conveniently connected via a single method call.
- We provide design and implementation of a run-time system for Ada 2005 that allows the execution of stream programs. Our run-time system manages the data channels between actors, load-balances and schedules actors among the parallel execution units of a processor, and provides the complete stream program execution infrastructure.
- Unlike previous approaches, we allow the dynamic creation of stream graphs. Instead of applying heuristics, we profile stream programs to load-balance actors among the parallel execution units of a processor.
- The initial release of AdaStreams is available for download at [1].

The remainder of this paper is organized as follows: in Sec. 2 we provide background information and survey related work. In Sec. 3 we introduce the type-based programming abstractions for stream-parallelism proposed for Ada 2005. In Sec. 4 we describe the design and implementation of the run-time system required to support applications that use our programming abstractions for stream parallelism. Sec. 5 contains our evaluation of AdaStreams on the Intel x86-64 architecture. We draw conclusions and outline future work in Sec. 6.

2 Background and Related Work

A survey on programming languages that include a concept of streams can be found in [22]. For example, Lustre [8] is a synchronous dataflow programming language used for safety-related software in aircrafts, helicopters, and nuclear power plants. However, it 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.

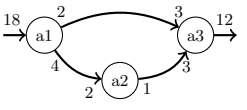
StreamIt [24] uses syntax similar to Java and is more flexible than its predecessors. A StreamIt programmer constructs a stream graph consisting of filters which are connected by a fixed number of constructs: Pipelines, SplitJoins, and FeedbackLoops. Two types of splitters are supported: Duplicate and RoundRobin. A FeedbackLoop allows 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.

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.

Our approach differs from Kahn process networks [15] which allow data-dependent communication. Leung et al. show in [18] how Kahn process networks can be mapped onto parallel architectures using MPI for communication.

Summing up, our approach for AdaStreams goes beyond that of StreamIt because we allow dynamic creation of stream graphs. Messaging can be done via standard Ada features such as protected objects. In contrast to the languages mentioned above, the whole spectrum of data types available in Ada can be used for streaming. However, currently we do not provide predefined structured stream graphs like StreamIt does with its Pipelines, SplitJoins, and FeedbackLoops. The filters, splitters and joiners provided by the AdaStreams library are sufficient to generate structured graphs. For example, Fig. 3(d) shows how a feedback loop can be constructed with AdaStreams. As explained in [23], it is yet not entirely clear whether structured stream graphs are sufficient for all possible applications. Our plan with AdaStreams is to survey the stream graph patterns arising from real-world applications and build higher-level stream graph constructs from commonly occurring patterns.

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 amount of parallelism it is still a challenging task to schedule a stream program on a parallel architecture. The obvious solution of assigning an Ada task 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 AdaStreams require that the amount of data consumed and produced by an



Actor	Iterations
a1	3
a2	6
a3	2

Fig. 2. Example: SDF and minimal steady-state schedule

actor is known *a priori*. Stream graphs with this property employ synchronous data-flow (SDF). Figure 2 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. 2 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→a3. Actor a3 will consume those 6 data items during its two executions. Computing the steady state for SDF graphs has been studied in [17][2]. StreamIt uses a variant of this algorithm for structured SDF graphs [16]. An SDF 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.

The stream programming paradigm is also applied in Google's recently-released systems programming language "Go" [10]. Go provides co-routines that communicate via channels.

3 A Type-Based Programming Abstraction for Stream Parallelism with Ada 2005

To add a stream programming abstraction to the Ada programming language, the following approaches are conceivable: (1) extend the core language itself through language extensions, (2) provide a compiler extension for streaming constructs, or (3) provide a programming library that the user can link with standard Ada application code. AdaStreams 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. We felt that at this stage the stream programming paradigm is still in the state of flux, which suggests to choose a library as a light-weight approach to begin with. Ada provides excellent support for packages, types and

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

generic programming, which facilitates library creation. A library-only solution is of course not perfect. We had to omit features that require compiler support, like filter fusion/fission to improve load-balancing of stream programs on multi-core architectures. However, libraries have been successfully applied to extend programming languages, as demonstrated by the POSIX threads library [7] and by the Intel Thread Building Blocks [21].

