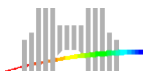
Automatic Parallelization for the Next Ten Years

Paul Feautrier

ENS de Lyon
Paul.Feautrier@ens-lyon.fr

June 17, 2012



The Context, I

Revolutionary Technologies are still more than 10 years in the future.

- quantum and one-electron transistors
- molecular, nano-tubes
- organic, DNA
- optical

Exception Non volatile mass memory devices.

The Context, II

The clock frequency no longer increases beyond about 3 Ghz.

- The electric power consumption and heat generation is proportional to frequency
- which is constrained by the amount of heat that can be dissipated.

It may even be lowered to around 1 Ghz to reduce electric power consumption.

The Context, III

The device density still increases, but at a lower rate:

- Doubling every 2-3 years
- The delay is expected to increase slowly
- and stop sometime in the next decade.

Hence the prevalence of multicores. Two approaches:

- A small number (≈ 10) of powerfull processors
- A much larger number (100 to 1000) of small processors

The Context, IV

The memory wall is still with us, only more so.

- The technology is not keeping pace (latency, bandwidth, pin count)
- Each core has a minimum bandwith requirement. Adding cores when the memory subsystem is saturated is useless.

Present and future architectures features deep memory hierarchies and rely on locality for performance.

The Context, V

Electric power consumption will be a major concern.

- Mobile devices
- Today HP computers draw around 1 MW.
- Extrapolating present technology toward exascale (10^{18} flops) requires GWs of power
- Admissible limit: 25 MW.

Something must be done. Observe that parallel computers are less power hungry than sequential ones.

Consequences, I

Very high degree of parallelism:

- around 10 for PC,
- around 1000 for embedded devices
- more than 10^9 for HPC.

Consequences, II

Deep memory hierarchy

- Several cache levels, scratchpads, not always hardware coherent (coherence does not scale).
- Dual Data Rate memories,
- Global memory, probably distributed (scalability again)
- Non volatile memory, ...

Served by several levels of interconnects, from shared registers, NoC to WLAN.
Shared memory for each *cluster* (a few tenth of processors), distributed memory beyond.

Other Problems

Not adressed here.

- Reliability
- Heterogeneity
- Debugging
- Scalability

The Parallel Programming Problem

Hand-crafting up to several billion threads is not practical.
The parallel programmer needs to take much more decisions than the sequential programmer:

- Distribute the workload among the processors
- Distribute the data among the memory subsystems
- Arrange for cooperation between processes

Parallel systems are inherently non deterministics, which make debugging very difficult.

Solutions?

Solutions, I

Leave parallel programming to specialists, which use low level parallel programming languages and libraries.

- Game and special effects designers
- DBMS creators
- The Joes and Stephanies approach

Not suitable for research and development.

Solutions, II

Parallel Programming Languages and Libraries

- Some examples: OpenMP, MPI, HPF.
- New Languages: Chapel, Parallex, X10
- Very high level languages: Concurrent Haskell

Problem: how much parallelism to hide? Efficiency / portability tradeoff.

Solution, III

Autoparallelization would be the best solution (ease of use, determinism, portability, efficency) but:

- works well only for highly regular programs (the polyhedral model)
- works well only for fine grain paralellism (vector or SIMD processors)

Solution, IV

There have been attempts to combine the two approaches:

- Hints to the compiler, Cray Fortran Translator pragmas, OpenMP
- Transformation scripts (Nicolas Vasilache's WrapIt, AlphaZ)

Problem: Human Computer Interaction: most users (physicists, biologists) do not understand the language of parallel compilers!

Solution, V

Distribute the work according to each partner expertise:
Human are good at finding coarse-grain parallelism:

- The real world is parallel, and most HPC programs are simulations or emulations
- Consider the success of MPI over OpenMP

Compilers are good at fine grain parallelism:

- Instruction Level Paralellism
- Software pipelining
- Vectorization
- Polyhedral programs

Example I: the Dependence Calculation

The most demanding part of a parallelizing compiler is the dependence calculation (quadratic in the size of the program). It would be nice to be able to parallelize it.

