Programming Models and Tools for High-Performance Reconfigurable Computing

Andrew W. H. House
PhD Candidate
University of Toronto
ahouse@eecg.toronto.edu

CERL Seminar Series
June 1, 2011
High-performance computing (HPC) is being used in more and more areas of scientific and industrial research.

Modern HPC systems operate in the teraflops to petaflops range.

These types of workloads might take weeks to decades on a single computer, depending on the problem size.

- Weather simulation and climate modeling.
- Molecular dynamics and biological simulations.
- Astrophysical simulations.
- Technical computing: computational fluid dynamics, tomography, etc.
Demand for High-Performance Computing

Consider the Top500 Supercomputing Sites:

Total computing power in TOP500

Figure: Performance Growth of the Top500 Supercomputing Sites

Source: http://commons.wikimedia.org/wiki/File:Top500.svg
Processor Performance Improvement

From the January 2011 issue of *IEEE Computer*:

![Microprocessor Performance Growth Trend](image)

**Figure:** Microprocessor Performance Growth Trend
This decrease in the rate of performance improvement does not bode well for “business as usual.”
In HPC there is always demand for **more performance**.

- Single-core performance is no longer increasing as it once did.
- Multicore is a great solution on the desktop, but not for HPC systems that already have many thousands of “cores”.
- For the kind of performance gains that are desired, it is necessary to consider alternate processing architectures.
Outperforming CPUs

How can we beat traditional microprocessors?

- Application-Specific Processors (ASPs)
  - Cell processors, DSPs, GPUs outperform general-purpose CPUs for certain applications.
  - General-purpose GPUs perform very well on many HPC applications, but are power hungry.

- Custom Hardware
  - A custom-designed ASIC should offer the best performance and power.
  - Extremely expensive, difficult, and very inflexible.
  - Some people still do it: MD-GRAPE, Anton

- Or...
FPGAs can offer much of the performance of an ASIC.
Can outperform GPUs for many applications.
Best solution for problems with bit-level parallelism.
Uses less power than CPUs/GPUs, but more than ASIC.
Cutting-edge process technology (28 nm for latest families).
Modern FPGAs are big:
- Thousands of independent, small, on-chip memories.
- Thousands of DSP units (MACs).
- Can implement million-gate designs.
Challenges for FPGA Computing

- Despite potential of FPGAs, GPUs are dominating the acceleration market.
  - GPUs are inexpensive commodity parts.
  - CUDA and OpenCL offer a relatively easy programming model.
- FPGAs are hard to program and integrate into systems.
  - Typical FPGA design flow is VHDL/Verilog circuit design.
  - High-level synthesis tools are improving, but have lower quality of result and work best in the hands of a circuit designer.
  - FPGA accelerator boards are typically expensive, uncommon, and using older technology.
- High-performance reconfigurable computing (HPRC) systems will need many FPGAs.
  - So programming in an HPRC environment will be even harder.
Why is programming for HPC hard?

- Consider a typical modern HPC system:

  - HPC systems are **massively parallel**.
  - Parallel programs are hard to write.
The Problem With Parallel

- Parallel software is hard to write.
- Best performance is achieved by tuning to the particular architecture.
- This makes it even harder for parallel programs scale.
- The de facto HPC programming “standard” today is C, C++, or Fortran that uses a Message Passing Interface (MPI) library.
  - Memory is owned by each process, data is shared via messages.
  - The most scalable form of this is single-program, multiple-data (SPMD).
- But this is not enough, and efforts are under way to find better models.
Using Accelerators in HPC

- Use of accelerators adds another programming language layer.
- Consider a "typical" accelerated HPC system:

```
Interconnection Network

CPU       ASP       CPU
  |       (GPU/FPGA)       |
  |                       |
MEM  CPU  ASP  CPU  ASP
  |       (GPU/FPGA)       |
  |                       |
MEM
```

- Programming is complex and hard to optimize.
- Getting data to/from the accelerator a major bottleneck.
Future Architectures

- Multi-FPGA systems already exist.
- Heterogeneous-peer systems are emerging in the market.
  - Convey HC-1, Nallatech, others.
  - Previous generation of Cray had reconfigurable blades.
  - Xilinx Zynq is also a heterogeneous device.
A New Programming Model Is Needed

