Goal of run-time parallelization

- Typical target: **irregular loops**
  
  ```
  for ( i=0; i<n; i++)
      a[i] = f ( a[g(i)], ...);
  ```

- Array index expressions `g, h...` depend on run-time data
- Iterations cannot be statically proved independent (and not either dependent with distance +1)

**Principle:**
- At runtime, inspect `g, h...` to find out the real dependences and compute a schedule for partially parallel execution
  - Can also be combined with speculative parallelization

Overview

- Run-time parallelization of irregular loops
  - DOACROSS parallelization
  - Inspector-Executor Technique (shared memory)
  - Inspector-Executor Technique (message passing)
  - Privatizing DOALL *
- Speculative run-time parallelization of irregular loops *
  - LRPD Test *
- General Thread-Level Speculation
  - Hardware support *

* = not yet covered in this lecture. See the references.

DOACROSS Parallelization

- Useful if dependence distances are unknown, but often > 1
- Allow independent subsequent loop iterations to overlap
- Bilateral synchronization
- Simple example for shared memory:
  ```
  for ( i=0; i<n; i++)
      a[i] = f ( a[g(i)], ...);
  ```

  ```
  sh flag done[n];  // flag array, initialized to 0 (not done);
  forall ( i=0; i<n; i++) {
      if (g(i)<i) wait until done[ g(i) ];
      a[i] = f ( a[g(i)], ... );
      set( done[i] );
  }
  ```

A kind of run-time software pipelining
Inspector-Executor Technique (1)

- Compiler generates 2 pieces of customized code for such loops:
  - **Inspector**
    - calculates values of index expression by simulating whole loop execution
      - typically, based on sequential version of the source loop (some computations could be left out)
    - computes implicitly the real iteration dependence graph
    - computes a parallel schedule as (greedy) wavefront traversal of the iteration dependence graph in topological order
      - all iterations in same wavefront are independent
    - schedule depth = #wavefronts = critical path length
  - **Executor**
    - follows this schedule to execute the loop

Inspector-Executor Technique (2)

- **Source loop**:
  ```c
  for (i=0; i<n; i++)
  a[i]  =  f ( a[ g(i) ], a[ h(i) ], ... );
  ```

- **Inspector**:
  ```c
  int wf[n];  // wavefront indices
  int depth = 0;
  for (i=0; i<n; i++)
    wf[i] = 0;   // init.
  for (i=0; i<n; i++) {
    wf[i] = max( wf[ g(i) ], wf[ h(i) ], ... ) + 1;
    depth = max( depth, wf[i] );
  }
  ```

  Inspector considers only flow dependences (RAW), anti- and output dependences to be preserved by executor

Inspector-Executor Technique (3)

- **Example**:
  ```c
  for (i=0; i<n; i++)
  a[i]  =  ... a[ g(i) ] ...;
  ```

- **Executor**:
  ```c
  float aold[n];  // buffer array
  aold[1:n] = a[1:n];
  for (w=0; w<depth; w++)
    for (i=0; i<n; i++)
      if (wf[i] == w)  {
        a[i] = (g(i) < i)? a[ g(i) ] : aold[ g(i) ];
        ...  // similarly, a2 for h etc.
        a[i] =  f ( a1, a2, ... );
      }
  ```

Inspector-Executor Technique (4)

- **Problem**: Inspector remains sequential – no speedup

- **Solution approaches**:
  - Re-use schedule over subsequent iterations of an outer loop if access pattern does not change
    - amortizes inspector overhead across repeated executions
  - Parallelize the inspector using doacross parallelization [Saltz,Mirchandaney'91]
  - Parallelize the inspector using sectioning [Leung/Zahorjan'91]
    - compute processor-local wavefronts in parallel, concatenate
    - trade-off schedule quality (depth) vs. inspector speed
  - Parallelize the inspector using bootstrapping [Leung/Z.'91]
    - Start with suboptimal schedule by sectioning, use this to execute the inspectorÆrefined schedule

Inspector-Executor Technique (5) - DMS

- Global address space (GAS) languages for DMS (HPF, UPC, NestStep, Co-Array Fortran, ...)
  - Compiler must insert necessary Send / Recv operations to move data from owning to reading processor
  - Necessary even for (irregular) parallel loops (iterations are statically asserted to be independent, e.g. by user directive)
  - Can use inspector-executor method for run-time scheduling of communication in irregular loops

