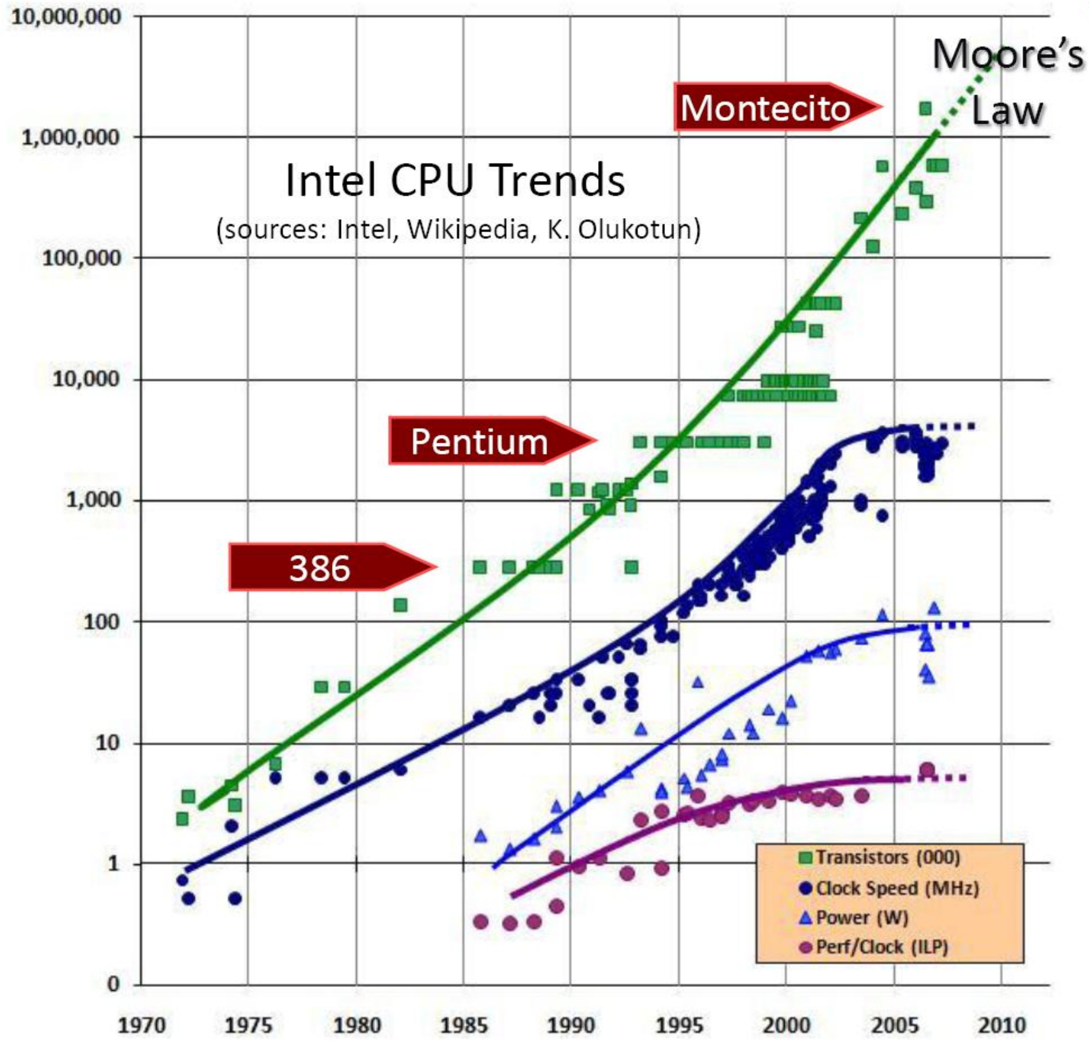
AdaStreams : A Type-based Programming Extension for Stream-Parallelism with Ada 2005

Gingun Hong*, Kirak Hong*, Bernd Burgstaller* and Johan Blieberger◇

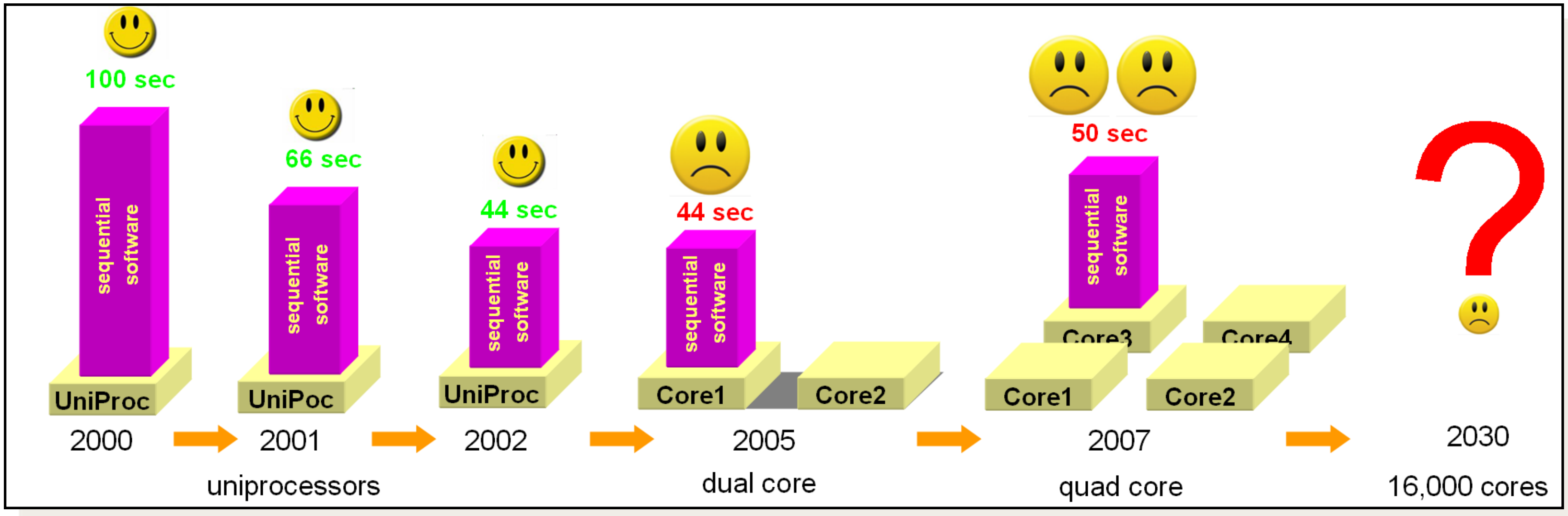
*Yonsei University, Korea
◇Vienna University of Technology, Austria