- Existing HPC systems need higher productivity.
- A programming model for future HPRC systems must also:
  1. support heterogeneous processing elements,
  2. be scalable,
  3. avoid features that are difficult to synthesize,
  4. assume only limited system services,
  5. support many types and classes of computation,
  6. allow the programmer to expose as much parallelism as possible,
  7. enforce separation of algorithm from implementation,
  8. be independent of the architecture it runs on,
  9. provide a degree of execution model transparency.
## Consider Other Programming Models

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Parallel</td>
<td>HPF</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td></td>
<td>C*</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Dataparallel C</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td></td>
<td>mpC</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RapidMind</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Dataflow and Functional</td>
<td>Simulink</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td></td>
<td>LabVIEW</td>
<td>✓</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td></td>
<td>Prograph</td>
<td>✓</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td></td>
<td>Lustre</td>
<td>✓</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td></td>
<td>Multilisp</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SISAL</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SAC</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Cilk</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CellSs</td>
<td>✓</td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mitron-C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Stream Computing</td>
<td>StreamIT</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CSP</td>
<td>Occam</td>
<td>✓</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MPI</td>
<td>✓✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PVM</td>
<td>✓✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Active Messages</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>HDLs</td>
<td>Verilog/VHDL</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Handel-C</td>
<td>✓</td>
<td>x</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Shared Memory</td>
<td>PRAM</td>
<td>x</td>
<td></td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SHMEM</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>OpenMP</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Linda</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Orca</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PGAS</td>
<td>Split-C</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>UPC</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Titanium</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Co-Array Fortran</td>
<td>x</td>
<td>✓</td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ZPL</td>
<td>x</td>
<td>✓</td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fortress</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>
Some Things to Avoid

A number of features tend to pose problems:

- SPMD programming style
- Dynamic memory allocation and pointers
- Explicit introduction of all parallelism
- Explicit communication between processes
- Explicit mapping of work to processing elements
- Only one kind of parallelism/computation
- Complex or costly process synchronization
Some Things to Include

Other features often provide solutions:

- Data-parallel operations
- Implicit communication and synchronization
- Emphasis on libraries and functions
- Functions without side effects
- Global view of memory
- Data streams and implicit parallelism
- Region-based array management
A New Language

We decided to build a new language: ARMADA

- Designed to satisfy our requirements.
- Tries to incorporate the good features and leave out the bad.
- Usability as much a concern as capability.
  - Intended users are scientists and researchers, not professional software developers or computer engineers.
  - Want something familiar and easy to use with a high level of abstraction.
  - Want to be able to express math and simulations naturally.
Platform Independence

- Armada programs can be written with no knowledge of the target platform.
  - A description of the target platform is provided separately; the tools map the program to the platform.
- Armada programs are intended to purely express the algorithm.
  - Implementation details can be embedded in “directives” or supplied by a separate application description file.
- Armada programs allow the programmer to express as much parallelism as possible.
  - It is up to the back-end tools to try to make the most out of it.
The Armada Design Flow

1. Armada program file(s)
2. Front-End Compiler
   - Application Description File
   - Platform Description File
3. Application Partitioning
   - Partitioned, Annotated AIR files
   - AIR files
4. Code Generators:
   - C++ Code Generator
   - C++ and MPI Code Generator
   - HDL/HLS Code Generator
   - OpenCL Code Generator
Armada supports a variety of data types.

- **Scalar types:**
  - Bitstring
  - Integer
  - Real
  - Complex
  - Boolean
  - String

- **Aggregate types:**
  - Array
  - Tuple
  - Graph
Armada:

- Is a static, strongly-typed compiled language.
- Provides high-level operators and built-in functions (matrix operations, etc.).
- Has no pointers, aliasing, or dynamic memory allocation.
- Is interpreted as dataflow.
- Has implicit communication and synchronization.
- Has a PGAS-style global view of memory.
Specific Language Features

Armada:

- Supports data-parallel operations.
  
  ```
  var a, b, c, is array[1 to 100] of int := /* initialization */ ;
  c := a * b; // element-wise multiplication of a and b
  ```

- Has functions that are free from side effects and can have multiple return values.
  
  ```
  x, y := foo(a, b, c);
  ```

- Has parallel for loops.
  
  ```
  forall (i in 1 to 10) {
      a[i] := b[i]^2 - 4*c[i];
  }
  ```

- Uses region-based array indexing.
  
  ```
  c[11 to 20] := a[1 to 10] * b[91 to 100];
  ```
Organization of Armada Programs

Armada programs execute in an explicitly defined order:

1. **Initialize**
   - Run on a host system that has disk I/O.
   - Reads input data from files, sets up initial data structures, passes control to main.