Fig. 3 shows the three actor programming primitives that AdaStreams provide: filters, splitters and joiners. Together, these primitives are sufficient to generate arbitrary stream graph structures. Fig. 3(d) shows how a loop can be constructed from a joiner, a filter and a splitter.

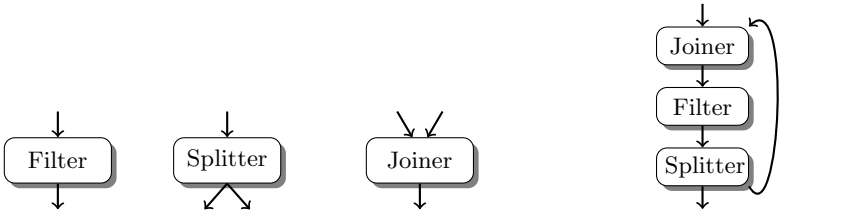


Fig. 3. Three AdaStreams stream-graph primitives and one compound statement

Each AdaStreams filter has an input and output type. That way filters are allowed to convert data, which allows the generation of heterogeneous stream graphs. Splitters and joiners are restricted to a single type. Types are used during stream graph creation to ensure type compatibility of adjacent stream graph primitives. Users may define arbitrary types by extending our abstract tagged root type `Root_Data_Type` as depicted in Fig. 4.

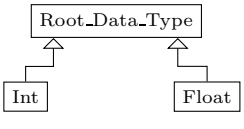


Fig. 4. Root data type hierarchy

We chose a hierarchy of tagged types depicted in Fig. 5 to represent stream program actors. The abstract type `Base_Filter` at the root of this hierarchy contains the commonalities among actors. Types `Filter`, `Splitter` and `Joiner` are generic types parameterized by the respective input or input and output types from the root data type hierarchy.

Type `Base_Filter` is depicted in Fig. 6. Every actor has to provide a primitive operation named `Work` (line 5) which encodes the actor's computation. The `Base_Filter` record can be extended by actors that need to keep state information across invocations of the work function.

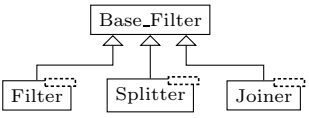


Fig. 5. Type hierarchy for AdaStream filters

```
1   with Root_Data_Type;
2   package Base_Filter is

3     type Base_Filter is abstract tagged private;
4     type Base_Filter_Ptr is access all Base_Filter'Class;

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

6     procedure Connect(f: access Base_Filter;
7                       b: access Base_Filter'Class;
8                       out_weight: Positive := 1;
9                       in_weight: Positive := 1) is abstract;

10    function Get_In_Type(f: access Base_Filter)
11                    return Root_Data_Type.Root_Data_Type'Class is abstract;
12    procedure Set_In_Weight (f: access Base_Filter; in_weight : positive) is abstract;
13  private
14    type Base_Filter is abstract tagged null record;
15  end Base_Filter;
```

Fig. 6. Base_Filter type

Every actor needs a `Connect` operation (lines 6–9) to attach its streamgraph successor(s). The arguments to the `Connect` operation are the downstream successor (line 7) and the number of output data items (line 8) of this actor plus the number of input data items (line 9) of the downstream successor. For example, to connect actors X and Y via edge $X \xrightarrow{1\ 2} Y$, operation `X.Connect(Y,1,2)` would be used by the AdaStreams 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 Sec. 2.

At run-time, operation `Connect` checks that the data types used in the filters to be connected are equivalent. Operation `Get_In_Type` (lines 10 and 11 in Fig. 6) is used to retrieve the input type of the downstream actor. If the data types differ, exception `RTS.Stream_Type_Error` is raised. Hence we combine a type secure approach with dynamic creation of arbitrary stream graphs.

