What is StarPU?

- A task-based runtime
- Similar in motivation to Legion
  - Very similar!
  - Even though the projects are independent

History

- First paper in 2008
  - First called StarPU in 2009
- Development and application papers through today

The Basics

- Task-based
  - Dependencies define execution order constraints between tasks
- Task inputs and outputs must be explicitly declared
  - Along with Read, Write, Read/Write
- Hierarchical partitioning of data
- Programmer gets
  - Automatic task scheduling
  - Automated data movement
Execution Pipeline

- The application submits tasks & dependencies
- The StarPU runtime maintains a dependence graph of tasks
- Tasks that are not dependent on other tasks are placed in a work queue
- A scheduler assigns tasks from the work queue to processors

Execution Model

Declaring Dependencies

The application is responsible for declaring dependencies between tasks

```plaintext
declare_deps(tagB, 1, tagA);
declare_deps(tagC, 1, tagA);
declare_deps(tagD, 2, tagB, tagC);
```

task1->tag_id = tagA;
task2->tag_id = tagD;
...
submit_task(task1);
submit_task(task2);
...
tag_wait(tagD);
```
Maintaining Dependencies

- Tasks also must declare privileges on data
  - Read, Write, Read/Write
- Data dependencies between tasks are discovered by the runtime
  - A dependency between A and B
  - A writes some data, B reads it
  - System will move the data if necessary to where B executes

Scheduling

- Given a set of ready-to-execute tasks:
  - Which one should be executed next?
  - On which processor?
- Tasks may have variants that allow the same task to be run on different kinds of processors
  - E.g., CPUs or GPUs
  - Just like Sequoia/Legion

Scheduling Heuristic

- Estimate the time $\text{Time}(t, p)$ to run task $t$ on processor $p$
  - Estimates can be obtained from programmer-supplied models or from profiling
- $\text{Latency}(p) = \sum \text{Time}(t, p)$
  - Where the sum is over tasks assigned to $p$
- Send a new task $t'$ to the processor $p'$ that minimizes $\text{Time}(t, p) + \text{Latency}(p)$

Priorities

- The scheduling heuristic is FIFO
- Tasks can also have priorities
  - Allow important tasks to jump the queue
  - Doesn't necessarily interact well with the scheduling heuristic
- Many other scheduling policies have been explored for StarPU
Partitioning Data

- Data can be partitioned using filters
  - Can express blocking of rectangular collections
- Can also be applied recursively
  - i.e., can express hierarchical partitioning
- And dynamically
  - All partitioning done at runtime

Partitioning Example

