Optimizing Chapel for Intra-Node, Multi-Core Environments

Richard Johnson and Jeffrey K. Hollingsworth
Department of Computer Science, University of Maryland, College Park

CHIUW
05/27/2016
Motivation: Building a better Chapel

- Evaluate how well Chapel performs in practice.
  - Comparison of Chapel benchmark performance against implementations in competitive parallel frameworks.
  - Identify opportunities to improve language performance.

- Goals: Investigating techniques to
  - Improve development practices for Chapel programmers.
  - Automate solutions that could be incorporated into future versions of the Chapel compiler and runtime framework.

- We will focus on single-locale environments.
Strategy

- Use benchmarks
  - Represent real world scientific computing applications
  - Embodies different usage of language features

- Performance tuning
  - Profile benchmarks to identify bottlenecks in performance.
  - Analyze performance gaps between parallel frameworks.

- Determine where changes are needed to close gaps.

- Generalize the lessons learned.
  - Improvement over original and competitive benchmark
  - Impact across other Chapel benchmarks
LULESH Overview and Pitfalls

- LULESH is a 'shock hydro' parallel benchmark designed for hydrodynamics calculations.

- Large array declarations inside subroutines:
  - Translate into large heap allocation requests.
  - Write operations are performed to set all elements to zero.
  - Occurs each time the function is invoked.

```
Lulesh.chpl (1695 lines)
CalcHourglassControlForElems()

proc CalcHourglassControlForElems(determ) {
    var dvdx, dvdy, dvdz, x8n, y8n, z8n: [Elems] 8*real;

    forall eli in Elems {
        ...
    }
    ...
}
```

18.8% of the wall time is spent on one line of code in the sequential part of the program.
LULESH Insights

- **Hoisting**
  - Store recurring requests of large local allocation for reuse.
  - Additionally store allocations of all compiler generated metadata structures related to each memory allocation.

- **Conservative Memory Initialization**
  - For each allocation, does there exist an entry in the subsequent code that is read prior to being set explicitly?
  - Static analysis: determine when to invoke `calloc` vs. `malloc` and `memset` for memory reuse in generated code.