The ‘free performance lunch’ is over



- By Moore’s Law, the number of transistors in CPU is still increasing
- Since 2000, Clock speed stopped going up
- Now: deliver more cores per chip (multicores, GPUs)
- “Every year we get ~~faster~~ **more** processors.”

The Fate of Sequential Programs...



- A sequential program is restricted to a single core.
  - Performance might even decrease on future multi-core architectures because of lower Perf/Clock ratio.
  - No more performance gains in foreseeable future for sequential programs on multicore architectures.

Programmers are Challenged...

With thread-and-lock based programs:

- race-conditions
- deadlocks
- starvation
- non-composeability of software

Hardware is back on the programmer's horizon:

- **performance bugs**
- Scalability problems
- Performance portability
- without knowing the underlying hw, it's impossible to write efficient parallel programs

Processor Architectures

Uniprocessors:

Common Properties
Single flow of control
Single memory image

Differences
Register file
Instruction set architecture
Functional units

Von-Neuman languages represent common properties and abstract away differences.

Multicores:

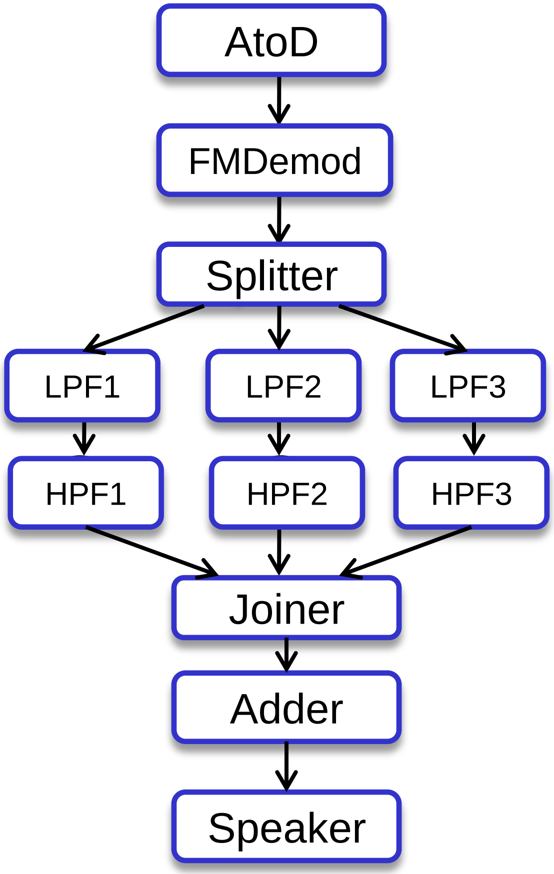
Common Properties
Multiple flows of control
Multiple local memories, e.g., Cell BE

Differences
Number and capabilities of cores
Communication model
Synchronization model

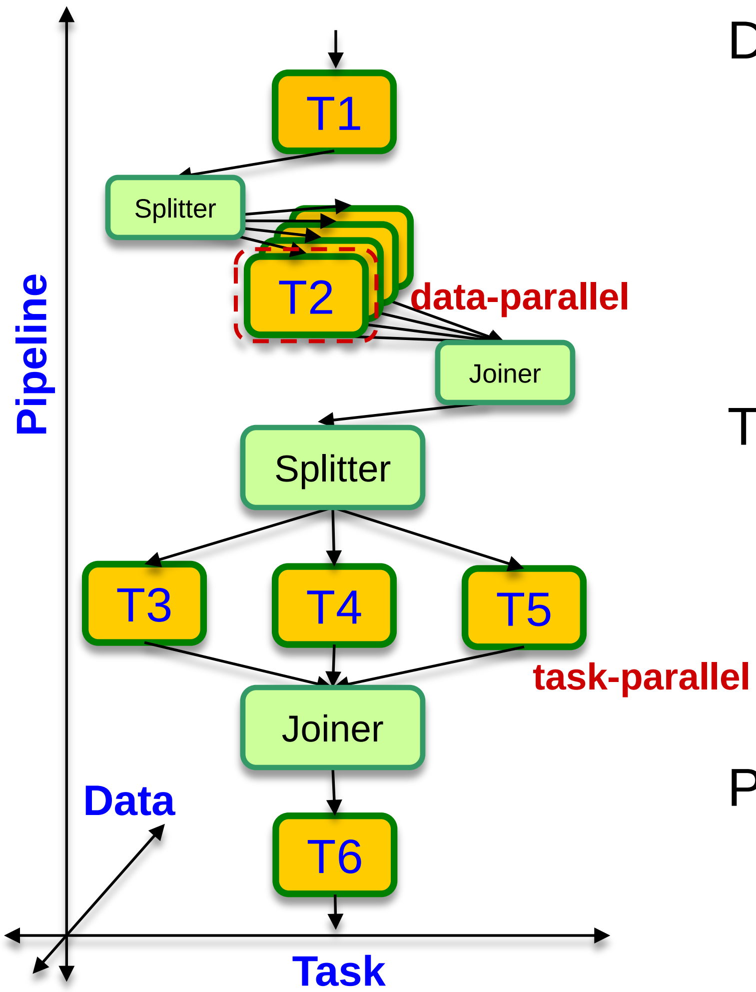
Need a common programming paradigm for multicore architectures.

