Outline

• Introduction
  – PGAS
  – Chapel
  – Motivation
• Related Studies
• Benchmarks
  – Versions
• Evaluation
• Conclusion
Introduction - PGAS

Actual

![Diagram of actual processing units with base addresses and global pointer]

Abstraction

![Diagram of abstracted processing units]

Engin Kayraklioglu - CHIUW 2016
PGAS Access

\[
\text{const DistDom} = \{1..100\} \text{ dmapped SomeDist()}; \\
\text{var distArr: [DistDom] int;} \\
\text{writeln(distArr[14]);}
\]
## Access Types in PGAS

<table>
<thead>
<tr>
<th></th>
<th>Local</th>
<th>Remote</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-distributed</td>
<td>OK</td>
<td>?</td>
</tr>
<tr>
<td>distributed</td>
<td>Locality Check Fine Grain</td>
<td>Locality Check Fine grain</td>
</tr>
</tbody>
</table>
Chapel

• Emerging Partitioned Global Address Space language
• Carries inherent PGAS access overheads
• Programmer can mitigate overheads
• How?
• At what cost?
## PGAS Access Types in Chapel

<table>
<thead>
<tr>
<th></th>
<th>Local</th>
<th>Remote</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-distributed</td>
<td>Fast</td>
<td>N/A</td>
</tr>
<tr>
<td>distributed</td>
<td>Locality Check</td>
<td>Fine grain</td>
</tr>
</tbody>
</table>

```chapel
const ProblemSpace = {0..#N, 0..#N};
var arr : [ProblemSpace] int;
// ... some code here ...
writeln(arr[i, j]);
```

```chapel
const DistProblemSpace = ProblemSpace dmapped Block(ProblemSpace);
var distArr: [DistProblemSpace] int;
// ... some code here ...
writeln(distArr[i, j]);
```
How to Avoid Overheads
local statement

forall (i,j) in distArr.domain do
  // ... find iKnowItsLocal ...
  if iKnowItsLocal then
    local writeln(distArr[i, j]);
  else
    writeln(distArr[i,j]);

Naive

var localDom = {0..#SIZE/4, 0..#SIZE};
var remoteDom = {SIZE/4..SIZE, 0..#SIZE};
local forall (i,j) in localDom do
  writeln(distArr[i, j]);
forall (i,j) in remoteDom do
  writeln(distArr[i, j]);

Better
How to Avoid Overheads
Bulk Copy

```c
var privCopy: [ProblemSpace] int;
var copyDomain = {15..25,15..25};
privCopy[copyDomain] = distArr[copyDomain];
```
Motivation - Contribution

- Applications that have well-structured accesses to distributed data
  - Explicit domain manipulation
    - distArr.localSubdomain()
    - Other domain manipulation methods in language
  - Affine transformation;
    - Locality check avoidance
    - Bulk copy

- Performance vs productivity analysis of such transformations in application level
Relevant Related Work

PGAS

• El-Ghazawi et al., “UPC performance and potential: A NPB experimental study”, SC02
  – Similar study on UPC with NPB
  – Comparable performance to MPI with higher productivity
• Chen et al., “Communication optimizations for fine-grained UPC applications”, PACT05
  – Berkeley UPC compiler optimizations
  – Redundancy elimination, split-phase communication, message coalescing
• Alvanos et al., “Improving performance of all-to-all communication through loop scheduling in PGAS environments” ICS13
  – Inspector/executor logic for runtime coalescing
  – 28x speedup in UPC
• Serres et al., “Enabling PGAS productivity with hardware support for shared address mapping: A UPC case study”, TACO16
  – Hardware solution for wide pointer arithmetic
  – Better performance then hand optimization
Relevant Related Work

Chapel

- Hayashi et al., "LLVM-based communication optimizations for PGAS programs", LLVM15
  - Language-agnostic, LLVM based optimizations
  - Remote access aggregation, locality analysis, runtime coalescing
  - Up to 3x performance
- Kayraklioglu et al., "Assessing Memory Access Performance of Chapel through Synthetic Benchmarks", CCGRID15
  - Locality check avoidance gains up to 35x in random accesses
- Ferguson et al., "Caching Puts and Gets in a PGAS Language Runtime", PGAS15
  - Software cache for remote data
  - Spatial and temporal locality
  - 2x improvement
Benchmarks

• Sobel
  – $2^{13} \times 2^{13}$
• MM
  – $C = A \times B^T$, $2^9 \times 2^9$
• MT
  – $2^{11} \times 2^{11}$
• 3D Heat diffusion
  – 3D, repetitive stencil
  – $2^8 \times 2^8 \times 2^8$
• STREAM
  – Full set: copy, scale, sum, triad
  – Bandwidth perspective
Versions

