Programming Models

Rebecca Schultz
Paul Lee
Varun Malhotra
Outline

- Motivation
- Existing sequential languages
- Discussion about papers
- Conclusions and things to ponder over
Principles of programming language design:
- One way is to make it so simple that there are obviously no deficiencies, and
- The other way is to make it so complicated that there are no obvious deficiencies.

(C. A. R. Hoare)
The Role of the Programming Model

- Parallelism in Application
- Programming model
- Support for Parallelism in Hardware
Motivation

- View of a programmer
  - Ease of programming
  - At times can distinguish parallel/independent code segments (not always)
    - Compiler can leverage this fact (whenever possible)
  - Doesn’t want to know the details of the underlying system
  - Hates to care about the locality of data usage
Problems with *common* imperative languages

- Inherently sequential
  - We love to think sequentially (step-by-step)
- Difficult for the compiler to exploit parallelism from the structure of the program

Possible Solutions
- Make programming language restrictive and boost performance
- More information from the programmer to the compiler (implicitly/explicitly)
Motivation

- Many techniques devised to automatically parallelize numerical array based applications
- Does not work well on most non-numerical applications

Main Idea

- Part of the problem is there are too many degrees of freedom at the individual instruction level
- Provide compiler with more information with how you are going to use data
What is a container?

- Any general-purpose aggregate data type
  - Matrices, lists, stacks, trees, graphs, I/O streams, hash tables, etc.
  - Paper focuses on linear containers, content-addressable containers
Abstract containers

- Container behavior is specified using abstract containers and abstract methods
- Abstract container/methods are what the compiler will understand and know how to deal with
- Need to be general enough to be useful
Parallelization techniques

- Loop parallelization
- Container privatization
- Exploit Associativity
  - Used to eliminate update structural dependence
Dependency tests

- For linear containers
  - Range checks
- For content addressable containers
  - Disjoint key analysis: need to assure memory independence
  - Overlapped keys: generally hard, but some patterns
Critique

- **Good**
  - No additional work for programmer when using already defined library such as STL

- **Bad**
  - Less generality ⇒ lost opportunities to parallelize
  - Another thing to get wrong (specifications)

- **Conclusion**
  - Relatively easy way to eke out more out of compilers without burdening programmers (once someone figures out how to do it)
  - Compiler analysis also performed at the container level
Programming in *Jade*

- Data-oriented language for parallelizing programs written in a imperative programming language
  
  *The most important thing in the programming language is the name. A language will not succeed without a good name. I have recently invented a very good name and now I am looking for a suitable language. 😊*
  
  *(Donald E. Knuth)*

- Difficult to program in an “explicit” parallel programming language
Parallel Programming = Parallel “Thinking”?

There is not now, and never will be, a language in which it is the least bit difficult to write bad programs 😊

- Anonymous

...a language is intended not so much to inspire programmers as to protect them from their own not inconsiderable frailties.

- Anonymous
Features of *Jade*

- Augmenting sections of code to be parallelized with data usage information
- Tasks interact through accesses to shared objects
- *Jade* implementation uses this local data usage info to relax program sequential order
Features…

\textbf{Jade\_construct} \{side effect specification\}

(parameters for the task body)

\begin{Verbatim}
\{
  \textbf{Task Body}
}\end{Verbatim}
Jade constructs

- withth
- withonly
  - Doesn’t specify the task will immediately carry out any of the specified effects
- with
  - Use with and withonly and hope that this will reduce the conflicts
- without
Critique

- **Strengths**
  - No burden of *parallel* thought process
  - Achieves speedup comparable to efficient *parallel* programs *(Authors’ claim)*

- **Weaknesses/Deficiencies**
  - Programmer has to identify the tasks to be parallelized
  - Cannot express algorithms requires bi-directional task communication
  - Implementation (discussed in the paper) assumes single address space
Motivation for StreamIt

- High level language for streaming applications
  - Compiler can infer parallelism and communication simply
  - Programmer can ignore architectural details like granularity and topology
- Current backend is RAW, also intended for other parallel architectures
The Programming Model

- Programmer expresses stream graph explicitly
  - *filter* – node in the stream graph (single input, single output)
  - *pipeline, splitjoin, feedbackloop* – connections between nodes
- Communicate between filters via infinite FIFO
- Requires a static-rate streams (I/O rate of each filter known at compile time)
- Programmer need not know details of the underlying topology
The Compiler

- Hierarchical stream graph easy to analyze
- Three phase process
  - **Partitioning** – merge and split filters to get $N$ load balanced filters ($N =$ number of processing units)
  - **Layout** – use simulated annealing and communication cost function to arrange filters
  - **Scheduling** – map infinite FIFO abstraction to actual communication network
StreamIt on the RAW Architecture

- Construct initialization and steady state schedule for the filter
  - Determines code size, buffer sizes etc.
- Complete the partition, layout and communication scheduling steps
  - One filter per tile
  - Can’t exploit parallelism at the tile
Strengths

- Programming Model
  - Programming model is simple, a graph of single-input, single-output nodes connected by predefined constructs

- Compiler
  - Adapts to changes in number of processors, communication network (to some extent)
Weaknesses

- Programming model
  - Requires programmer to learn a new programming model
  - Filters cannot interleave sends and receives, all data must be received before execution
  - Requires static-rate streams

- Compiler
  - Partitioning, layout and scheduling steps assume architecture similar to RAW
Crux of the Story

- Best performance if programming model maps well onto the hardware (StreamIt)
- Large performance gains if some additional information is provided by the programmer (Containers, Jade)
Food for thought

- Where to draw the line between *generality* and *efficiency/performance*
  - Trade-off (e.g., StreamIt)

- A new language in the market, C++-- 😊
  - How much of a performance boost do you expect before you use it
  - What other features do you expect