2. **Main Execution**
   - Runs on whole system.
   - No disk or terminal I/O allowed.
   - Sends results to finalize.

3. **Finalize**
   - Runs on a host system that has disk I/O.
   - Writes results passed from main execution to terminal/file/etc.
Armada Code Sample (N-Body Simulation)

```plaintext
for (i := 1 to NUMSTEPS)
{
  // Calculate forces
  foreach (i, j in i := 1 to NP : j := i+1 to NP)
  {
    var triple is forceij := calculateForce(pMass[i], pMass[j],
                                           pPos[i], pPos[j]);
    allForces[i,j] := forceij;
    allForces[j,i] := -forceij;
  }

  // sum columns of allForces array
  summedForces := sum(allForces[1 to NP, 1 across NP]);

  // update velocities and positions
  var a is array[1 to NP] of triple := summedForces/pMass;
  pPos := pPos + (pVel*TIMESTEP) + (0.5 * (TIMESTEP^2) * a);
  pVel := pVel + (a*TIMESTEP);
}
```

Andrew W. H. House
Programming Models and Tools for HPRC
The Armada language is the front-end of the system. It allows programmers to express their algorithm. But Armada relies on the back-end to map the application to a particular platform.

An intermediate representation (IR) is a representation meant to allow easy manipulation of the program.

- Simplifies optimizations, various kinds of error checking, etc.

Software IR: typically a form of three-address code (pseudo-instructions).

Hardware IR: typically a netlist of modules.
A New IR

Armada is both a hardware and software language:

- **Common software IRs make assumptions:**
  - Memory model.
  - Sequential operation.
  - Low level.

- **Netlists also have built-in assumptions:**
  - Microarchitecture.
  - Timing.
  - Low level.

- **We need something new for a new programming model.**
We have developed the Armada IR (AIR):

- AIR represents programs a control/dataflow hypergraph.
  - Nodes are operations.
  - Edges are data movement.
- There are no loads and stores.
  - Data movement can be mapped to memory read/write, register read/write, wires, bus transfers, function calls, etc.
  - Allows late optimization to target specific platform.
- Nodes are high level.
  - Represent operation, size of data being operated on, annotations on suggested implementation, and more.
  - Nodes also represent loop entry/exit, conditionals, etc.
A Closer Look at AIR

- Edges have a single source node but can have multiple destination nodes.
- Writes to a variable are represented as new edges with the same name.
- Data-parallelism captured within nodes.
- Instruction/task parallelism captured in graph structure.
AIR Example

```
// Note that a, b, c, d are arrays.
c := (a + b) / 2;
for each (i, j in i := 1 to 10 : j := i to 10) {
    d[i, j] := d[i, j] + c[i] * c[j];
}
```

- Nodes and edges are annotated with various information:
  - Data type, data size, implementation directives.
  - Also “computational density” of nodes.
Advantages of AIR

- AIR keeps the algorithm high level.
  - Worry about low level optimization when targeting specific device.

- AIR allows flexibility in mapping.
  - Memory mapping, implementation of nodes is flexible.

- AIR exposes communication patterns.
  - Edges represent all data movement.
  - Easy to find high-bandwidth communications, highly-connected operations, etc.

- Exposes amount of computation.
  - Number of nodes, size of data each operates on can give a sense of the “computational density”.

The Armada language provides a friendly face to users.

AIR provides a convenient representation for software tools.

Those tools are where the magic has to happen.

The Armada back-end is designed to leverage as much existing infrastructure as possible.

- Source-to-source compilation allows use of compilers and APIs designed for each target processing element.
- Makes use of communication standards where possible. (MPI)
- Avoids having to reinvent the wheel.
Recall: The Armada Design Flow

Armada program file(s)

Application Description File

Platform Description File

Front-End Compiler

AIR files

Application Partitioning

Partitioned, Annotated AIR files

C++ Code Generator

C++ and MPI Code Generator

HDL/HLS Code Generator

OpenCL Code Generator
Describing Target Platforms

- Armada makes no assumptions about the target platform.
- Impossible to compile if we know *nothing*, though.
- A separate platform description must be provided.
- Decided to use XML as the format.
- But XML is tedious and verbose.
- This forced me to create Alpha:
  - Armada Language for Platform Hardware Abstraction
  - Alpha is a domain-specific language to facilitate the hierarchical description of computing platforms.
Alpha Features

- Alpha describes systems as a graph connecting nodes, clusters, and hubs.

