Shoal: smart allocation and replication of memory for parallel programs

Stefan Kaestle, Reto Achermann, Timothy Roscoe, Tim Harris

ETH Zurich  $Oracle Labs Cambridge, UK
Problem

• Congestion on interconnect
• Load imbalance of memory controllers
• Performance of parallel application depends on memory allocation

Suboptimal allocation $\rightarrow$ bad performance
Shoal

• Memory abstraction: Arrays
• Statically derive access patterns from code
• Choose array implementation at runtime

• Reduces runtime:
  – 4x over naïve memory allocation
Example: PageRank

runtime [s]

<table>
<thead>
<tr>
<th>cores</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>150</td>
<td>75</td>
<td>37.5</td>
<td>18.75</td>
</tr>
</tbody>
</table>

8x8 AMD Opteron 6378 Bulldozer
4 Sockets
512 GB RAM

SNAP Twitter graph
41M nodes
1468M edges
size in RAM: 2.5 GB
Example: PageRank

- Runtime [s] vs. number of cores (8, 16, 32, 64)
- Comparison between perfect scalability and single-node performance
- 8x8 AMD Opteron 6378 Bulldozer
  - 4 Sockets
  - 512 GB RAM
- SNAP Twitter graph
  - 41M nodes
  - 1468M edges
  - Size in RAM: 2.5 GB
Problem: implicit allocation

```c
void *data = malloc(SIZE);
memset(data, 0, SIZE);
```

- **Implicit Linux policy** on where to allocate memory
- First touch → all memory on same NUMA node
What we would like to do?

• Partitioning
  – Split working space, put on different nodes
• Replication
  – Copy array
  – Updates: consistency

→ Reduce load-imbalance
→ Localizes access → reduces interconnect traffic

• DMA
• 2M/1G pages
What we have today:

• Explicit placement of memory
  – libnuma

• Advise Kernel about use of memory region
  – madvise
SHOAL
Exploiting DSLs

• High-level API
• Efficient parallelization
• widely used
  – Machine learning
  – Signal processing
  – Graph processing

Idea: derive access patterns
Green Marl: graph storage

- **nodes**
  - 0
  - 1
  - 3
  - 5

- **edges**
  - 2
  - 0
  - 2
  - 0
  - 1

- **r_nodes**
  - 0
  - 1
  - 3
  - 5

- **r_edges**
  - 1
  - 2
  - 2
  - 0
  - 1
Overview: Green Marl

- high-level program (e.g. Pagerank)
- high-level compiler
- low-level code (e.g. C++)
- compiler (CC)
- program
Modifications to Green Marl

- **high-level program** (e.g. Green Marl)
  - high-level compiler
  - analysis
  - low-level code (e.g. C++)
    - memory abstraction
  - compiler (e.g. gcc)
  - program
  - library
1) Array abstraction

- high-level program (e.g. Pagerank)
  - high-level compiler analysis
  - low-level code (e.g. C++)
  - memory abstraction
  - compiler (e.g. gcc)
  - program library
Array abstraction

• get() and set()
• copy_from(arr) and init_with(const)
• array_malloc(size, access_patterns)
Shoal’s access patterns

- Read-only
- Sequential
- Random
- Indexed

Indexed:
for (i=0; i<SIZE; i++) {
    foo(arr[i]);
}
→ sequential + local
2) Compiler

- Use Shoal memory abstraction
- Extract memory access patterns
Derivation of access patterns

Foreach (t: G.Nodes) {
  Double val = k * \text{Sum}\left( nb: t.\text{InNbrs}\right)\{nb.rank / nb.\text{OutDegree()}\} ;
  diff += | val - t.pg_rank |;
  t.pg_rank <= val @ t;
}
Derivation of access patterns

Foreach (t: G.Nodes) {
    Double val = k * Sum(nb: t.InNbrs) {
        nb.rank / nb.OutDegree();
    }
    diff += | val - t.pg_rank |;
    t.pg_rank <= val @ t;
}
Green Marl: graph storage

- **nodes**: 0, 1, 3, 5
- **edges**: 2, 0, 2, 0, 1
- **r_nodes**: 0, 1, 3, 5
- **r_edges**: 1, 2, 2, 0, 1
Green Marl: graph storage

Nodes:
- 0
- 1
- 3
- 5

Edges:
- 2
- 0
- 2
- 0
- 1

Ranks:
- 0.2
- 0.1
- 0.2

r_nodes:
- 0
- 1
- 3
- 5

r_edges:
- 1
- 2
- 2
- 0
- 1
Deriving access patterns

```
Sum(nb: t.InNbrs) {
    // ...
};
```

Operation: `InNbrs` - neighbors of node `t`:

- `s = r_nodes[t]`
- `e = r_nodes[t+1]-1`
- `nb = [r_edges[x] for x in (s..e-1)]`