Streaming as a New Programming Paradigm

- For programs based on streams of data
  - Audio, video, DSP, networking, cryptographic processing kernels
  - Examples: HDTV editing, radar tracking, cell phone base stations, computer graphics
- Properties of streams:
  - Independent filters (aka ‘actors’) communicating via data-channels
  - Regular and repeating computation & communication
  - Task, data, and pipeline parallelism expressible



Task+Data+Pipeline Parallelism



Data Parallelism

- Same operation on different data items
- Placed within splitter/joiner pair (*fission*)
  - e.g., 4 x T2

Task Parallelism

- Between filters *without* producer/consumer relationship
  - e.g., T3, T4, T5

Pipeline Parallelism

- Between producers and consumers
  - e.g., T1, T2, …

AdaStreams

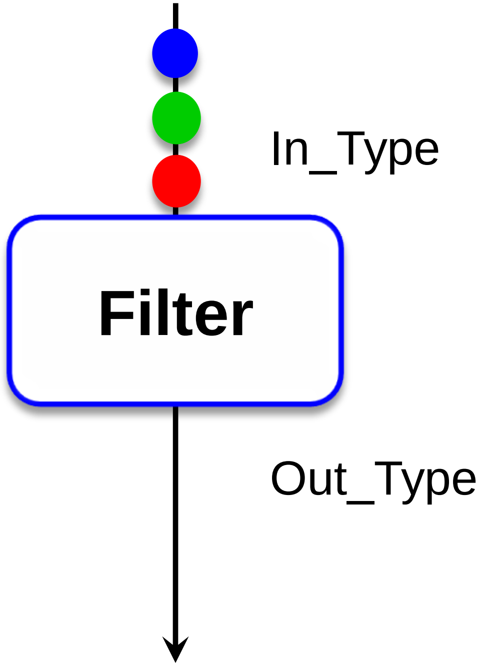
- Programming library in Ada 2005
  - Adds stream programming functionality to Ada
  - Existing Ada code is reusable
  - Lowers entry barrier to stream programming

- How to use AdaStreams:
  1) User defines actors by extending provided type-hierarchy
     - Three basic actor types : filters, splitters and joiners
     - User specifies how actors will work
  2) User connects actors to build stream graph
  3) User starts execution

  ► Runtime system manages efficient execution on multi-core hardware

Defining actors

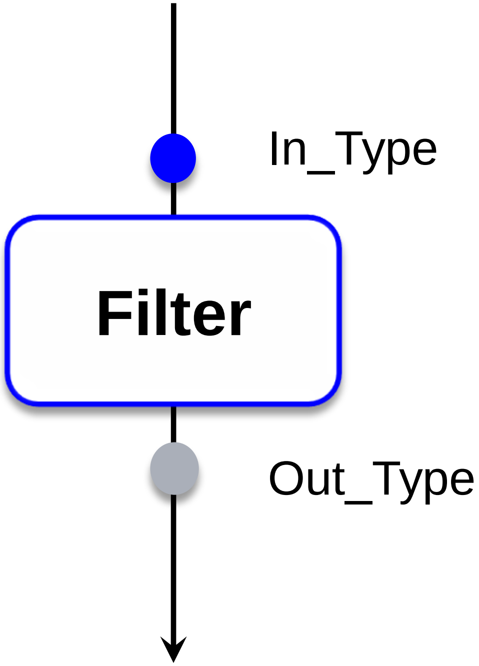
- Filter as a basic unit of computation
  - Tagged type with AdaStreams
  - Designated input and output type
  - User defines filter's Work() function




```
Procedure Work (f:access Filter) Is
   Item : In_Type;
   Ret  : Out_Type;
Begin
   F.Pop(Item);
   F.Pop(Item);
   Do_Something(Item, Ret)
   F.Push(Ret);
End Work;
```

Defining actors

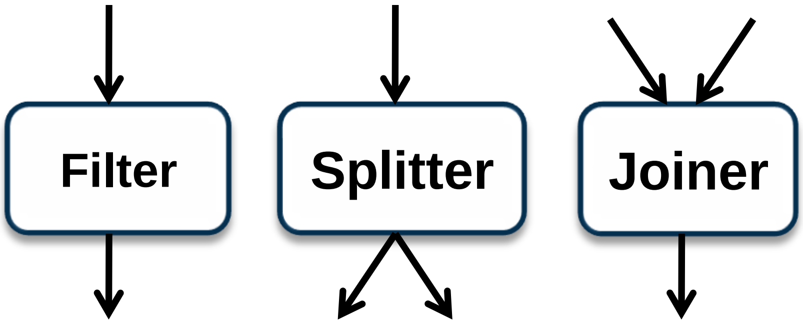
- Filter as a basic unit of computation
  - Tagged type with AdaStreams
  - Designated input and output type
  - User defines filter's Work() function

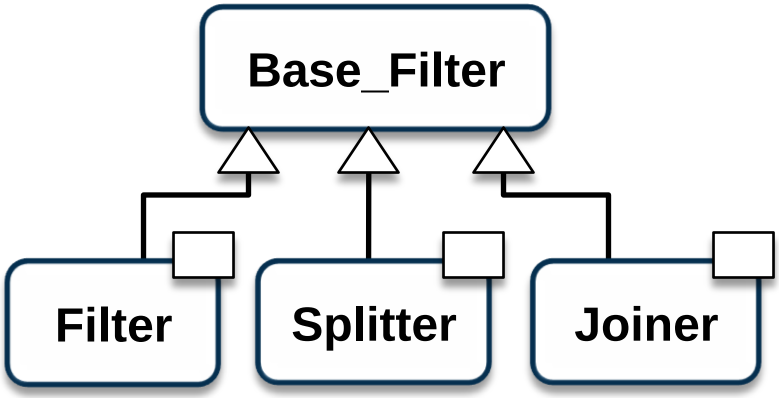



```
Procedure Work (f:access Filter) Is
   Item : In_Type;
   Ret  : Out_Type;
Begin
   F.Pop(Item);
   F.Pop(Item);
   Do_Something(Item, Ret)
   F.Push(Ret);
End Work;
```