- It is obvious that each dependence test is independent of all others, but this may not be evident from the code. Ask the programmer.
- Each test is an LP program (a variation on Gaussian elimination) which is almost polyhedral (selection of the pivot). A case for autoparallelization.

Example II: Dataflow

A program in which no control is specified: an operation is "fired" as soon as its operands are available.

- The "grain" (size of an atomic operation) is a design choice in the model (made by the programmer):
  - for `pmake`, compilation of one file,
  - for a dataflow processor, one machine instruction.
- Parallelism: several operations may be ready to fire at the same time and can execute in parallel (resources permitting).
- Some examples:
  - Alpha
  - Lucid, Sisal
  - Synchronous languages such as Lustre and Signal.

Streaming

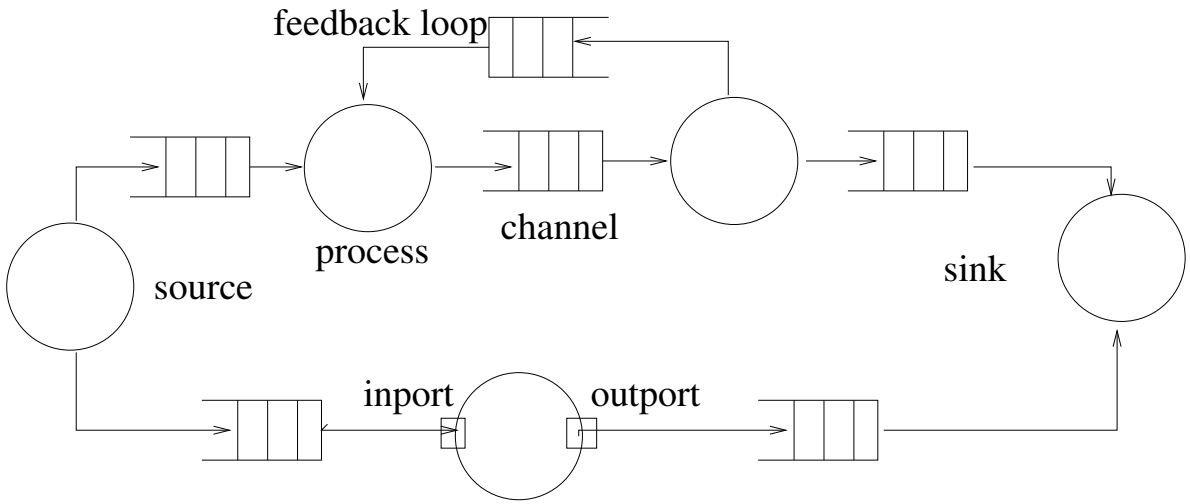
A streaming program is simply a program with an outermost infinite loop, consuming and producing data (i.e., streams).

- Depending on the model, this loop can be:
  - implicit: KPN, SDF, Lustre, Signal, StreamIt, Faust
  - explicit: CRP (see later), Stream-OpenMP, Alpha

Streaming Dataflow Programs

A dataflow program in which the set of operations is infinite.

- Especially adapted to signal processing.
- Usually presented as a network of processes (or filters) interconnected by channels. Processes consume data (tokens) from their input ports and create tokens on their output ports.



Communication

Communication style:

- Channels: sends & receives via a communication medium.
  - FIFO (e.g., KPN).
  - Sliding window (e.g., StreamIt, Stream-OpenMP).
- Shared memory: read & write via addressable memory regions.

Questions

Some properties that are obvious for sequential programs are conjectural for process networks:

- Determinism: does the network always give the same result, or always have the same behaviour? Sub-question: what is a result, or a behaviour?
- Deadlocks: does the network stall for ever?
- Does or can the network run in bounded memory?
- Synchronization/communication: how to implement channels?
- Optimization: can the grain be changed? Can the degree of parallelism be changed?

All these problems may be solved by the compiler, or left to the programmer, or solved – at least partially – by a library.