Filters, splitters, and joiners (see Fig. 3) specific to a chosen root data type can be instantiated from the generic packages `Filter`, `Splitter`, and `Joiner`. These packages are parameterized by the respective input or input and output types.

The generic package for filters is depicted in Fig. 7. AdaStreams filters provide primitive operation `Pop` to retrieve a single data item from a filter's input stream.

```
1  with Root_Data_Type, Base_Filter;
2  generic
3      type In_Type is new Root_Data_Type.Root_Data_Type with private;
4      type Out_Type is new Root_Data_Type.Root_Data_Type with private;
5  package Filter is
6      type Filter is abstract new Base_Filter.Base_Filter with private;
7      procedure Work(F: access Filter) is abstract;
8      procedure Push(F: access Filter; Item: Out_Type);
9      function Pop(F: access Filter) return In_Type;
10     ...
11 private
12     type Filter is abstract new Base_Filter.Base_Filter with record
13         In_Var : aliased In_Type;
14         Out_Var : aliased Out_Type;
15         In_Weight : Positive;          -- # data items Work() pops per invocation
16         Out_Weight : Positive;     -- # data items Work() pushes per invocation
17     end record;
18 end Filter;
```

Fig. 7. Generic package providing the AdaStreams filter type

Likewise, operation `Push` allows a filter to write a data item onto the output stream. Operations `Push` and `Pop` are to be used within a filter's `Work` operation. As already mentioned, by overriding the abstract primitive operation `Work` of a filter, the user implements the actual behavior of the filter. The `Work`-operations 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.

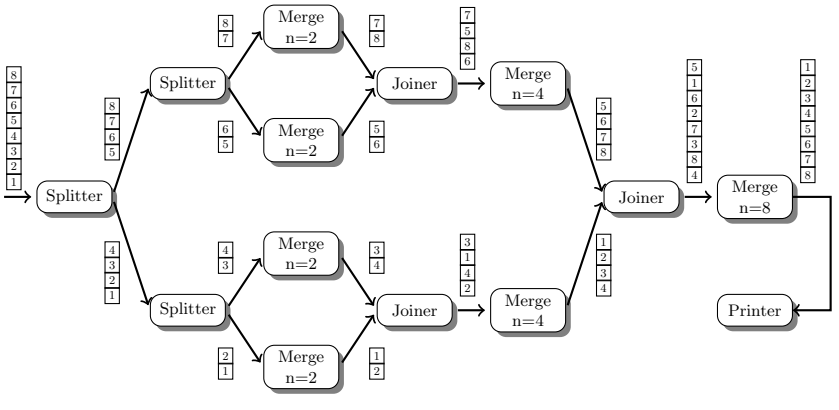


Fig. 8. The merge filters in the merge sort benchmark, with N=8

Fig. 8 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, 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:

1. First the stream data type is declared by extending the `Root_Data_Type`. In our case it is an integer type (see Fig. 9). The implementation of the operations for this type are not shown since they are straight-forward.
2. Next the filters needed for Mergesort are defined by extending the standard filter type. We need a filter for the source of the stream to be sorted. This is filled 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. 10. Note that for space-considerations we had to move the implementations of the above filter's `Work`-operations to the paper's accompanying technical report [9].
3. Procedure Main[3] 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. 8. Runtime arguments of Main are the number of CPUs to use and the number of iterations of the stream graph.

```
1  package Root_Data_Type.Int is

2     type Int is new Root_Data_Type with record
3        I : Integer;
4     end record;

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

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

7  end Root_Data_Type.Int;
```

