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

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

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
SHOAL
Exploiting DSLs

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

Idea: derive access patterns
Green Marl: graph storage

- State of the Art graph analytics framework
- Stores graph in adjacency arrays
- Provides high-level graph functionality
  - Node/Edge/Neighbors iterator
  - HasEdgeTo, OutDegree, Exists
  - Node/Edge attributes
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;
}
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**

Set() / Get() interface
2) Compiler

1. Use memory abstraction
2. Extract access patterns

- high-level program (e.g. Pagerank)
- high-level compiler analysis
- low-level code (e.g. C++)
- memory abstraction
- compiler (e.g. gcc)
- program library
Shoal’s access patterns

- Read-only
- Sequential
- Random
- Indexed
Shoal’s access patterns

- Read-only
- Sequential
- Random
- Indexed

indexed:
for (i=0; i<SIZE; i++) {
    foo(arr[i]);
}
→ sequential + local
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
Runtime: Choice of arrays

start

indexed?

partitioned
Runtime: Choice of arrays

- **Indexed?**
  - Yes: Partitioned
  - No: Read-only?
    - Yes: Distributed
    - No: Partitioned
Runtime: Choice of arrays

start

indexed?

read-only?

fits all nodes?

partitioned

distributed

replicated
EVALUATION
Single node allocation

![Diagram showing single node allocation with a bar chart illustrating the runtime in seconds for different numbers of cores. The chart shows that the runtime increases as the number of cores increases.]
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

<table>
<thead>
<tr>
<th>Cores</th>
<th>Single-node</th>
<th>Distributed</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>150</td>
<td>200</td>
</tr>
<tr>
<td>16</td>
<td>175</td>
<td>220</td>
</tr>
<tr>
<td>32</td>
<td>200</td>
<td>250</td>
</tr>
<tr>
<td>64</td>
<td>225</td>
<td>275</td>
</tr>
</tbody>
</table>
Shoal

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

• Memory abstraction, arrays
• Compiler analysis → derive access patterns
• Runtime library → selects implementation
• Works well with domain specific languages
• Also: support for manual annotation

• Public release - github.com/libshoal
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

edges

r_nodes

r_edges

ranks

0.2 0.1 0.2
Deriving access patterns

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

Operation: InNbrs - neighbors of node $t$:

- $s = r_{nodes}[t]$
- $e = r_{nodes}[t+1]-1$
- $nb = [r_{edges}[x] for x in [s..e-1]]$

- indexed
- read-only
- sequential
- read-only

Nodes:

```
| 0 | 1 | 3 | 5 |
```

Edges:

```
| 2 | 0 | 2 | 0 | 1 |
```

r_n nodes:

```
| 0 | 1 | 3 | 5 |
```

r_edges:

```
| 1 | 2 | 2 | 0 | 1 |
```

ranks:

```
| 0.2 | 0.1 | 0.2 |
```
Deriving access patterns

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

Operation: `rank` - rank of neighbor \(nb\):

\[ rnk\_tmp = \text{rank}[nb] \]

- random
- read-only

<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>ranks</th>
<th>0.2</th>
<th>0.1</th>
<th>0.2</th>
</tr>
</thead>
</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>
Deriving access patterns

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

Operation: \texttt{OutDegree()} - of neighbor \( w \):

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

- random
- read-only

\[
\begin{array}{c}
0 & 1 & 3 & 5 \\
\text{nodes} & 2 & 0 & 2 & 0 & 1 \\
\text{edges} & 0 & 1 & 3 & 5 & 2 & 0 & 1 & 2
\end{array}
\]

\[
\begin{array}{c}
0 & 1 & 3 & 5 \\
\text{r_nodes} & 0 & 1 & 2 & 2 & 0 & 1 \\
\text{r_edges}
\end{array}
\]
Performance breakdown

Runtime [s]

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

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

• Public release - github.com/libshoal
DMA results

![Graph showing DMA results with two lines: one for 20 x 2 hyperthreads and another for 20 cores. The x-axis represents the percentage of fraction DMA transfers, and the y-axis represents the copy time in milliseconds. The graph shows a decrease in copy time as the percentage of DMA transfers increases for both hyperthreads and cores.]
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
Carrefour: reactive tuning

![Diagram of Carrefour architecture]

![Bar chart showing runtime for different core configurations (8, 16, 32, 64 cores) for single-node, distributed, and Carrefour executions.]