Kahn Process Networks (KPN)

- Channels: unbounded FIFOs, with one writer & one reader.
- Process body: sequential program augmented with `send` and `receive` statements. There is *no* `select` statement.
- KPNs are deterministic by construction provided that the process bodies are.
- KPNs may have deadlocks. Testing for deadlocks?
- It is in general impossible to bound the size of the channels, and doing so may introduce new deadlocks.
- KPNs have been implemented mostly as libraries.

Communicating Regular Processes (CRP)

The problems with KPN:

- For KPN analysis, one has to pair a send and a receive: the send must be executed earlier than the corresponding receive.
- This is done by counting, but if the send or the receive is in a multidimensional loop nest, the count is a polynomial.

The CRP approach:

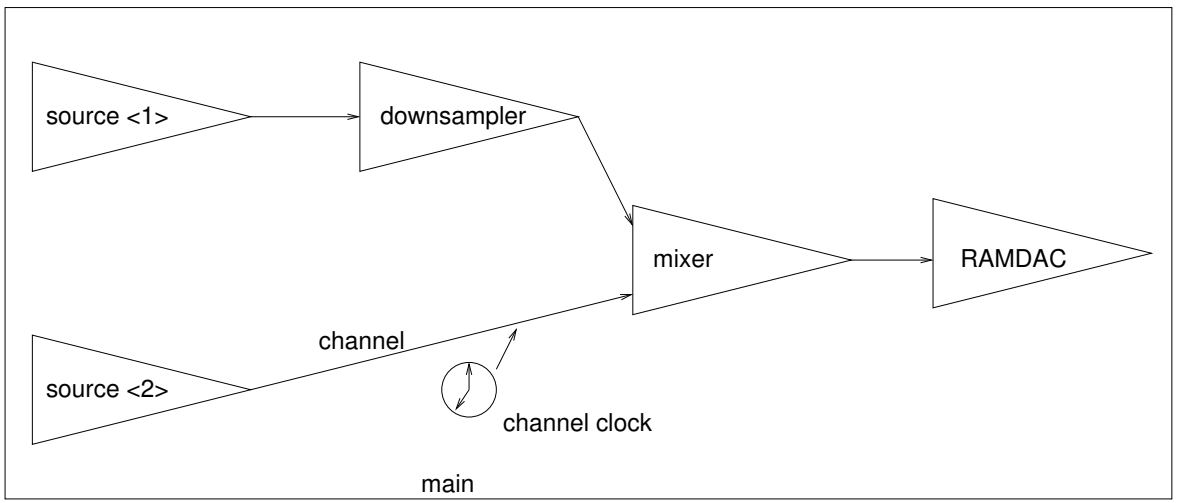
- Instead of `send` and `receive`, write or read to an unbounded array of the same dimension as the loop nest.
- It is the responsibility of the compiler to shrink each array as much as possible.
- Determinism guaranteed when in write once / read many mode.

Answers

- Analysis: static analysis and static scheduling are possible.
- Determinism: CRP are deterministic if the Single Assignment property holds.
- Deadlocks: if scheduling succeeds, no deadlock in the network.
- Buffer size:
  - A partial solution: increase the size until deadlocks disappear.
  - Incidentally, resizing buffers adjusts the degree of parallelism.
  - Use array contraction and code transformations (e.g., tiling).

Modular Scheduling

- Introduce channel clocks.
- Schedule `source`, `downsampler`, `mixer`, `RAMDAC` independently, with the channel clocks as parameters.
- Schedule `main` (i.e., compute the channel clocks).
- Substitute solution into the process schedules.
- One can impose size constraints on the channels.



Evaluation, Pros

CRP has a compiler, a scheduler, and a rudimentary code generator for shared memory (i.e. Pthreads). It provides:

- Modularity and reuse, both at design time and at compile time.
- Scalability
- Determinism
- Safety: deadlocks can be detected without running the target program.

Evaluation, Cons

The present version is entirely static: it is not possible to extend the process network at run time.

The handling of bulky data (e.g. a whole TV frame) is inefficient.

