A Distributed Dataflow Model for Task-uncoordinated Parallel Program Execution

Lucas A. Wilson$^{1,2}$ and Jeffery von Ronne$^1$

$^1$Dept. of Comp. Sci., Univ. of Texas San Antonio
$^2$Texas Advanced Computing Center, UT Austin

P2S2 Workshop, Minneapolis, MN. Sept. 12, 2014
Outline

• Motivation
  – The problem with scaling out
  – Handling node failure
  – What is needed?

• Task-uncoordinated distributed dataflow
  – The Relentless Execution Model (REM)
  – Cooperative pruning

• Prototype Evaluation
  – Single-node
  – Multi-node
  – Data locality

• Coming Attractions
MOTIVATION
The Problem with Scaling Out

- Clusters are the dominant architecture in parallel computing
- Per-node performance increases according to Moore’s Law*
- Increases in performance are generally achieved by scaling out the number of nodes in the cluster
- Unfortunately, MTTF drops as node count increases

*Your mileage may vary
Handling Node Failure

• MPI has little capability for handling loss of tasks due to hardware failure
  – FT-MPI provides some of this
  – Charm++ performs task rescheduling
• Current Solution – Checkpoint/Restart Files
  – Large amount of I/O
  – Expensive relative to computation
  – As clusters scale out, checkpoint writing will dominate wall time
What is needed?

• Parallel programs need to be able to handle periodic hardware failures
  – Ideally we could elastically add/remove nodes from an executing parallel program
  – We shouldn’t spend more time checkpointing than computing

• Parallel programs need to be able to start up in a reasonable amount of time
  – As we push towards exascale, parallel programs need the ability to start doing useful work before the first failure
What is needed?

- Improve failure resilience
- Reduce startup cost

© L. Wilson, September 2014
The Relentless Execution Model (REM)

• Each processing element executes a search agent which performs dynamic task scheduling based on the availability of data in a distributed, eventually-consistent dictionary

• Dictionary labels are used to provide state information for their associated values

• No two tasks are ever in direct communication; all interaction is done through the dictionary
  – Computational tasks are expendable
  – Can be added/removed at runtime with little-to-no additional cost
The Relentless Execution Model

© L. Wilson, September 2014
REM Tasks

- REM search agents dynamically execute single assignment *tasks* which are chained together with data dependencies

- Why single assignment?
  - Can easily form a directed acyclic graph (DAG) of tasks
  - Allows dynamic scheduling algorithm to ignore parts of the DAG for which output labels have been generated
    - We call this *cooperative pruning*
REM Tasks

- REM takes inspiration from SISAL’s IF1/2 intermediate representation for handling dynamic graphs
  - Conditional branches and loops are contained within hierarchical tasks
  - Top-level graph and sub-graphs are all finite, even if the program has to execute an indeterminate number of tasks
REM Dependency Description

• REM uses a compact data dependency description for storing task and task dependency information
  – Storing DAGs directly can be expensive
  – Many scientific codes have a small number of tasks repeated many hundreds or thousands of times
REM Dependency Description

Dependency Descriptions contain:

1) A finite set $T$ of tasks to be performed
2) A finite set $L$ of labels to be used as keys in the dictionary
3) A set $R \subseteq L$ of result labels whose association with values completes a program’s execution
4) A function $\textit{producer} : L \rightarrow T$ that maps labels to the task which produces the value for that label
5) A function $\textit{requires} : T \rightarrow P(L)$ that maps each task to the labels of the values required before that task can execute
6) A function $\textit{computes} : T \rightarrow (L \leftrightarrow V) \leftrightarrow (L \leftrightarrow V)$ that maps each task to the partial function that it computes
REM Search

• Each search agent performs a backwards, depth-first random walk of a DAG generated by the dependency description
  – Start at the final output(s) of the graph, walk back to the inputs/leaves

• **Compute-by-need**
  – If the label listed as a task dependency has been associated with a value, use that value for the task
  – Otherwise, execute the task responsible for computing that dependency

• **Cooperative Pruning**
  – The contributions of each agent are propagated to other agents through the dictionary, allowing other agents to avoid repeating work