- Blue bars represent single-node execution.
- Red bars represent distributed execution.
- Green bars represent Carrefour execution.

The chart shows the runtime in seconds for each configuration.
Shoal

- Knowledge of access patterns
- Replication
- Distribution
- Large pages (2M)
BACKUP
Streamcluster results

scalability PARSEC streamcluster

runtime [ms]

original

Shoal

number of cores

0

2 \cdot 10^5

4 \cdot 10^5

6 \cdot 10^5
Green Marl results

![Graphs showing scalability of Green Marl for PageRank, hop distance, and triangle count across different number of cores.](image)
RAM throughput

![Diagram showing throughput over time for different optimization strategies: no optimizations, distribution, replication, and both. The x-axis represents time in seconds, and the y-axis represents throughput in GB/s. The colors and markers indicate different operations and phases such as graph loading and pre-processing.]
Machine details

- L1 D$: 16K / thread
- L2/L3: 2048 K / 6144 K
OLD PRESENTATIONS
Observation: HW complex

- Multicores: complex memory subsystem
  - non-uniform memory access (NUMA)
  - complex interconnect network
  - hardware features
    - different page sizes (4KB, 2/4MB, 2GB)
    - DMA engines
  - no single global address space
  - no cache-coherence

worse:

diversity, constant changes
Implications / Problem

→ programmers struggle

• where to allocate memory?
• how to access this memory?

if wrong → no scalability & bad performance:
  – interconnect contention
  – saturation of memory controllers
Goal

• **simplify** the life of programmers
• **hardware-aware** memory allocation & access
• automatic tuning, **without:**
  - constantly fine-tune applications
  - understand the hardware
• support for wide range of current/future HW
Our System: **Shoal**

- **array** abstraction
- **rich alloc call: access patterns** as arguments
  - manually or automatic from high-level description
- various implementations
  - e.g. replicated, distributed, partitioned
- **selection automatically** based on
  - access patterns and
  - **HW characteristics**
Overview

- low-level code (e.g. C)
  - Shoal array abstraction

- compiler

- hardware characteristics
  - access patterns

- program
  - Shoal library
Overview

Procedure pagerank(G: Graph, e,d: Double, 
max: Int; pg_rank: Node_Prop<Double>)
{
  Double diff;
  Int cnt = 0;
  Double N = G.NumNodes();
  G.pg_rank = 1 / N;
  Do {
    diff = 0.0;
    Foreach (t: G.Nodes) {
      Double val = (1-d) / N + d* 
      Sum(w: t.InNbrs){
        w.pg_rank / w.OutDegree()}

      diff += | val - t.pg_rank |;
      t.pg_rank <= val @ t;
    }
    cnt++;
  } While ((diff > e) && (cnt < max));
}
Evaluation: Scalability

Green Marl

- pagerank scales fairly well

- hop_dist

- triangle_count

runtime (x10^5) [ms] vs. cores for OpenMP
Evaluation: Scalability

Green Marl

- **pagerank**
  - Runtime (x10^5 [ms])
  - Cores: 8, 16, 32, 64
  - Up to 2x improvement

- **hop_dist**
  - Runtime (x10^5 [ms])
  - Cores: 8, 16, 32, 64

- **triangle_count**
  - Runtime (x10^5 [ms])
  - Cores: 8, 16, 32, 64

Graphs showing performance improvements for different cores with Green Marl.
Conclusion / Future work

• library to auto-tune memory allocation & access
  – based on: access patterns & hardware

• 2x improvement for Green Marl graph algorithms
  – no modifications to input program

• Next:
  – scheduling
  – synchronization
  – heterogeneous machines
  → long term: maybe multikernel programming model
Evaluation: arrays

Comparison of array implementations

normalized runtime

partitioning → slower, need proper runtime-system integration + scheduling