```c
h = register_matrix( &matrix, ptr, n, n, ...)
map_filters(matrix, 2, filter_row, 3, filter_col, 3)
block = get_sub_data(matrix, 2, 2, 0);
map_filters(block, 2, filter_row, 2, filter_col, 2);
subblock = get_sub_data(block, 2, 1, 0);
```

Automated Data Movement

- Multiple tasks may access the same data
- And in different ways
  - Reading, writing, reading and writing
- Need to solve two problems
  - Be lazy - don’t move data unless necessary
    - E.g., to have multiple copies if everyone is reading
  - But need to ensure tasks have most recent version
    - If a task writes, future reads must come from that version of the data
Cache Coherence

- Managing data coherence is not a new problem
- The original and best known version occurs in cache coherent multiprocessors

Cache Coherence Problem

- Want to cache shared data to reduce access time
- Problem
  - Two caches with inconsistent values

1. Single-Writer, Multiple-Read (SWMR) Invariant
   For any memory location A, in any given epoch, there is
   - one processor that may write (and read) A, or
   - some number of processors that may only read A

2. Data-Value Invariant
   The value of the memory location at the start of an epoch is the same as the value of the memory location at the end of its last read-write epoch
Snoopy Cache Coherence

- Solution
  - Have caches “snoop” on other caches’ reads and writes
  - Writes: Invalidate other cached copies (READ EXCLUSIVE)
  - Reads: Get latest version from other cache, if it has a “dirty” copy (READ SHARED)
  - Coherence at granularity of cache lines

Snoopy Cache Coherence 2

- Solution
  - Have caches “snoop” on other caches’ reads and writes
  - Writes: Invalidate other cached copies (READ EXCLUSIVE)
  - Reads: Get latest version from other cache, if it has a “dirty” copy (READ SHARED)
  - Coherence at granularity of cache lines

Snoopy Cache Coherence 3

- Solution
  - Have caches “snoop” on other caches’ reads and writes
  - Writes: Invalidate other cached copies (READ EXCLUSIVE)
  - Reads: Get latest version from other cache, if it has a “dirty” copy (READ SHARED)
  - Coherence at granularity of cache lines

Snoopy Cache Coherence 4

- Solution
  - Have caches “snoop” on other caches’ reads and writes
  - Writes: Invalidate other cached copies (READ EXCLUSIVE)
  - Reads: Get latest version from other cache, if it has a “dirty” copy (READ SHARED)
  - Coherence at granularity of cache lines
**Cache Coherence Protocol:**

**MSI State Diagram**

<table>
<thead>
<tr>
<th>Processor initiated</th>
<th>Bus initiated</th>
</tr>
</thead>
<tbody>
<tr>
<td>PrRd / PrWr / PrWr</td>
<td>PrWr / --</td>
</tr>
<tr>
<td>M</td>
<td>BusRdX / BusWB</td>
</tr>
<tr>
<td>S</td>
<td>BusRdX / --</td>
</tr>
<tr>
<td>I</td>
<td>PrRd / --</td>
</tr>
</tbody>
</table>

**Abbreviation**

<table>
<thead>
<tr>
<th>Action</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>PrRd</td>
<td>Processor Read</td>
</tr>
<tr>
<td>PrWr</td>
<td>Processor Write</td>
</tr>
<tr>
<td>BusRd</td>
<td>Bus Read</td>
</tr>
<tr>
<td>BusRdX</td>
<td>Bus Read Exclusive</td>
</tr>
<tr>
<td>BusWB</td>
<td>Bus Writeback</td>
</tr>
</tbody>
</table>

**MSI Invalidate Protocol**

- Read obtains block in "shared" even if only cached copy
- Obtain exclusive ownership before writing
  - BusRdX causes others to invalidate
  - If M in another cache, will cause writeback
  - BusRdX even if hit in S promote to M (upgrade)

**A Cache Coherence Example**

<table>
<thead>
<tr>
<th>Proc Action</th>
<th>P1 State</th>
<th>P2 state</th>
<th>P3 state</th>
<th>Bus Act</th>
<th>Data from</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. P1 read u</td>
<td>S --</td>
<td>--</td>
<td>--</td>
<td>BusRd</td>
<td>Memory</td>
</tr>
<tr>
<td>2. P3 read u</td>
<td>S --</td>
<td>S</td>
<td>--</td>
<td>BusRd</td>
<td>Memory</td>
</tr>
<tr>
<td>3. P3 write u</td>
<td>I --</td>
<td>M</td>
<td>--</td>
<td>BusRdX</td>
<td>Memory, P3</td>
</tr>
<tr>
<td>4. P1 read u</td>
<td>S --</td>
<td>S</td>
<td>S</td>
<td>BusRd</td>
<td>P3’s cache</td>
</tr>
<tr>
<td>5. P2 read u</td>
<td>S S I</td>
<td>S S</td>
<td>S</td>
<td>BusRd</td>
<td>Memory</td>
</tr>
<tr>
<td>6. P2 write u</td>
<td>I M I</td>
<td>BusRdX</td>
<td>P2’s cache</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Single writer, multiple reader protocol

---

**Back to the Story ...**
StarPU Implements a MSI Protocol

- Each task has its own local “cache”
  - The copies of the data it is using
- When a task finishes, the data remains
  - Not immediately reclaimed
  - Either in modified or shared state
- Thus, new tasks may have choices
  - Of which of several versions in shared state to use
  - If a task writes, invalidates other copies

What About Hierarchy?

- But StarPU’s data model also has hierarchy
  - May be working on a subset of a larger collection
- How is partitioning/hierarchy incorporated?

What About Hierarchy?

- Answer:
  - Tasks can only use the finest partition available
  - When done with a partition, an explicit release writes modified subsets back to the containing collection
- Thus, tasks work on the leaves of the partitioning hierarchy
  - Creating a new level of partition will cause copies from the coarser to finer level when tasks run
  - A release flushes changes back to coarser level
- Allows MSI protocol to be used more or less unchanged

Legion/Regent

- Legion and Regent have the same issues
  - But allow multiple partitions of the same data
  - And parent/child regions can be used simultaneously
- Add open and close operations to MSI
  - And more states
  - Open: A subtree is opened by a task using a subregion
  - Close: A subtree is closed by copying dirty data back to the root of the subtree
Comparison StarPU & Regent

- StarPU
  - Relatively small, lightweight system

- Regent
  - Much bigger system
  - Why?

What Does StarPU Not Do?

- Two smaller things:

  - Less automatic management
    - Of dependencies
      - Programmer responsible for declaring dependencies
    - Of data coherence
      - Programmer responsible for open/close operations

- Not as aggressive about scheduling ahead
  - Data movement dependencies handled separately
  - Overlaps communication/computation, but task launch not tied to data necessarily being ready

Big Ticket Item #3

- Data model is dense arrays
  - And all examples are dense linear algebra

- No distinct support for unstructured or sparse data

Big Ticket Item #2

- No support for multiple views of data

- One partitioning of the data can exist at a time

- The language of expressible partitions is also limited
  - To things that are very efficient to compute
  - Seems necessary given previous point
Big Ticket Item #1

- Less support for launching large numbers of long-running tasks
- E.g.,
  - Regent’s SPMD transformation
  - Legion’s explicitly parallel features
- Needed to run on large node counts

Summary

- StarPU is a close cousin of Legion/Regent
- Well designed!
- Different decisions due to focus on
  - Single node
  - Simpler data model