Fig. 9. Root_Data_Type.Int

```
1  with Root_Data_Type.Int, Base_Filter, Filter, Splitter, Joiner;
2  package UserFilters is
3     package Int_Filter is new Filter (Root_Data_Type.Int.Int, Root_Data_Type.Int.Int);
4     package Int_Splitter is new Splitter (Root_Data_Type.Int.Int);
5     package Int_Joiner is new Joiner (Root_Data_Type.Int.Int);

6     type Merger (aValue : Integer) is new Int_Filter.Filter with record
7        N : Integer := aValue;
8     end record;
9     procedure Work (F : access Merger);
10
11    type Source is new Int_Filter.Filter with null record;
12    procedure Work (F : access Source);

13    type Printer is new Int_Filter.Filter with null record;
14    procedure Work (F : access Printer);
15 end UserFilters;
```

Fig. 10. Mergesort_Filters

[3] Shown in the paper's accompanying technical report [9].

4 The AdaStreams Run-Time System

We implemented the AdaStreams run-time system (RTS) as an Ada package that must be compiled and linked with applications that wish to use the AdaStreams library. Package RTS contains several child packages as shown in Fig 11 and exports only two procedures, as depicted in Fig 12. 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 RTS about connections between actors. Child-component RTS.Stream_Graph maintains the stream-graph topology from calls to `RTS.Connect`.

After the stream graph has been created, the RTS client calls `RTS.Run` to execute the stream graph on `NrCPUs` for `NrIterations`. At this stage RTS executes the following steps:

1. The steady-state for the given stream graph is calculated as outlined in Sec. 2. We use a thin binding to the GiNaC C++ symbolic algebra package [3]. Package RTS.Stream_Graph sets up a system of linear equations that models the input-output behavior of the stream graph. The solution to this equation system denotes the steady state iterations for each actor. For a stream graph that has no steady state RTS raises exception Invalid_Stream_Graph. This is not a limitation of our 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 [17].

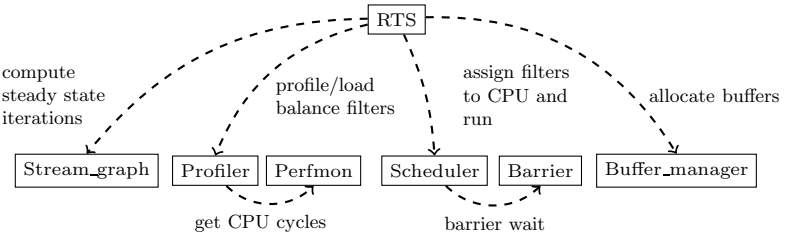


Fig. 11. Component diagram for the AdaStreams run-time system

```
1   with Base_Filter;
2   package RTS is
3      Stream_Type_Error : exception;
4      -- Raised with connections of type-incompatible filters.

5      Invalid_Stream_Graph : exception;
6      -- Raised for a stream that has no steady state.

7      procedure Connect(From : Base_Filter.Base_Filter_Ptr;
8                        To : Base_Filter.Base_Filter_Ptr;
9                        Out_Weight : Positive := 1;
10                       In_Weight : Positive := 1);

11     procedure Run (NrCPUs : Positive; NrIterations : Natural);
12  end RTS;
```

Fig. 12. RTS run-time system package specification

2. Buffers representing the 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.
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. Profiling uses the x86-64's hardware cycle counters exported by the clock library from [6] (again we use a thin binding). 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.
4. For every CPU 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.