- Provide optional compiler support for language feature similar to static in C.
  - Avoid having to use globals.

```plaintext
proc foo() {
  persistent var a: [dom] int;
  ...
}
LULESH Performance

- Speedup over Original in Chapel:
  - Allocation Hoising (AH): 1.10
  - AH + Conservative Memory Init (CMI): 1.54
  - Metadata Hoisting (MH) + AH + CMI: 3.01

- Speedup over OpenMP in C++:
  - AH + CMI: 0.71
  - Metadata Hoisting (MH) + AH + CMI: 1.96
MiniMD Overview, Pitfalls, and Insights

- **Mini** parallel benchmark for **Molecular Dynamics**
- Avoid repetitive mapping from one domain to another when iterating over nested loops.

### UpdateFluff()

<table>
<thead>
<tr>
<th>Original</th>
<th>Optimized</th>
</tr>
</thead>
</table>
| forall (P,D,S) in zip (PosOffset, Dest, Src) {  
  Pos[D] = Pos[S];  
  Count[D] = Count[S];  
  
  // offset positions  
  forall d in D do  
    Pos[d][1..Count[d]] += P;  
| forall (P,D,S) in zip (PosOffset, Dest, Src) {  
  forall (d, s) in zip (D,S) do {  
    Pos[d] = Pos[s];  
    Count[d] = Count[s];  
    for i in 1..Count[d] do {  
      Pos[d][i] += P;  
  } |

- Remove unnecessary autoCopy / autoDestroy calls
  - Found inside ‘coforall_fn_chpl#’ loops generated from the parallel loops of ‘Build Neighbors’ and ‘ForceLJ compute’
MiniMD Performance

Elapsed Time (sec)

<table>
<thead>
<tr>
<th></th>
<th>Elapsed Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td></td>
</tr>
<tr>
<td>Optimized</td>
<td></td>
</tr>
<tr>
<td>OpenMP</td>
<td></td>
</tr>
</tbody>
</table>

- Remaining
- Update
- ForceLJ Compute
- Build Neighbor
SSCA#2 Overview and Implementations

- **Scalable Synthetic Compact Applications #2**
  - Generates weighted, directed multigraph.
  - Performs approximate betweenness centrality (BC).

- **Chapel vs. OpenMP version of SSCA#2**
  - Different approaches to betweennessCentrality()
  - Developed ports to achieve a more fair comparison.

- **Each version of the benchmark was ported to the other framework respectively.**
  - Algorithm I: Chapel benchmark
  - Algorithm II: OpenMP benchmark
SSCA#2 Pitfalls and Insights: Alg. I

- Algorithm I was not optimized for single-locale.
  - One task private variable (TPV) data structure per core instead of per locale.
- Managing parallel redundancies in nested loops.
- User specific thread initialization for nested loops.
  - Removing the need for task private data management could improve parallel loop performance by 12% or more.
- Selectively disable redundant memory initializations ‘init_elts#’ found in ‘initialize#’ in the generated code.
  - Shown to improve performance of other benchmarks too.
SSCA#2 Pitfalls and Insights: Alg. II

- Initial port into Chapel performed 4.9x slower
  - Overhead of parallelization in BC: 46% of overall BC time
  - Up to 54.5% of parallel time in BC was spent on variable synchronizations (locks)
  - Fluctuating number of iterations in BC inner loops
    - Non-uniform workload distribution

- Developed a proxy to model parallelization of BC.
  - Overhead of parallel loops nested inside sequential loops
  - Compare uniform and non-uniform workload performance
  - Comparisons between parallel frameworks.
SSCA#2 Insights: Alg. II

- BC proxy lessons learned:
  - Non-uniform workloads
    - Chapel: 4.7x slower, OpenMP: unaffected
    - Chapel performance on par with OpenMP (static, 1 scheduler)
  - No usage of #pragma omp parallel: 28x slower
  - Chapel: parallelizing outer loop instead: 15% speed up

- Application towards BC in Chapel port (Alg. II)
  - Parallelize the outermost loop over starting vertices:
    - Reduces the sequential parts of BC and parallel overhead.
    - Allows for the removal of most synchronization variables.
SSCA#2 Performance

<table>
<thead>
<tr>
<th>Problem Size</th>
<th>Algorithm I</th>
<th>Algorithm II</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>2.0 OpenMP Port, 1.5 Chapel Opt</td>
<td>2.25 OpenMP, 1.75 Chapel Opt Port</td>
</tr>
<tr>
<td>16</td>
<td>1.5 OpenMP Port, 1.0 Chapel Opt</td>
<td>2.0 OpenMP, 1.0 Chapel Opt Port</td>
</tr>
<tr>
<td>20</td>
<td>1.0 OpenMP Port, 0.5 Chapel Opt</td>
<td>1.5 OpenMP, 1.0 Chapel Opt Port</td>
</tr>
</tbody>
</table>
CLOMP Overview and Pitfalls

- Coral Collaboration Benchmark Codes
- **CLOMP:** *C* version of *Livermore OMP* benchmark
  - Skeleton benchmark for measuring the overhead of different OpenMP primitives.
  - Sequential loop test: serial
  - Parallel loop tests: static, dynamic, and manual
- Chapel benchmark
  - Ported serial and a generic version of parallel loop test.
  - Chapel does not allow for explicit thread control.
- Redundant memory initializations; Memory structure
CLOMP Performance

Problem Size

Parts

Zones per Part

12
640000

65536
10

1024
640000

65536
6400

Elapsed Time (sec)

- OpenMP (linked lists)
- Chapel Port (arrays)
- OpenMP (matrix)
- Chapel Port (matrix)
# Overlap and Impact of Bottlenecks

<table>
<thead>
<tr>
<th>Degradation</th>
<th>LULESH</th>
<th>MiniMD</th>
<th>SSCA#2</th>
<th>CLOMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reoccurring local allocations</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread / task private allocations</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adaptive memory reset</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Redundant memory init_elts#</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>X</td>
</tr>
<tr>
<td>Redundant autoCopy / autoDestroy</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Redundant parallelism</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Domain remapping overhead</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Application bottleneck</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Memory structure</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>

**X:** Major impact,  **x:** Minor impact
Conclusion

Performance gain over:

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Original Chapel</th>
<th>OpenMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>LULESH</td>
<td>3.0x</td>
<td>2.0x</td>
</tr>
<tr>
<td>MiniMD</td>
<td>5.3x</td>
<td>0.4x</td>
</tr>
<tr>
<td>SSCA#2 (I)</td>
<td>6.3x</td>
<td>On par</td>
</tr>
<tr>
<td>SSCA#2 (II)</td>
<td>7.9x</td>
<td>1.6x</td>
</tr>
<tr>
<td>CLOMP</td>
<td>4.8x</td>
<td>1.7x</td>
</tr>
</tbody>
</table>

Future work

- Explore Chapel performance and develop optimization strategies for inter-node (multi-locale) environments.
  - Task delegation and memory localization over PGAS
  - Communication access patterns
  - Remote prefetch and caching
- Automate optimizations in Chapel reference compiler.