Defining actors

- All actors extend tagged type Base_Filter
- Splitters and Joiners
  - Have no computations, just data transfers
  - Enable data and task parallelism



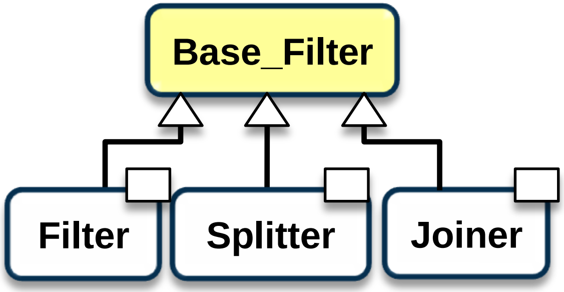


Actor class hierarchy

Actor Root Type: Base_Filter

```
package Base_Filter is

   type Base_Filter is abstract tagged private;

   procedure Work (f: access Base_Filter) is abstract;

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

private
   type Base_Filter is abstract tagged null record;
end Base_Filter;
```

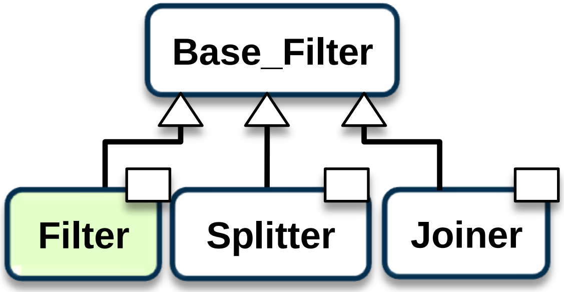
- Base_Filter is parent of all actor types
- Actors override Base_Filter's primitive operations
  - Work()
  - Connect()



Generic Filter Package

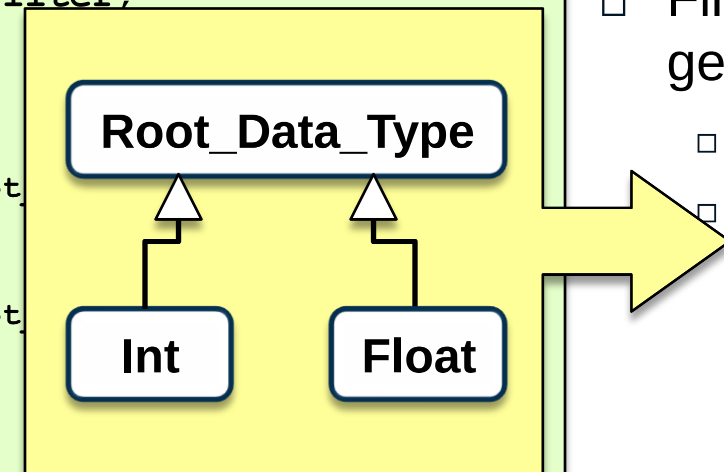
```
with Root_Data_Type, Base Filter;
generic
   type In_Type is
        new Root_Data_Type.Root_Data_Type with private;
   type Out_Type is
        new Root_Data_Type.Root_Data_Type with private;
```

- Filter type depends on generic types
  - In_Type, Out_Type
  - User-defined extension of Root_Data_Type

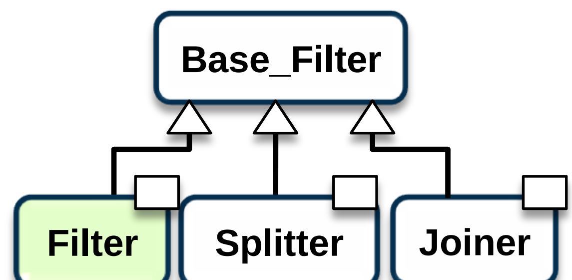


Generic Filter Package

```
with Root_Data_Type, Base_Filter;
generic
   type In_Type is
        new Root_Data_Type.Root
   type Out_Type is
        new Root_Data_Type.Root
```



- Filter type depends on generic types
  - In_Type, Out_Type
  - User-defined extension of Root_Data_Type

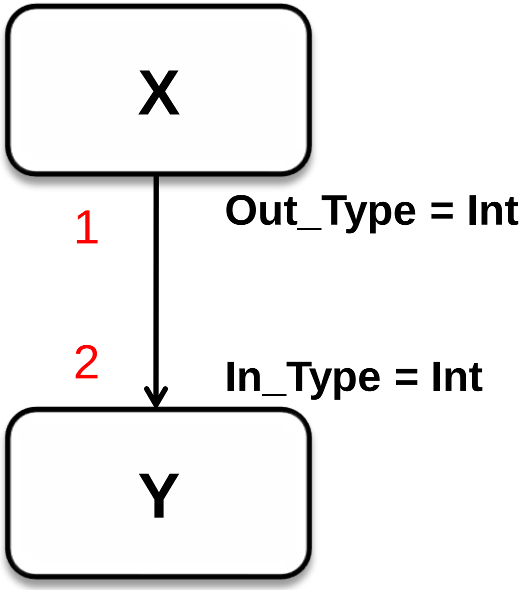


Generic Filter Package

```
with Root_Data_Type, Base Filter;
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 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
…
end Filter;
```