- O0
  - Simplest implementation
  - Highest programmer productivity
  - Very intuitive
- O1
  - Locality check avoidance for local accesses
  - Added programming complexity
- O2
  - Bulk copy
  - Added programming complexity (generally)
Performance Evaluation

- **George - Cray XE6/XK7**
  - 56 nodes, dual Magny Cours with 12 hw threads each
  - Chapel version 1.12.0
  - qthreads, GasNET
  - 1-32, power-of-two nodes
Results
Sobel

[Graphs showing execution time, improvement over O0, and scalability as a function of the number of locales.]
Results
Sobel - Detail

![Graphs showing execution time ratio vs number of locales for Sobel operation with different time segments labeled: O1 Total, O1 Remote, O1 Local, O2 Total, O2 Copy, O2 Compute.](image)
Results

MM
Results
MM - Detail

![Graphs showing execution time ratio vs. number of locales for different operations.](image)
Results
MT

![Graph 1: Execution Time (s) vs. Number of Locales]

![Graph 2: Improvement Over O0 vs. Number of Locales]

![Graph 3: Scalability vs. Number of Locales]
Results
MT - Detail
Results
3D Heat Diffusion
Results
3D Heat Diffusion - Detail

![Graph 1: Execution Time Ratio vs. Number of Locales for O1 and O2]

- O1 Total
- O1 maxrem
- O1 maxloc

![Graph 2: Execution Time Ratio vs. Number of Locales for O2 maxcopy and O2 maxcomp]
Results
Stream Scale

![Graph 1: Memory Bandwidth vs. Number of Locales](image1)

- Blue line: Oprom
- Green line: O0
- Red line: O1

![Graph 2: Improvement vs. Number of Locales](image2)

- Blue line: Oprom
- Green line: O0
- Red line: O1
Results
Stream Triad

Memory Bandwidth (GB/s)

Number of Locales

Improvement

Number of Locales
Productivity Evaluation

• What comprises “productivity”
  – How fast you learn?
  – How fast you implement?
  – How maintainable?
  – How correct?
• Qualitative, very subjective
• List of measures covered;
  – # lines of code,
  – # arithmetic/logic operations
  – # function calls
  – # loops
# Productivity Evaluation

<table>
<thead>
<tr>
<th></th>
<th>Sobel</th>
<th>MM</th>
<th>MT</th>
<th>Heat Diff</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>O0</td>
<td>O1</td>
<td>O2</td>
<td>O0</td>
</tr>
<tr>
<td>LOC</td>
<td>1</td>
<td>13</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>A/L</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Func</td>
<td>2</td>
<td>17</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Loop</td>
<td>1</td>
<td>5</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>X</td>
<td>1.0</td>
<td>1.8</td>
<td>3.8</td>
<td>1.0</td>
</tr>
</tbody>
</table>

- O0 is highly productive
  - <10 LOC for all
- O2 seems more productive compared to O1
  - Memory footprint of O2 is not studied
Possible Directions

• More breadth
  – Sparse arrays
  – Task parallelism
  – Different applications

• More depth
  – Low-level routines, extern C functions
  – A productivity model
  – … vs Memory vs power
Recap

• PGAS access characteristics
• Application-level optimizations
• Performance vs Productivity

• Compile time affine transforms
• Runtime prefetching
Thank you

engin@gwu.edu
Backups
Productivity Evaluation
Sobel

- **01**
  - Local subdomain queries
  - Rectangular domain methods

<table>
<thead>
<tr>
<th></th>
<th>Sobel</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>00</td>
</tr>
<tr>
<td>LOC</td>
<td>1</td>
</tr>
<tr>
<td>A/L</td>
<td>0</td>
</tr>
<tr>
<td>Func</td>
<td>2</td>
</tr>
<tr>
<td>Loop</td>
<td>1</td>
</tr>
<tr>
<td>X</td>
<td>1.0</td>
</tr>
</tbody>
</table>

- **02**
  - bulk copy of local subdomain expanded by 1
Productivity Evaluation

**MM**

- **O1**
  - Subdomains are calculated arithmetically

- **O2**
  - Manual replication

### Table

<table>
<thead>
<tr>
<th></th>
<th>MM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>O0</td>
</tr>
<tr>
<td>LOC</td>
<td>4</td>
</tr>
<tr>
<td>A/L</td>
<td>2</td>
</tr>
<tr>
<td>Func</td>
<td>0</td>
</tr>
<tr>
<td>Loop</td>
<td>2</td>
</tr>
<tr>
<td>X</td>
<td>1.0</td>
</tr>
</tbody>
</table>

### Diagram

\[ X \cdot O1 \cdot O2 \]

Subdomains are calculated arithmetically.

Manual replication.

X