SPIRE: A Methodology for Sequential to Parallel Intermediate Representation Extension

Dounia Khaldi  Pierre Jouvelot  François Irigoin  Corinne Ancourt

CRI, Mathématiques et systèmes
MINES ParisTech
35 rue Saint-Honoré, 77300 Fontainebleau, France

HiPEAC Computing Systems Week, Paris, France
May 03, 2013
Context and Motivation

- Cilk, UPC, X10, Habanero-Java, Chapel, OpenMP, MPI, OpenCL...
- Parallelism handling in compilers
- A Parallel Intermediate Representation
  - Trade-off between expressibility and conciseness of representation
  - Generic (language-neutral) and simplicity
- Huge compiler platforms
  - GCC (more than 7 million lines of code)
  - LLVM (more than 1 million lines of code)
  - PIPS (600 000 lines of code)

Proposal

SPIRE: Sequential to Parallel Intermediate Representation Extension methodology
### Rich Related Work!

<table>
<thead>
<tr>
<th>AST</th>
<th>Parallel IR</th>
<th>Parallellism</th>
</tr>
</thead>
</table>
| Habanero-Java [Cavé et al., 2011] | HPIR [Zhao and Sarkar, 2011] | New nodes:  
AsyncRegionEntry,  
AsyncRegionExit,  
FinishRegionEntry,  
FinishRegionExit |
| — | InsPIRe [Insieme, ] | Built-ins                                      |
| — | PLASMA [Pai et al., 2010] | Vector operators:  
add, reduce and par |

<table>
<thead>
<tr>
<th>Graph</th>
<th>Parallel Program Graph [Sarkar and Simons, 1993]</th>
<th>mgoto control and synchronization edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>—</td>
<td>Hierarchical Task Graph [Girkar and Polychronopoulos, 1992]</td>
<td>New nodes for tasks</td>
</tr>
<tr>
<td>—</td>
<td>Stream Graph [Choi et al., 2009]</td>
<td>Nodes for streams</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LLVM IR [Team, 2010]</th>
<th>LLVM PIR</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>llvm.loop.parallel</td>
</tr>
</tbody>
</table>
Outline

1. Design Approach
2. PIPS (Sequential) IR
3. SPIRE, a Sequential to Parallel IR Extension
4. SPIRE Operational Semantics
5. Validation: Application to LLVM
6. Conclusion
"parallelism or concurrency are operational concepts that refer not to the program, but to its execution." [Dijkstra, 1977]

<table>
<thead>
<tr>
<th>Language</th>
<th>Execution</th>
<th>Synchronization</th>
<th>Memory</th>
<th>Data distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cilk (MIT)</td>
<td>—</td>
<td>spawn</td>
<td>—</td>
<td>cilk_lock</td>
</tr>
<tr>
<td>Chapel (Cray)</td>
<td>forall coforall cobegin</td>
<td>begin</td>
<td>—</td>
<td>PGAS (Locales)</td>
</tr>
<tr>
<td>X10(IBM), Habanero-Java(Rice)</td>
<td>foreach async future</td>
<td>finish</td>
<td>next force get</td>
<td>(on)</td>
</tr>
<tr>
<td>OpenMP</td>
<td>omp for omp sections</td>
<td>omp task</td>
<td>omp taskwait</td>
<td>Shared</td>
</tr>
<tr>
<td>OpenCL</td>
<td>EnqueueND-RangeKernel</td>
<td>EnqueueTask</td>
<td>Finish</td>
<td>private, shared...</td>
</tr>
<tr>
<td>MPI</td>
<td>MPI_Init</td>
<td>MPI_spawn</td>
<td>MPI_Finalize</td>
<td>Distributed</td>
</tr>
<tr>
<td>SPIRE</td>
<td>sequential, parallel</td>
<td>spawn</td>
<td>barrier signal wait</td>
<td>Shared, Distributed</td>
</tr>
</tbody>
</table>


instruction = call + forloop + sequence;

statement = instruction x declarations:entity*;

entity = name:string x type x initial:value;

forloop = index:entity x
          lower:expression x upper:expression x
          step:expression x body:statement;

sequence = statements:statement*;
SPIRE: Execution

execution = sequential:unit + parallel:unit;

- Add execution to control constructs:
  - loop
  - sequence

forall I in 1..n do
  t[i] = 0;

forloop(I,1,n,1,
  t[i] = 0,
  parallel)

forall in Chapel, and its SPIRE core language representation
**SPIRE: Synchronization**

\[
\text{synchronization} = \text{none}\!: \text{unit} + \\
\text{spawn}\!: \text{entity} + \text{barrier}\!: \text{unit} + \\
\text{single}\!: \text{bool} + \text{atomic}\!: \text{reference};
\]