- **Nodes** are self-contained computational units (FPGA, GPU, multicore processor).

- **Hubs** are communication points used to represent shared bandwidth.

- **Clusters** are higher-order modules that can contain nodes, hubs, or other clusters.
  - Clusters present an aggregate of the capabilities of all the components.
  - Communication bandwidth specified by the ports of nodes, hubs, and clusters, not the links themselves.
import "bee3.hpd";

node SomeFPGA(inout full a[1800]) {
    type = FPGA;
    internal = <256, 8, 4096>, <16, 8, 65536>;
    soft = <100000, 6>;
}

hub LocalHub(inout full a1[1800]) {
    capacity = 5000;
}

cluster FPGACluster(inout full a[4000]) {
    link unichan1, unichan2;
    array mgt[0 + 3 - 1 to 3*1] of link;
    array n[0 to 3] of SomeFPGA;
    for (i = 0 to 1) {
        if (i = 0) {
            n[i](mgt[i*10/10+0]);
        } elsif (i = 1) {
            n[i](mgt[i*10/10+1-1]);
        } else {
            n[i](mgt[i]);
        }
    }
    OtherFPGA n3(unichan1, unichan2);
    LocalHub h1(mgt[0+1*1-1]);
}

- Computational resources are described by some mix of:
  - Element type.
  - Internal memory.
  - External memory.
  - Processor cores.
  - Soft logic (FPGA fabric).
  - Hard logic (FPGA DSP blocks, for example).

- Obviously only a rough approximation, but hopefully enough to be useful.
Communication between Processing Elements

- We want to leverage existing communication infrastructure where possible.
  - Use MPI to communicate between processors in a system.
  - Use OpenCL or CUDA to communicate with GPUs.
- What about FPGAs?
  - Need to get data to the FPGAs.
  - FPGAs need to talk to each other.
  - There is no standard.
  - Would like to use MPI if we could.
ArchES-MPI evolved from TMD-MPI.

Software library allows CPUs to talk to FPGAs, processors on FPGAs to talk to each other.

The Message Passing Engine (MPE) provides a hardware interface to the on-chip network.

Hardware and software processes can talk to each other.

We have developed interfaces to ArchES-MPI from Impulse-C, AutoPilot, Cynthesizer.
Application Partitioning

Partitioning is key.

- Planned approach is to partition AIR graph into subgraphs intended for each processing element.
- Edges that connect subgraphs to each other will be implemented using MPI or other appropriate protocols.
- Partitioning will be driven by:
  - Communication bandwidth. (Fit value of edges between subgraphs to capabilities of platform.)
  - Computational density versus resources. (Fit AIR nodes into processing elements capable of supporting them.)
  - Type of computation. (Maybe. Eventually.)

- **This will be a hard problem!**

- I hope to get “good enough” performance.
Code Generation

- Code generation system is designed to be easily extensible.
- Takes as input an AIR graph/subgraph, generates output code for target processing element.
- Inserts appropriate communication calls (MPI, OpenCL) at the edges of the graph.
- Attempts to generate source code that downstream tools will have an easy time with.
- Generates scripts to compile everything. (Maybe. Eventually.)
- Requires a collection of appropriate downstream tools for all the available processing elements in the target system.
Remaining Work

- Completed or In Progress
  - Alpha to XML compiler
  - Armada to C++ compiler
  - Armada to AIR compilation

- To Be Started
  - Code generation from AIR
  - Partitioning (the big one)
Future Directions

- Extending support from just FPGAs to other processing elements.
- Tighter integration with downstream tools.
- Improving partitioning.
  - Matching parts of the program to the architecture best-suited.
  - Tuning my partitioning approach or finding better ones.
- Incorporating automated precision analysis.
- Supporting partial reconfiguration?
Conclusion

- New programming models are needed for emerging heterogeneous HPC architectures.
- Future systems at all levels will be heterogeneous: manycore processors, GPUs, FPGAs all in one system.
- We developed the Armada tool chain as an attempt to meet these challenges.
  - Armada language eases application development.
  - AIR provides a framework on which tools can be built.
  - Alpha describes target computing systems.
  - Partitioning maps algorithms onto systems.
  - Code generation uses downstream tools and existing infrastructure to make it all work.
- The Armada approach has much to offer, but realistically new programming models will likely be developed incrementally.