- Filter type depends on generic types
  - In_Type, Out_Type
  - User-defined extension of Root_Data_Type
- Work() procedure is abstract
  - User defines Work() procedure
  - Push() writes data to output data channel
  - Pop reads data from input data channel

Stream Graph Construction



- Connect() operation attaches downstream actor:

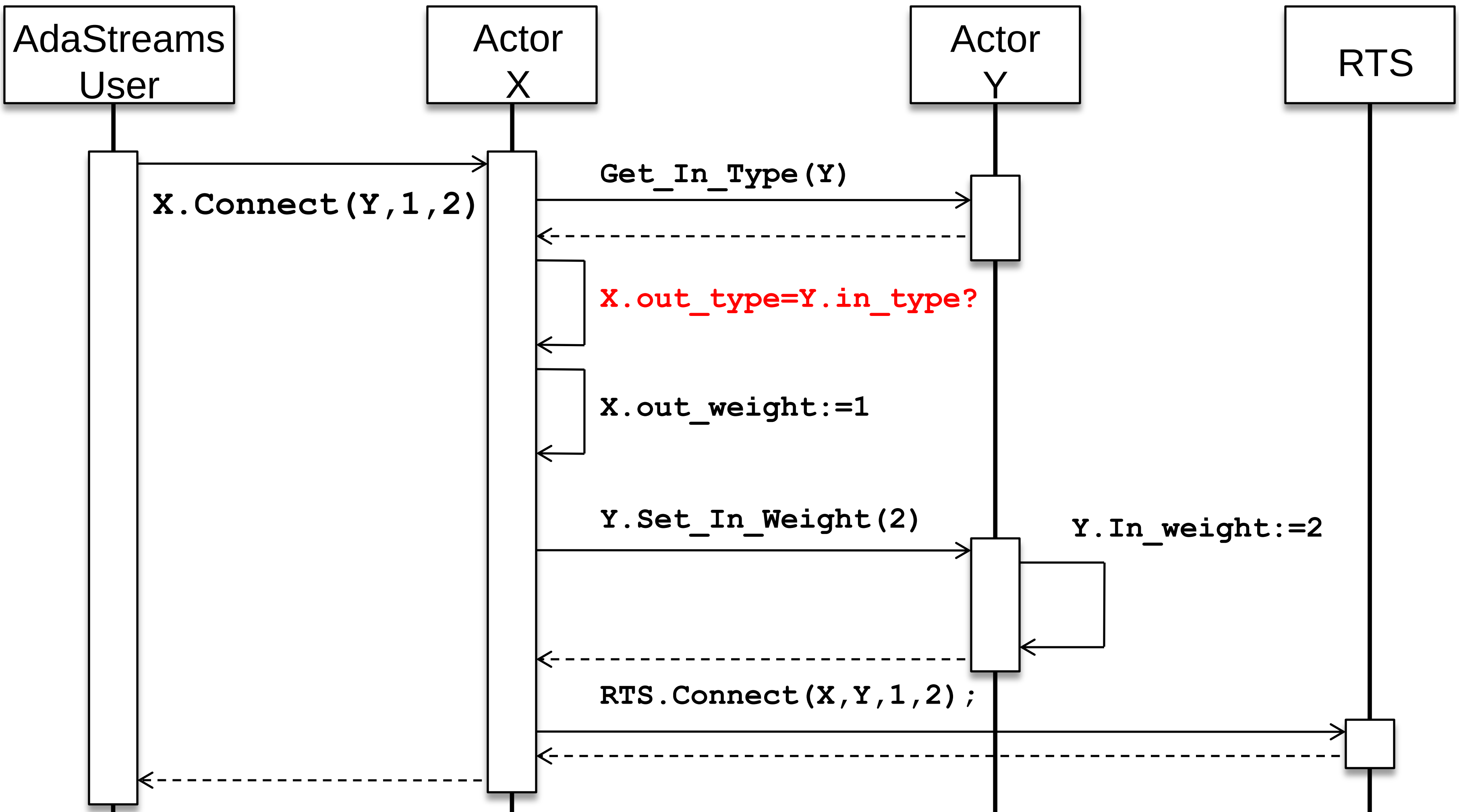
```
X.Connect (Y, 1, 2);
```

  - Arguments:
    - downstream actor (**Y**)
    - # items produced by source (**1**)
    - # items consumed by sink (**2**)

- Run-time type check:
  - prevents type-clash of connected actors

- Call to run-time system (RTS):
  - to build stream graph representation

Stream Graph Construction



Executing stream programs

- Run-time system (RTS) manages execution
  - Initiated by RTS.Run()
  - Maps stream-graph onto # available cores
  - Executes periodic schedule # iterations times

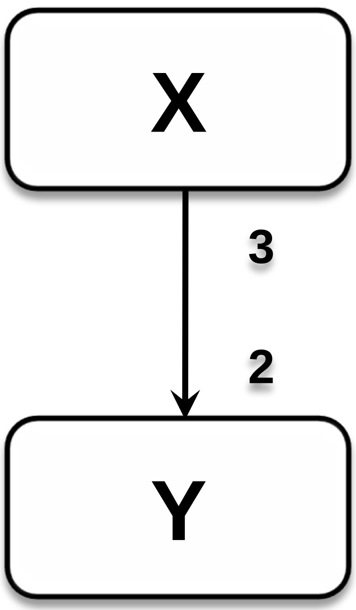
```
Package RTS is
   Stream_Type_Error : exception
   --Raised with connections of type-incompatible actors

   procedure Connect(…);

   procedure Run(NrCPUs : Positive;
                 NrIterations : Natural);
End RTS;
```

Run-time system support

1) Determine a periodic schedule for stream graph execution
2) Allocate data channels between actors
3) Profile actors
4) Load balance actors among available cores