---

<table>
<thead>
<tr>
<th>nodes</th>
<th>0</th>
<th>1</th>
<th>3</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>edges</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>r_nodes</th>
<th>0</th>
<th>1</th>
<th>3</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>r_edges</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

ranks: 0.2, 0.1, 0.2
Deriving access patterns

\[ \text{Sum(nb: } t.\text{InNbrs}) \{ \]
\[ // \ldots \]
\[ \} ; \]

Operation: \text{InNbrs} - neighbors

\[
s = r\text{\_nodes}[t] \\
e = r\text{\_nodes}[t+1]-1 \\
nb = [r\text{\_edges}[x] \text{ for } x \text{ in } s..e-1] \\
\]

\[ \text{nodes} \]

\[
\begin{array}{cccc}
0 & 1 & 3 & 5 \\
\end{array}
\]

\[ \text{r\_nodes} \]

\[
\begin{array}{cccc}
0 & 1 & 3 & 5 \\
\end{array}
\]

\[ \text{edges} \]

\[
\begin{array}{cccc}
2 & 0 & 2 & 0 \\
1 & 2 & 0 & 1 \\
\end{array}
\]

\[ \text{r\_edges} \]

\[
\begin{array}{cccc}
1 & 2 & 2 & 0 \\
0 & 1 & 1 & 1 \\
\end{array}
\]
Deriving access patterns

$$\text{Sum}(\text{nb: t.InNbrs}) \{ \text{nb.rank} / \text{nb.OutDegree()} \};$$

Operation: $\text{rank}$ - rank of neighbor $\text{nb}$:

$$\text{rnk}\_\text{tmp} = \text{rank}[^\text{nb}]$$

- random
- read-only

nodes

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>3</th>
<th>5</th>
</tr>
</thead>
</table>

edges

| 2 | 0 | 2 | 0 | 1 |

ranks

| 0.2 | 0.1 | 0.2 |

nodes

| 0 | 1 | 3 | 5 |

r_nodes

| 0 | 1 | 3 | 5 |

r_edges

| 1 | 2 | 2 | 0 | 1 |
Deriving access patterns

\[ \text{Sum}(\text{nb}: \text{t.InNbrs}) \{ \text{nb.rank / nb.OutDegree()} \} ; \]

Operation: `OutDegree()` - of neighbor \( w \):

\[ \text{nodes}[nb+1] - \text{nodes}[nb] \]

- random
- read-only

<table>
<thead>
<tr>
<th>nodes</th>
<th>r_nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 3 5</td>
<td>0 1 3 5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>edges</th>
<th>r_edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 0 2 0 1</td>
<td>2 1 2 0 1</td>
</tr>
</tbody>
</table>
3) Runtime

- high-level program (e.g. Pagerank)
  - high-level compiler
    - analysis
    - low-level code (e.g. C++)
      - memory abstraction
        - compiler (e.g. gcc)
          - program
            - library
Runtime: Choice of arrays

- **Start**: indexed?
  - y: indexed
  - n: partitioned

- **Partitioned**: read-only?
  - n: partitioned
  - y: distributed

- **Distributed**: fits all nodes?
  - n: H/W characteristics
  - y: replicated

- **Replicated**
Runtime: Choice of arrays

- Start
  - Indexed?
    - No
      - Read-only?
        - No
          - Partitioned
        - Yes
          - Replicated
    - Yes
      - Replicated
  - Yes
    - Replicated

- Hardware characteristics:
  - #nodes
  - #CPUs
  - Size RAM / node
  - DMA
  - Huge/large page
Shoal workflow

- high-level program
  (e.g. Pagerank)

- high-level compiler
  analysis

- low-level code (e.g. C++)
  memory abstraction

- access patterns

- hardware characteristics

- compiler
  (e.g. gcc)

- program
  library
Alternative approaches

• Search-based auto-tuning
• HW page migration
• Carrefour: Online analysis of access patterns
  – Simon Fraser University, ASPLOS 2013
  – Performance counters to monitor accesses
  – Dynamically migrate and replicate pages
EVALUATION
Single node allocation

![Diagram of single node allocation]

- Runtime [s]
- Cores

- Single-node allocation

![Bar chart showing runtime for different core counts]

- 8 cores: 150 [s]
- 16 cores: 200 [s]
- 32 cores: 250 [s]
- 64 cores: 300 [s]
Distribute memory

```c
#pragma parallel omp for
for (int i=0; i<SIZE; i++)
data[i] = 0;
```

Problem: this is OS / HW specific

Default Green Marl behavior
Carrefour: reactive tuning
Shoal

- Knowledge of access patterns
- Replication
- Distribution
- Large pages (2M)
Performance breakdown

- **single-node**
- **partitioning**
- **partitioning + replication**
- **partitioning + replication + 2M pages**
Conclusion

• Memory abstraction, arrays
• Compiler analysis $\rightarrow$ derive access patterns
• Runtime library $\rightarrow$ selects implementation
• Works well with domain specific languages
• Also: support for manual annotation
  – Too complex, too dynamic $\rightarrow$ Online

• Public release next week