Forsythia: Computing Over Hundreds of Thousands of Potentially Spotty Machines

Zhihao Jia
Joint work with Dennis Fetterly, JP Martin, and Yuan Yu
Motivations

• Data center is hugely under utilized
  • Large numbers of idle cycles
  • Unlikely to be significantly improved very soon
  • Better to exploit transient idle machines
  • Amazon exposes spot instance

• Data-parallel applications needs increasingly more computing resources
  • E.g., Bing’s AIM project requires more than tens of thousands of CPU cores

Is it possible to scale out applications by exploiting large numbers of transient machines?
Two Key Ideas

1. Break computation into standalone work items
   • Can be reassigned and reexecuted at anytime

2. Divide execution into two layers
   • Reliable layer: support Dryad-like dataflow computation, further partition vertices into work items
     • E.g., machines in Dryad/Hadoop clusters
   • Spotty layer: opportunistically execute work items
     • E.g., idle VMs in Azure
     • E.g., Amazon spot instances
Programming Abstraction

• Similar to threadpool
  • Virtual core: transient physical CPU core
    • Pre-emptible, unreliable, and unpredictable
  • Virtual core pool (VCPool): manage collection of virtual cores

• Easy to use

/* Example: compute func(input) on virtual core */
VCPool pool = new VCPool(config);
VCHandler<R> handler = pool.QueueWorkItem<T, R>(func, input);
handler.WaitUntilDone();
/* access result via handler.result */
VCPool/VCWorker Architecture

User code

VCHandler handler = QueueWorkItem(func, input)

Notify Completion

VCPool

PendingWI Queue

InProgressWI Queue

Reliable layer

Spotty layer

Perform Computation

Pipeline IO with computation

VCWorker
Bing AIM project: machine learning application for web search engine

• Goal: mapping queries and docs to a high dimensional space, so that semantically-related docs and queries are nearby in terms of distance

Computations:
1. Queries/docs correlation
   • L32D
2. Vector gradient
   • Query updates
L32D: Correlations between Docs and Queries

Approximate computation: primarily IO intensive, not compute intensive

Accurate computation: primarily compute intensive, holds >95% entire execution time
L32D: Virtual Core Design

Normal QBlocks (14K)
- Query Array
  - QBlocK
  - QBlocK
  - QBlocK
  - QBlocK

Important QBlocks (50)
- Query Array
  - QBlocK
  - QBlocK
  - QBlocK
  - QBlocK

Doc Array
- DBlock | DBlock | DBlock | DBlock

Localy run cheap workload on VCPool machines

Work items (input: 4MB, result: 0.25MB, execution: 3mins)
- Assign work items to VCWorkers
- Assign work items to VCWorkers
- Assign work items to VCWorkers
L32D: Optimizations

- Scalability is bottlenecked by network bandwidth on reliable layer
- **Optimization 1:** Cache frequent input on VCWorkers
  - E.g., Cache all important QBlocks (50) on VCWorkers (~110MB in largest dataset) => shrink amortized input size
- **Optimization 2:** Allow local aggregation on VCWorkers
  - E.g., Final L32D result is the sum of all work items’ results.
  - Locally aggregate L32D result on VCWorkers => shrink amortized result size
- **AIM result:** VCPool network usage is reduced by a factor of 10, with both optimizations
- Generally useful for compute intensive applications
Query Update: Compute Vector Gradient for Query Mapping

Approximate computation: primarily IO intensive, not compute intensive

Normal QBlocks (14K)

Query Array

- QBlocK
- QBlocK
- QBlocK
- QBlocK

Query Gradient Important QBlocks (50)

Work items (input: 4MB, output: 64MB, execution: 4mins)

VCPool needs 0.5 second to receive result of a work item (1Gbps network)

=> receive 480 results in 4 mins without aggregation

=> Local aggregation is critical
Evaluation

• Can single VCPool scale to thousands of VCWorkers?
• How would VCPool perform in presence of failure?
• Can multiple VCPools scale simultaneously?
1. Can Single VCPool Scale to Thousands of VCWorkers?

• Birnam cluster: 8 cores on each machine, 1Gb network bandwidth
• Use AIM as the benchmark
• Start 1,000 VCWorkers on 125 Birnam machines
• Start a single VCPool that tries to manage all 1,000 VCWorkers
• Use both optimizations (caching frequent input, local aggregation)
L32D Throughput

Number of finished work items

Time since start (minutes)

VC Runtime  Ideal case (1000 cores)
Query Update Throughput

![Graph showing query update throughput with time since start (minutes) on the x-axis and number of finished work items on the y-axis. The graph includes two lines: one for VC Runtime and another for the Ideal case (1000 cores).]
2. How would VCPool perform in presence of failure?

• Introduce failure model:
  • Divide time line into equivalent time slots, with length $T$
  • During a time slot, a VCWorker may stop working/become unavailable with probability $P$
  • VCWorker independently makes decision during each time slot

• Experiments:
  • Single VCPool manages all 1,000 VCWorkers
  • $T = 6$mins, $P = 20\%$
L32D Throughput with Failure

- Number of finished work items vs. Time since start (minutes)

- Graph showing VC Runtime, Ideal Case (1000 cores), and Ideal Case (800 cores)
3: Can Multiple VCPools scale simultaneously?

• Start 10 machines in reliable layer. Each creates a VCPool and manages 100 VCWorkers.
• 10 VCPools assign work items, receive results simultaneously.
Overall L32D Throughput with 10 VCPools

Number of finished work items vs Time since start (minutes)

- VC runtime overall
- Ideal case (1000 cores)
Conclusion

• Scalability test (Experiment 1): a single VCPool can scale to thousands of VCWorkers with negligible overhead
• Failure test (Experiment 2): VCPool can make progress and scale with presence of failure
• Multiple VCPools test (Experiment 3): multiple VCPools can scale simultaneously
• Expectation: hundreds of machines in reliable layer could potentially manage hundreds of thousands of transient machines.
Future Work

• Make AIM project scale to hundreds of thousands of transient machines
• Programming API for high-level operations
  • E.g., automatically support relational operations (Select) in DryadLINQ
• Introduce virtual core into Azure.
Thanks