Compute Periodic Schedule

- Periodic schedule is a finite schedule of actors
  - Invokes each actor at least once
  - Produces no net change in amount of buffered data
  - That is, the number of tokens on each edge is the same before/after schedule execution

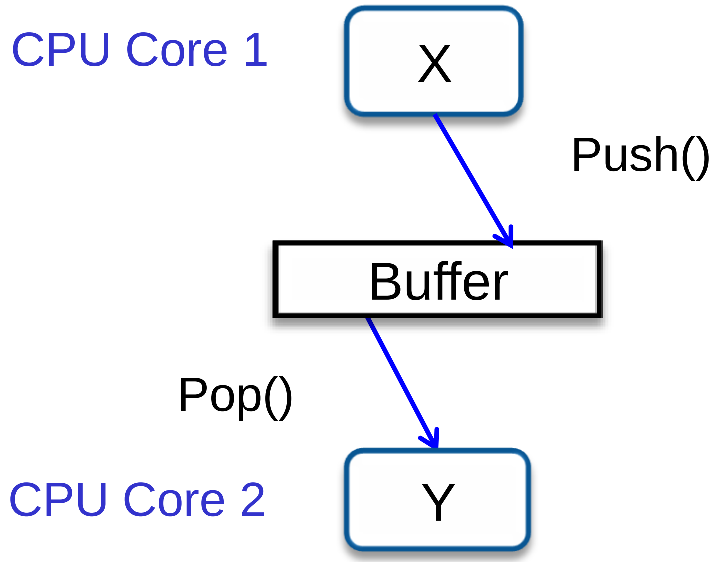


X produces 3 items
Y consumes 2 items

XX YYY is a periodic schedule

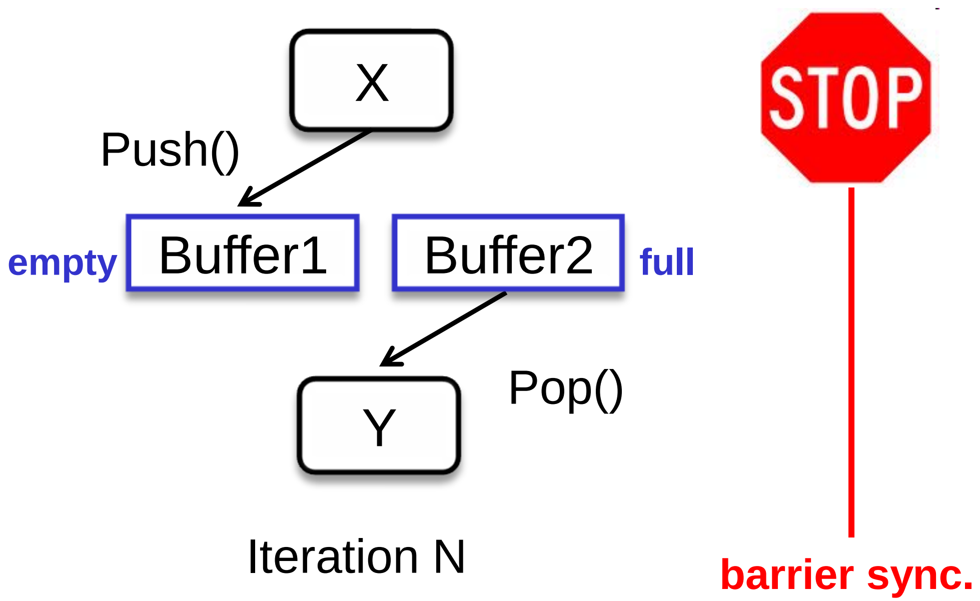
Buffer Communication

- Concurrent actor execution requires buffer synchronization
- Synchronization limits parallelism
  - producer/consumer synchronize once per buffer access!
  - Cache-coherence causes additional slow-down!



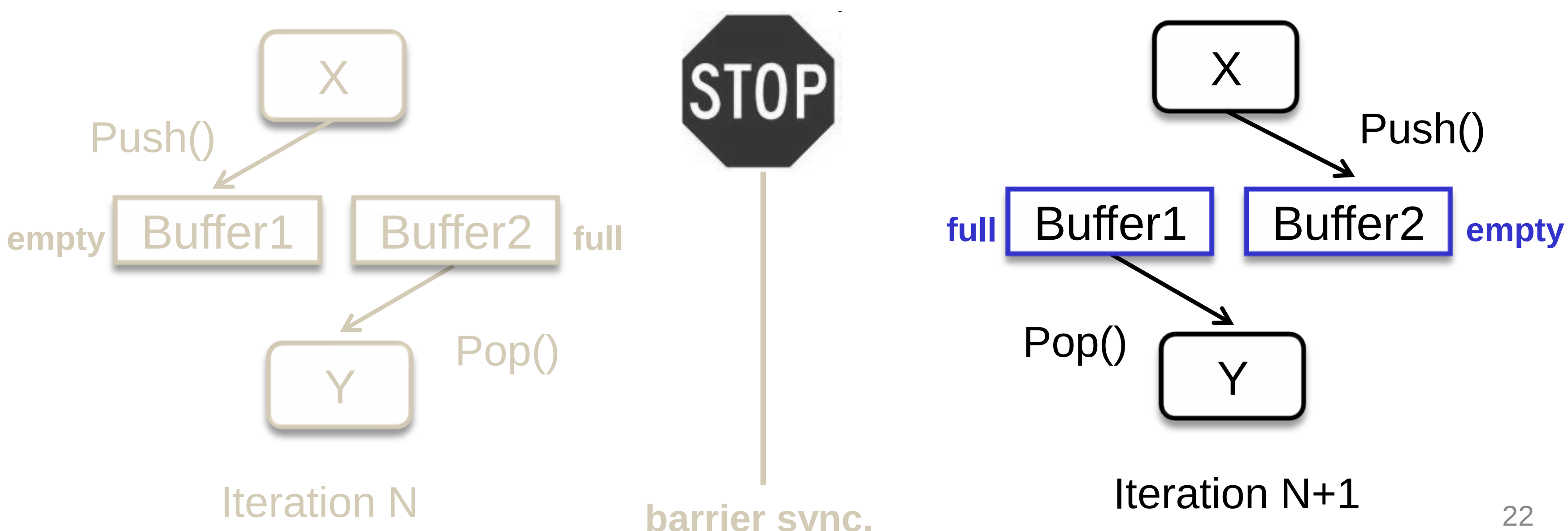
Double Buffering

- Empty buffer
  - Filled by upstream actor's Work() function
- Full buffer
  - Drained by downstream actor's Work() function
- All actors synchronize only **once** at barrier before next iteration.