- **Inspector**
  - determines communication map + reverse map (schedule):
    - Who has to send which owned elements to whom
  - allocate buffer for received elements; adapt access functions

- **Executor**
  - communicates according to schedule
  - executes loop

Example:
```c
forall (i, 0, 12, #)
  y[i]  =  y[i] + a[ ip[i] ] * x[i]
```
Inspector-Executor Technique (6) - DMS

Inspector step 1: construct communication map (here, in parallel)

<table>
<thead>
<tr>
<th>dest</th>
<th>source</th>
<th>data</th>
<th>local buffer area (private)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>P1</td>
<td>a[0], a[6]</td>
<td>lb[0:1]</td>
</tr>
</tbody>
</table>

Inspector-Executor Technique (7) - DMS

Inspector step 2: construct reverse communication map (communication schedule)

<table>
<thead>
<tr>
<th>source</th>
<th>dest</th>
<th>data</th>
<th>remote buffer area</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>P1</td>
<td>a[0], a[6]</td>
<td>lb[0:1]</td>
</tr>
</tbody>
</table>

Inspector-Executor Technique (8) - DMS

Inspector, step 3: Construct modified access functions (represented as local table of pointers)

```
<table>
<thead>
<tr>
<th>ip[i]</th>
<th>owner of y[i]</th>
<th>local memory</th>
<th>remote buffer area</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>P0</td>
<td>a[0]</td>
<td>lb[0:1]</td>
</tr>
<tr>
<td>1</td>
<td>P0</td>
<td>a[1]</td>
<td>lb[2]</td>
</tr>
</tbody>
</table>
```

Remark: Communication maps and address tables can be reused if ip[] does not change between subsequent executions of the source loop.

Inspector-Executor Technique (9) – DMS

Executor:

```plaintext
// send data according to reverse communication map:
for each Pj in dest
    send requested a[i] elements to Pj
// receive data according to communication map:
for each Pi in source
    recv a[i] elements, write to respective lb entries
// Remark: the above part can be skipped in subsequent
// executions of the executor if IP[i] and a[i] do not change.
// execute loop with modified access function:
forall (i, 0, 12, #)
    y[i] = y[i] + (* accesstable[i] ) * x[i];
```

Some references on run-time parallelization


Thread-Level Speculation
Speculative execution

- For automatic parallelization of sequential code where dependences are hard to analyze statically
- Works on a task graph
  - constructed implicitly and dynamically
- Speculate on:
  - control flow, data independence, synchronization, values
We focus on thread-level speculation (TLS) for CMP/MT processors. Speculative ILP is not considered here.
- Task:
  - statically: Connected, single-entry subgraph of the control-flow graph
    - Basic blocks, loop bodies, loops, or entire functions
  - dynamically: Contiguous fragment of dynamic instruction stream within static task region, entered at static task entry

Speculative execution of tasks

- Speculation on inter-task control flow
  - After having assigned a task, predict its successor task and start it speculatively
- Speculation on data independence
  - For inter-task memory data (flow) dependences
    - conservatively: await write (memory synchronization, message)
    - speculatively: hope for independence and continue (execute the load)
- Roll-back of speculative results on mis-speculation (expensive)
  - When starting speculation, state must be buffered
  - Squash an offending task and all its successors, restart
- Commit speculative results when speculation resolved to correct
  - Task is retired

TLS Example

- Code view
- Sequential thread view
- TLS thread view
- Exploiting module-level speculative parallelism (across function calls)

Data dependence problem in TLS

- Original thread T1, T2, T3
- Dependences between threads
- Task dependencies
- Task is retired

Selecting Tasks for Speculation

- Small tasks:
  - too much overhead (task startup, task retirement)
  - low parallelism degree
- Large tasks:
  - higher misspeculation probability
  - higher rollback cost
  - many speculations ongoing in parallel may saturate the resources
- Load balancing issues
  - avoid large variation in task sizes
- Traversal of the program’s control flow graph (CFG)
- Heuristics for task size, control and data dep. speculation

TLS Implementations

- Software-only speculation
  - for loops [Rauchwerger, Padua ’94, ’95]
  - ...
- Hardware-based speculation
  - Typically, integrated in cache coherence protocols
  - Used with multithreaded processors / chip multiprocessors for automatic parallelization of sequential legacy code
  - If source code available, compiler may help e.g. with identifying suitable threads
Some references on speculative execution / parallelization