Some compile time checks are not implemented:

- The only one writer rule
- Reading an undefined channel cell

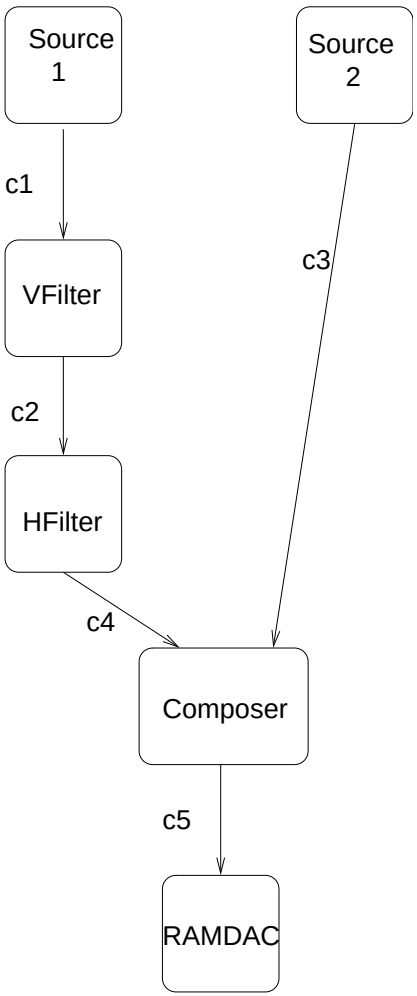
Future Work

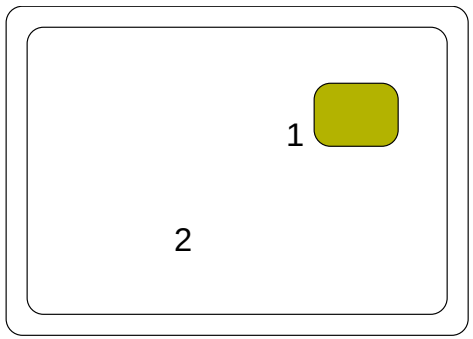
- Write one (or more) code generator
- Enlarge the language model
  - Conditionals and `while` loops
  - Dynamic creation of processes and channels, recursion.
- The user may have other interesting knowledge:
  - reductions and scans
  - convergence and numerical stability
  - don't cares

Questions

QUESTIONS ?

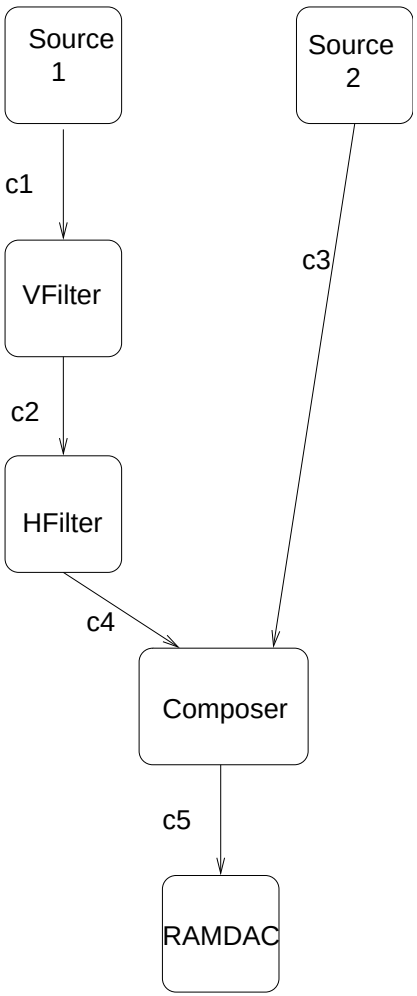
A Video Example, I





- Picture-in-picture.
- Two video sources.
- Source 1 is scaled down.
- Composer: for each screen pixel, select source 1 or 2.
- RAMDAC: paint the screen.