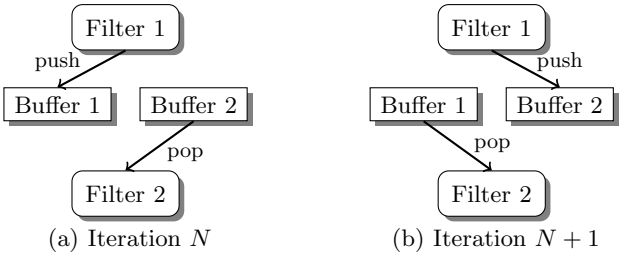


Fig. 13. Double-buffering applied with data channels

There is no need for synchronization of actor execution within a single CPU, because schedulers invoke work functions sequentially. However, across CPUs schedulers need to be synchronized. It is worth noting that we require only a single barrier (implemented as a protected object) for scheduler synchronization. As depicted in Fig. 13, we actually employ two buffers between adjacent actors. Two buffers ensure that both the reader and the writer have their own buffer and need not synchronize with each buffer access. After every steady state iteration schedulers synchronize on the barrier and swap the read and write buffers before the next iteration. Barriers with double buffering reduce synchronization overhead among schedulers and keep the computation-to-communication ratio of stream programs high.

5 Experimental Results

We devised the following Ada benchmark programs to conduct an initial evaluation of the AdaStreams library:

Table 1. Characteristics of benchmark programs implemented with AdaStreams

Benchmark	Filters	Splitters	Joiners
Synthetic	58	1	1
Mergesort	33	15	15
Matrix Multiply	44	5	5
Block Matrix Multiply	31	7	7

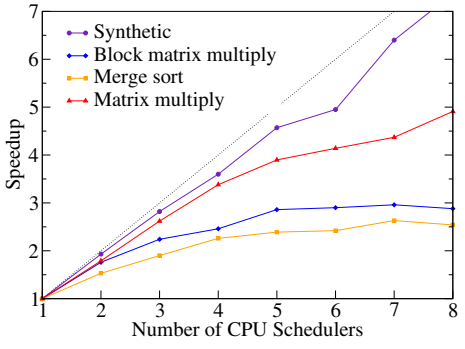


Fig. 14. Scalability of stream benchmark programs

1. Synthetic: this is a synthetic benchmark that uses a busy wait loop to spend CPU cycles. In this benchmark each work function spins for one second before pushing a single data item. Consequently, this benchmark has a very high computation-to-communication ratio.
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 Sec. 3).
3. Block matrix multiply: Block matrix multiply splits each matrix in the stream into blocks and multiplies blocks with small communication overhead. Blocks are added and combined.
4. Matrix multiply: multiplies two square matrices. It transposes one of them and multiplies two matrices in parallel.

Table 1 shows the characteristic features of our benchmark stream programs. All benchmarks were compiled with the 64-bit version of GNAT GPL 2009 (20090511). To determine the scalability of the AdaStreams implementation with respect to the number of CPU cores, we executed all benchmarks on an Intel x86-64 server with two Xeon 5120 quadcore CPUs. As expected, the synthetic benchmark with its high workload scaled best. Matrix multiply also scaled very well, with a speedup of a factor of almost 5 with 8 CPU cores. Block matrix multiply achieved a speedup of almost three times with more than four cores. The work functions of Mergesort show very fine-grained parallelism; under those circumstances scalability was reasonable. It should be noted that scalability of stream programs was achieved without changing even a single line of source

code—mapping of actors to different numbers of CPU cores was all transparently handled by the AdaStreams library.

6 Conclusions and Future Work

In this paper we have proposed a type-based stream programming extension to Ada 2005. AdaStreams is a type-hierarchy for actor-specification together with a run-time system that supports the execution of stream programs on multicore architectures. AdaStreams is non-intrusive in the sense that no change of an Ada 2005 programming language implementation is 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, AdaStreams allows dynamic creation of stream programs. Messaging (known from StreamIt) can be done via standard Ada features such as protected objects. The whole spectrum of data types available in Ada can be used for streaming. Each AdaStreams filter has an input and output type. That way filters are allowed to convert data, which allows the generation of heterogeneous stream graphs. Splitters and joiners are restricted to a single type.

We have implemented AdaStreams for Intel multicore architectures. We provide initial experimental results that show the effectiveness of our approach on an Intel X86-64 quadcore processor. The initial release of our work is available for download at [1].