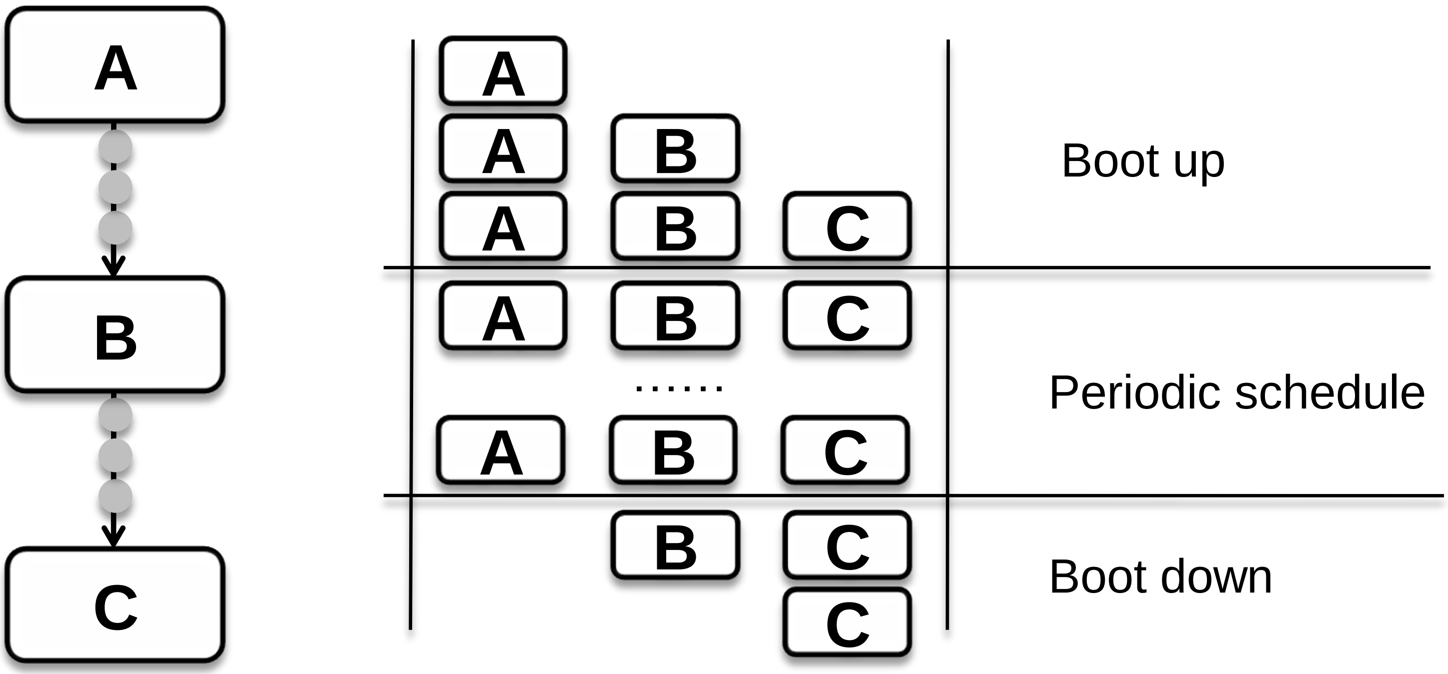
Double Buffering

- Empty buffer
  - Filled by upstream actor's Work() function
- Full buffer
  - Drained by downstream actor's Work() function
- All actors synchronize only **once** at barrier before next iteration.