- Add synchronization to statement

```c
mode = OUT_OF_ORDER_EXEC_MODE_ENABLE;
c = clCreateCommandQueue(context, device_id, mode, &err);
clEnqueueTask(c, k_A, 0, NULL, NULL);
clEnqueueTask(c, k_B, 0, NULL, NULL);
cEnqueueBarrier(c);
clEnqueueTask(c, k_C, 0, NULL, NULL);
```

OpenCL example illustrating spawn and barrier statements, and its SPIRE core language representation
event newEvent(int i);
void freeEvent(event e);
void signal(event e);
void wait(event e);

- Add event as a new type

```java
finish{
    phaser ph=new phaser();
    for (j = 1; j <= n; j++) {
        async phased(
            ph<SIG_WAIT>){
                S; next; S';
        }
    }
}
```

```java
barrier(
    ph=newEvent(-(n-1));
    forloop(j, 1, n, 1,
        spawn(j, S;
            signal(ph);
            wait(ph);
            signal(ph);
            S'),
            parallel);
    freeEvent(ph)
)
```

A phaser in Habanero-Java, and its SPIRE core language representation
$S \in \text{Stmt} ::= \text{nop} \mid \text{I=E} \mid S_1;S_2 \mid \text{loop}(E,S)$

$S \in \text{SPIRE} (\text{Stmt}) ::=$

$\text{nop} \mid \text{I=E} \mid S_1;S_2 \mid \text{loop}(E,S) \mid$

$\text{spawn}(I,S) \mid$

$\text{barrier}(S) \mid$

$\text{wait}(I) \mid \text{signal}(I) \mid$

$\text{send}(I,I') \mid \text{recv}(I,I')$

$m \in \text{Memory} = \text{Ide} \rightarrow \text{Value}$

$\kappa \in \text{Configuration} = \text{Memory} \times \text{Stmt}$

$\zeta \in \text{Exp} \rightarrow \text{Memory} \rightarrow \text{Value}$

$$v = \zeta(E)m$$

$$(m, I = E) \rightarrow (m[I \rightarrow v], \text{nop})$$
\( \kappa \in \text{Configuration} = \text{Memory} \times \text{Stmt} \)

\( \pi \in \text{State} = \text{Proc} \to \text{Configuration} \times \text{Procs} \)

\( i \in \text{Proc} = N \)

\( c \in \text{Procs} = \mathcal{P} (\text{Proc}) \)

\( \text{dom} (\pi) = \{ i \in \text{Proc} / \pi (i) \text{ is defined} \} \)

\( \pi [i \to (\kappa, c)] \) the state \( \pi \) extended at \( i \) with \( (\kappa, c) \)

\[
\begin{align*}
\kappa & \to \kappa' \\
\pi [i \to (\kappa, c)] & \leftrightarrow \pi [i \to (\kappa', c)]
\end{align*}
\]

\[
\begin{align*}
n & = \zeta (I)m \\
\pi [i \to ((m, \text{spawn}(I, S)), c)] & \leftrightarrow \\
\pi [i \to ((m, \text{nop}), c \cup \{n\})] \\
[n & \to ((m, S), \emptyset)]
\end{align*}
\]
Validation: LLVM

function = blocks: block*;
block = label: entity x phi_nodes: phi_node* x
       instructions: instruction* x terminator;
phi_node = call;
instruction = call;
terminator = conditional_branch + unconditional_branch +
            return;
conditional_branch = value: entity x label_true: entity x
                    label_false: entity;
unconditional_branch = label: entity;
return = value: entity;
function = blocks:block*;
block = label:entity x phi_nodes:phi_node* x
      instructions:instruction* x terminator;
phi_node = call;
instruction = call;
terminator = conditional_branch + unconditional_branch +
            return;
conditional_branch = value:entity x label_true:entity x
                    label_false:entity;
unconditional_branch = label:entity;
return = value:entity;

entry:
    br label %bb1
bb: ; preds = %bb1
%0 = add nsw i32 %sum.0, 2
%1 = add nsw i32 %i.0, 1
br label %bb1
bb1: ; preds = %bb, %entry
%sum.0 = phi i32 [42,%entry],[%0,%bb]
%i.0 = phi i32 [0,%entry],[%1,%bb]
%2 = icmp sle i32 %i.0, 10
br i1 %2, label %bb, label %bb2