Our plan with AdaStreams is to survey the stream graph patterns arising from real-world applications and build higher-level stream graph constructs from commonly occurring patterns. We will investigate improvements to our greedy actor allocation algorithm.

References

1. AdaStreams Web Site, `http://elc.yonsei.ac.kr`
2. Andrews, J., Baker, N.: Xbox 360 System Architecture. IEEE Micro 26(2), 25–37 (2006)
3. Bauer, C., Frink, A., Kreckel, R.: Introduction to the GiNaC Framework for Symbolic Computation within the C++ Programming Language. J. Symb. Comput. 33(1), 1–12 (2002)
4. Belina, F., Hogrefe, D.: The CCITT-Specification and Description Language SDL. Computer Networks 16, 311–341 (1989)
5. Berry, G., Gonthier, G.: The Esterel Synchronous Programming Language: Design, Semantics, Implementation. Sci. Comput. Program. 19(2), 87–152 (1992)
6. Bryant, R.E., O'Halloran, D.R.: Computer Systems: A Programmer's Perspective. Prentice-Hall, Englewood Cliffs (2003)
7. Buttlar, D., Farrell, J., Nichols, B.: PThreads Programming. O'Reilly, Sebastopol (1996)
8. Caspi, P., Pilaud, D., Halbwachs, N., Plaice, J.: Lustre: A Declarative Language for Programming Synchronous Systems. In: POPL 1987, pp. 178–188 (1987)

9. Hong, G., Hong, K., Burgstaller, B., Blieberger, J.: AdaStreams: A Type-based Programming Extension for Stream-Parallelism with Ada 2005. Technical Report TR-0003, ELC Lab, Dept. of Computer Science, Yonsei University, Seoul, Korea (March 2010), http://elc.yonsei.ac.kr
10. Google. The Go Programming Language Specification, http://golang.org (retrieved November 2009)
11. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann, San Francisco (2008)
12. IBM Redbooks. Programming the Cell Broadband Engine Architecture: Examples and Best Practices (August 2008), http://www.redbooks.ibm.com
13. IDC. PC Semiconductor Market Briefing: Re-Architecting the PC and the Migration of Value (June 2008), http://www.idc.com
14. ISO/IEC 8652:2007. Ada Reference Manual, 3rd edition (2006)
15. Kahn, G.: The Semantics of a Simple Language for Parallel Programming. In: Rosenfeld, J.L. (ed.) Information Processing, Stockholm, Sweden, August 1974, pp. 471–475. North Holland, Amsterdam (1974)
16. Karczmarek, M., Thies, W., Amarasinghe, S.: Phased Scheduling of Stream Programs. SIGPLAN Not. 38(7), 103–112 (2003)
17. Lee, E.A., Messerschmitt, D.G.: Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. IEEE Trans. Comput. 36(1), 24–35 (1987)
18. Leung, M.-K., Liu, I., Zou, J.: Code Generation for Process Network Models onto Parallel Architectures. Technical Report UCB/EECS-2008-139, EECS Department, University of California, Berkeley (October 2008)
19. Lin, C., Snyder, L.: Principles of Parallel Programming. Addison Wesley, Reading (2008)
20. Mattson, T.G., Sanders, B.A., Massingill, B.L.: Patterns for Parallel Programming. Addison Wesley, Reading (2007) (3rd printing)
21. Reinders, J.: Intel Threading Building Blocks. O'Reilly, Sebastopol (2007)
22. Stephens, R.: A Survey of Stream Processing. Acta Informatica 34, 491–541 (1997)
23. Thies, W.: Language and Compiler Support for Stream Programs. PhD thesis, Massachusetts Institute of Technology (February 2009)
24. Thies, W., Karczmarek, M., Amarasinghe, S.P.: StreamIt: A Language for Streaming Applications. In: Horspool, R.N. (ed.) CC 2002. LNCS, vol. 2304, pp. 179–196. Springer, Heidelberg (2002)