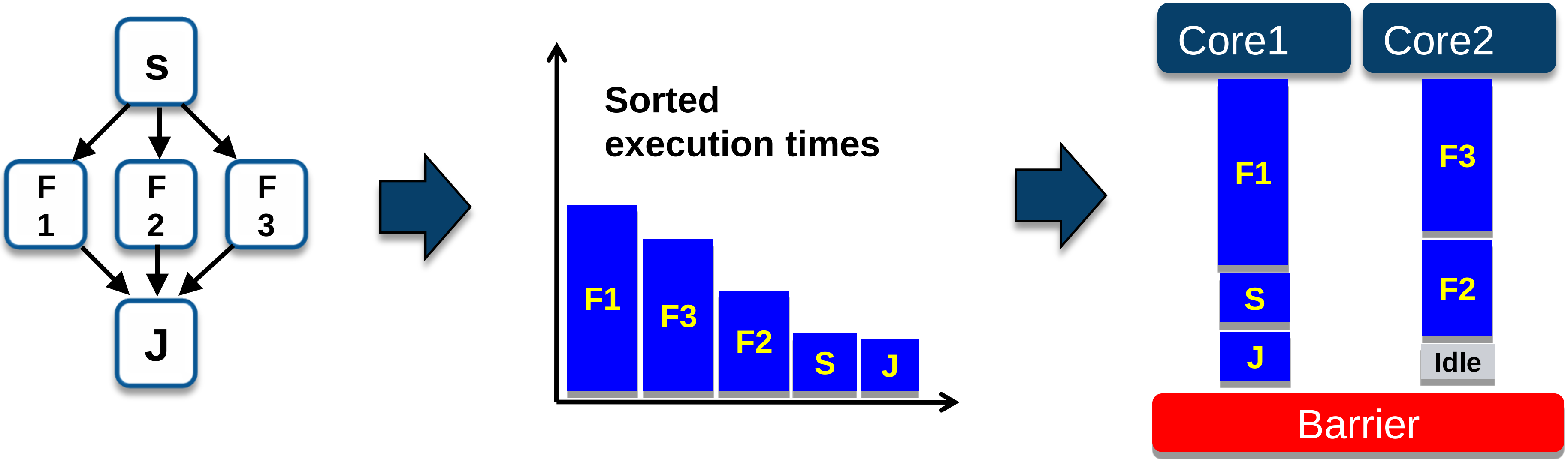
Profiling

- Find out CPU cycles that actor spends in its Work() procedure
  - Done during execution because of actors' side effects
  - Profiler counts CPU cycles in the booting phase

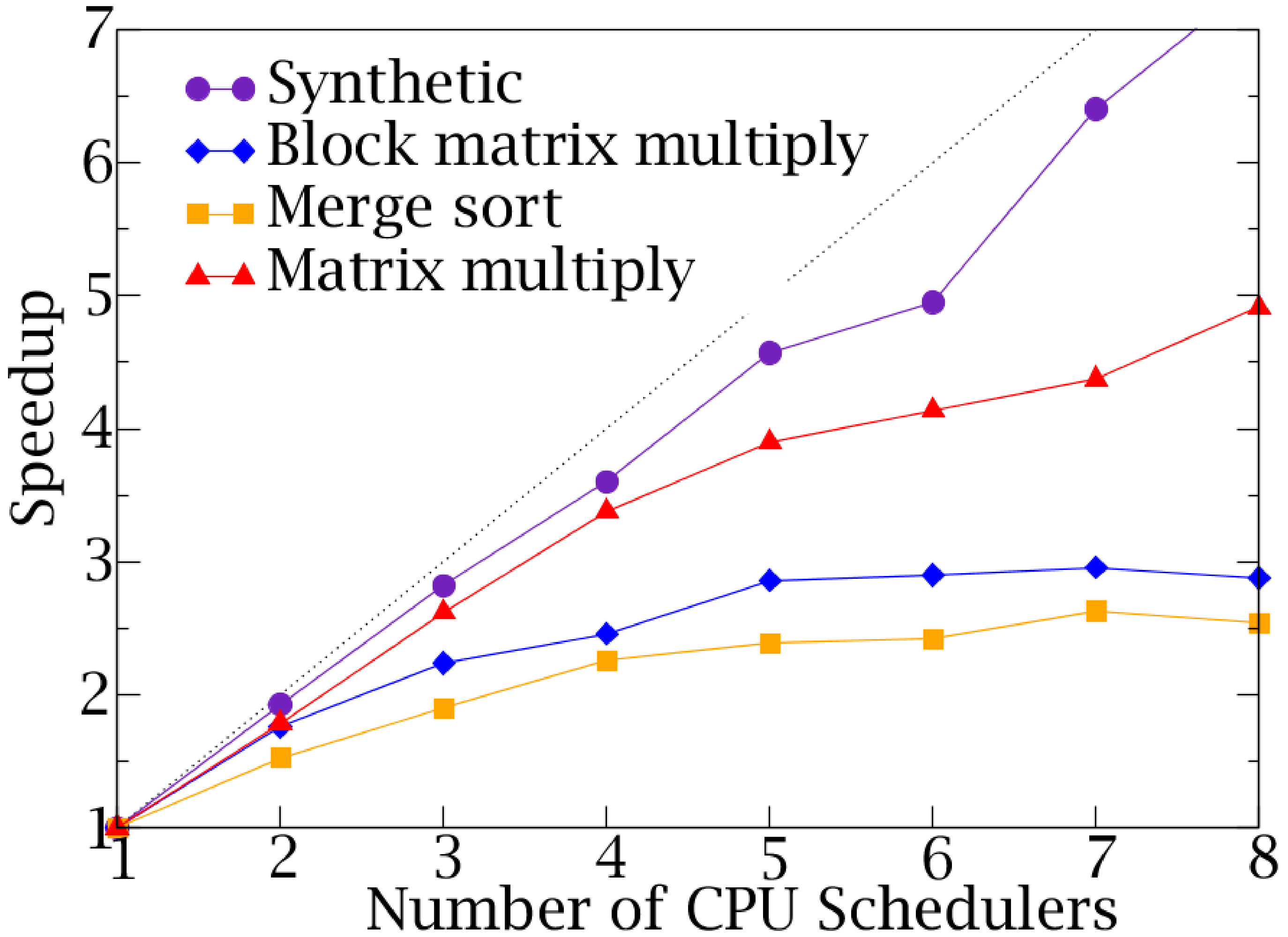


Actor-to-CPU Assignment

- Load-balance actors among CPUs
  - Multiple Knapsack problem, NP-complete
  - Greedy approximation algorithm used
    - Actors sorted by execution time from largest to smallest
    - Assigned to CPU cores based on accumulated load.
- Execute program with the number of iterations



Benchmark Results



Conclusions

- Add stream programming functionality to Ada2005
  - Lowers entry barrier to stream programming
  - Existing Ada code is reusable
  - Abstracts away underlying parallel hardware
- Runtime system supports efficient program execution
  - Computes periodic schedules
  - Profiles and load-balances actors
- Unlike previous approaches
  - stream-graphs can be constructed at run-time
- Compute-intensive applications show best speedups.

Q&A

Thank you

☺

AdaStreams sources are available at

http://elc.yonsei.ac.kr/AdaStreams.htm