© L. Wilson, September 2014
REM Search (2)

for all $r \in Result$ do
    $SOLVE(r)$
end for

function $SOLVE(label)$
    if $label \notin \text{dom(Dictionary)}$ then
        task $\leftarrow$ $PRODUCER(label)$
        $Missing \leftarrow \{l \mid l \in \text{REQUIRES}(task) \wedge l \notin \text{dom(Dictionary)}\}$
        for all $m \in Missing$ do
            $SOLVE(m)$
        end for
        Inputs $\leftarrow \{(l \mapsto v) \mid (l \mapsto v) \in \text{Dictionary} \wedge l \in \text{REQUIRES}(task)\}$
        Dictionary $\leftarrow$ Dictionary $\cup$ COMPUES(task)(Inputs)
    end if
end function
Example - Cooperative Pruning

Graph has a single output with multiple levels and shared dependencies
Example - Cooperative Pruning (2)
Example - Cooperative Pruning (3)

Agent A chooses to resolve a dependency exclusive to itself

Agent B chooses to resolve a dependency that is shared with A (although it does not know that)
Example - Cooperative Pruning (4)

Agent A looks again at original task, all dependencies now resolved

Agent B looks again at original task, must resolve 1 additional dependency
Example - Cooperative Pruning (5)

Agent A now begins looking at the result for new tasks to compute.

Agent B moves to complete remaining dependency of original task.
PROTOTYPE EVALUATION
Prototype Problem

- Wrote a program to solve the 1-D heat equation using forward in time, central in space (FTCS) approach with a three point stencil
- Prototype system uses memcached for dictionary implementation
- Program was written in StenSAL*, a DSL designed for writing stencil algorithms for REM
- Domain is 5001 elements across
- Simulation runs for 100 time steps
- Initial values are generated by StenSAL tasks

*Submitted to WOSC @ SPLASH 2014

```
task finDiff
creates phi(x)(t) as n
for
  x=1:4999
  t=1:100
using
  phi(x)(t-1) as c
  phi(x-1)(t-1) as l
  phi(x+1)(t-1) as r
code
  n = c+0.0125*
      (l+r-2*c)
end code
end task
```
Prototype Evaluation

• All experiments were performed on Stampede at TACC
  – 6,400-node dual-socket, 8-core “Sandy Bridge” Intel Xeon E5-2650
  – 6,880 Intel Xeon Phi SE10P “Knight’s Corner” coprocessor cards
    • 61 core, 244 thread contexts
  – FDR Infiniband interconnect
    • Fat-tree topology
    • 1.2:1 oversubscription

• Testing the idea of cooperative pruning
  – We should see speedup when increasing number of agents

© L. Wilson, September 2014
Single-Node Speedup

Threads

- Ideal
- Single Node

© L. Wilson, September 2014
Multi-Node Speedup (1 task/socket)
Work Distribution (2 nodes)
Data Distribution & Locality

• On two nodes, data is fairly evenly distributed
  – 49.84% of label/value pairs map to node 1
  – 50.12% of label/value pairs map to node 2

• Unfortunately, data locality is poor
  – 75.18% of all stencil updates required remote data to be accessed
COMING ATTRACTIONS
Coming Attractions

- Testing different hashing functions for improving locality in our simple stencil application
  - Improve scaling by reducing remote accesses
  - Investigating ways of automatically determining appropriate partitioning scheme for a particular dependency description/DAG
- Investigating ways to perform task coalescence on our dependency descriptions to enable vectorization and register reuse
- Evaluating our prototype on the Xeon Phi
  - Testing single Phi, multiple Phi, and mixed Phi/Xeon on Stampede

© L. Wilson, September 2014
Conclusion

• Bigger clusters are coming
  – Shorter MTTF
  – Checkpoint/restart at scale is not feasible

• Distributed dataflow could be used to eliminate explicit task coordination
  – Elastic insertion/removal of computational agents
  – Automatic recovery from data loss

• Initial scaling tests demonstrate the viability of this idea
  – First steps on a long road
    • Lots of work left to be done
Questions?

Lucas A. Wilson
lucaswilson@acm.org

Jeffery von Ronne
vonronne@acm.org