sum = 42;
for(i=0; i<10; i++)
    sum = sum + 2;
}
function = blocks: block*;
block = label: entity x phi_nodes: phi_node* x
       instructions: instruction* x terminator;
phi_node = call;
instruction = call;
terminator = conditional_branch + unconditional_branch +
              return;
conditional_branch = value: entity x label_true: entity x
                    label_false: entity;
unconditional_branch = label: entity;
return = value: entity;
function = blocks: block*;
block = label: entity x phi_nodes: phi_node* x
  instructions: instruction* x terminator;
phi_node = call;
instruction = call;
terminator = conditional_branch + unconditional_branch +
  return;
conditional_branch = value: entity x label_true: entity x
  label_false: entity;
unconditional_branch = label: entity;
return = value: entity;

function \leadsto function x execution;
block \leadsto block x execution;
instruction \leadsto instruction x synchronization;
Intrinsic Functions: send, recv, signal, wait
**Conclusion**

- SPIRE methodology as an IR transformer:
  - 10 concepts collected in three groups
    - execution, synchronization, data distribution
  - SPIRE (PIPS), SPIRE (LLVM)
  - Operational semantics for SPIRE [Khalid et al., 2012b]
  - Trade-off between expressibility and conciseness of representation

- Applications:
  - Implementation of SPIRE (PIPS)
  - Implementation of a new BDSC-based task parallelization algorithm [Khalid et al., 2012a]
  - Generation of OpenMP and MPI code from the same parallel IR

**Future Work**

- Transformations for parallel languages encoded in SPIRE
- Representation of the PGAS memory model
- Addition of programming features such as exceptions and speculation
References

In 9th International Conference on the Principles and Practice of Programming in Java (PPPJ).

Stream Compilation for Real-Time Embedded Multicore Systems.

Re: “Formal Derivation of Strongly Correct Parallel Programs” by Axel van Lamsweerde and M. Sintzoff.
circulated privately.

The Program Dependence Graph And Its Use In Optimization.

Automatic Extraction of Functional Parallelism from Ordinary Programs.

Insieme.
Insieme - an Optimization System for OpenMP, MPI and OpenCL Programs.
http://www.dps.uibk.ac.at/insieme/architecture.html.

Parallelizing with BDSC, a Resource-Constrained Scheduling Algorithm for Shared and Distributed Memory Systems.

SPIRE: A Sequential to Parallel Intermediate Representation Extension.
SPIRE: A Methodology for Sequential to Parallel Intermediate Representation Extension

Dounia Khaldi  Pierre Jouvelot  François Irigoin  Corinne Ancourt

CRI, Mathématiques et systèmes
MINES ParisTech
35 rue Saint-Honoré, 77300 Fontainebleau, France

HiPEAC Computing Systems Week, Paris, France
May 03, 2013
void **send** (int dest, entity buf);
void **recv** (int source, entity buf);

MPI_Init(&argc, &argv[]);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
if (rank == 0)
    MPI_Recv(sum, sizeof(sum), MPI_FLOAT, 1, 1, MPI_COMM_WORLD,&stat);
else if(rank == 1) {
    sum = 42;
    MPI_Send(sum, sizeof(sum), MPI_FLOAT, 0, 1, MPI_COMM_WORLD);
}
MPI_Finalize();

forloop(rank,0,size,1, test(rank==0, recv(one,sum),
    test(rank==1, sum=42;
    send(zero,sum),
    nop)
), parallel)
void **send**(int dest, entity buf);  
void **recv**(int source, entity buf);  

```c
MPI_Init(&argc, &argv[]);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
if (rank == 0)
    MPI_Recv(sum, sizeof(sum), MPI_FLOAT, 1, 1, MPI_COMM_WORLD, &stat);
else if (rank == 1) {
    sum = 42;
    MPI_Send(sum, sizeof(sum), MPI_FLOAT, 0, 1, MPI_COMM_WORLD);
}
MPI_Finalize();
```

```c
forloop(rank, 0, size, 1,
    test(rank==0, recv(one, sum),
        test(rank==1, sum=42;
            send(zero, sum),
                nop)
            ),
        parallel)
```
SPIRE: Data Distribution

- Non-blocking send ≡

  \[
  \text{spawn}(\text{new, } \text{send}(\text{zero, } \text{sum}))
  \]

- Non-blocking receive ≡

  \[
  \text{finish\_recv} = \text{false};\nn\text{spawn}(\text{new, } \text{recv}(\text{one, } \text{sum});\n  \text{atomic}(\text{finish\_recv=\text{true}}))
  \]

- Broadcast ≡

  \[
  \text{test}(\text{rank==0,}\nn  \text{forloop}(\text{rank,1,}\text{size,1,}\nn  \text{send}(\text{rank,}\text{sum}),\n  \text{parallel}),\n  \text{recv}(\text{zero,}\text{sum}))
  \]