An Evaluation of UPC++ Using Distributed Parallel Graph Algorithms

Presenter: Alex Ding
Mentor: Yan Gu
Special Thanks to: Julian Shun
Parallel Computing

**Serial**
- Program
  - Task N
  - Task 3
  - Task 2
  - Task 1
- Processor

**Parallel**
- Program
  - Task 1
  - Task 2
  - Task 3
  - Task 4
- Processor
  - Processor
  - Processor
  - Processor

Sends Instructions
Parallel Computing

Benefits
- Speed *(only way)*
- Scalability
- Real world is parallel

Challenges
- Synchronization
- Work distribution
- Communication overhead
- Hard to debug
Shared Memory vs. Distributed Memory

- **Shared Memory**
  - Memory shared by all processes
  - Communicate through shared memory
  - Like our laptop: one shared memory block for multiple cores

- **Distributed Memory**
  - No shared memory
  - Connected together by network
  - Often on large network of computers, each with its own memory
Shared Memory vs. Distributed Memory

- Easy to program (data is shared)
- Fast communication
- Low scalability
  - Processors
  - Memory

- Hard to program
- Latency in message passing
- High scalability
  - Processors
  - Memory
UPC++: Partitioned Global Address Space
[Zheng et al., IPDPS 14]

- An attempt to unify the two models

![Global address space diagram](https://bitbucket.org/berkeleylab/upcxx/downloads/upcxx-guide-2019.9.0.pdf)

- Memory is distributed, but UPC++ exposes global address interface
- Handles message passing

UPC++ (cont’d) and Motivations

UPC++’s Goals

- Easy programming
- Take advantage of scalability of distributed memory system
- Allows programmer to use the same API for local and non-local data
  - Handles details of reading/writing non-local data

Our Question: Promises Delivered?

1. How scalable? (Overhead?)
2. How fast?
3. How easy to use? (Does it feel distributed or shared when coding?)
Our Work

UPC++ vs. shared memory library (OpenMP): Scaling & Speed

1. Implemented common graph algorithms on UPC++ and OpenMP
2. Ran tests on a single-node, multi-core system
   a. Varying core counts
   b. Real-world and randomly-generated graphs
3. Implemented optimizations (significant work)
   a. Dynamic top-down/grounds-up decision based on frontier density
   b. Different graph partition methods to maximize locality and minimize communication
Experiment Setup

- Single node, multi-core system on AWS
  - C5.18xlarge instance (36 Intel Xeon cores, 144 GBs memory)
- Breadth-first-search implemented on UPC++ and OpenMP
- Graph: ego-Gplus (social circles from Google Plus)
  - 107,614 nodes, 13,673,453 edges, diameter 6
  - Retrieved from Stanford Network Analysis Project
- Compare runtime of program on UPC++ and OpenMP with different numbers of cores used
- Goal is to explore
  - Scaling
  - UPC++’s overhead compared to OpenMP
Results

- Overhead on single node (2.82x)
- Great scaling on UPC++
  - 2x cores ~ ½ runtime
- Bad scaling on OpenMP
  - Overhead takes over
Other Results

- Other algorithms include: Bellman-Ford, Connected-Components, PageRank
- Real graphs
  - Range of overhead: [0.66, 6.9]
  - Consistently good scaling on UPC++

<table>
<thead>
<tr>
<th>Graph</th>
<th>Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ego-Facebook</td>
<td>4,039</td>
</tr>
<tr>
<td>ego-Twitter</td>
<td>81,306</td>
</tr>
<tr>
<td>ego-Gplus</td>
<td>107,614</td>
</tr>
<tr>
<td>com-Youtube</td>
<td>1,134,890</td>
</tr>
<tr>
<td>com-Orkut</td>
<td>3,072,441</td>
</tr>
</tbody>
</table>

Random graphs
- 1,000 - 1,000,000 nodes
- 1-100 edges per node
Conclusions

- Easy to work with
- Have to code with locality in mind to achieve good results
- Manageable local overhead
  - Communication has latency, but that depends on hardware
  - Given the advantages of distributed parallelism, overhead is acceptable
- Highly scalable
Future Work

- Run tests on multi-node machines (in progress)
  - Waiting on supercomputer hours
- Optimize codebase for fast code
  - Implement the Gemini system [Zhu et al., OSDI 16]
  - Compare with state-of-the-art distributed graph algorithms
Questions?