A Video Example, II

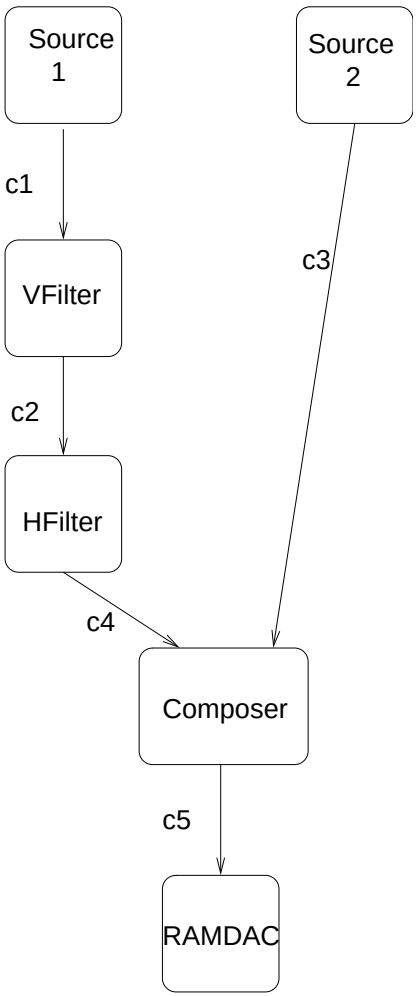


The HFilter

```
struct bigLine {
  char pixel[960];
}
struct smallLine {
  char pixel[120];
}
process HFilter(
    inport struct bigLine x[],
    outport struct smallLine y[]) {
 int i, j;

 for(i=0;;i++)
   for(j=0; j<120; j++)
     y[i].pixel[j] = x[i].pixel[8*j];
}
```

A Video Example, III

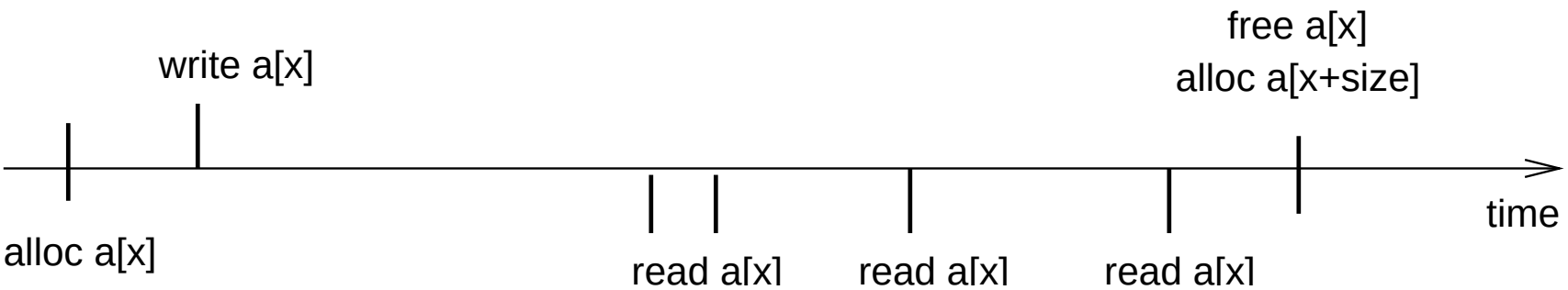


The Glue Code

```
void main(){
  channel struct bigLine c1[];
  channel struct bigLine c2[];
  channel struct bigLine c3[];
  channel struct smallLine c4[];
  channel struct bigLine c5[];

  source(c1, 1);
  source(c3, 2);
  VFilter(c1,c2);
  HFilter(c2, c4);
  composer(c3, c4, c5);
  ramdac(c5);
}
```

Buffer Size



$$\begin{aligned} \theta(\text{write } a[x]) &\geq \theta(\text{alloc } a[x]), \\ \theta(\text{read } a[x]) &> \theta(\text{write } a[x]), \\ \theta(\text{free } a[x]) = \theta(\text{alloc } a[x+\text{size}]) &\geq \theta(\text{read } a[x]). \end{aligned}$$

Apply Farkas and solve.