A Memory Management System Optimized for BDMPI’s Memory and Execution Model

Jeremy Iverson¹  George Karypis¹

¹University of Minnesota, Minneapolis, MN, USA

EuroMPI 2015

October 4, 2015
Consider this parallel application

- **Node 1**: rank 1 data
- **Node 2**: rank 2 data
- **Node 3**: rank 3 data
- **Node 4**: rank 4 data
One *simple* solution

![Diagram showing four nodes labeled Node 1 to Node 4 with DRAM ranks 1 to 4.](image-url)
A more realistic solution

1. DRAM
2. DRAM
3. DRAM
4. DRAM
5. DRAM
6. DRAM
7. DRAM
8. DRAM
9. DRAM
10. DRAM
11. DRAM
12. DRAM
But what if hardware is fixed?
Let’s look at a serial application
Let’s look at a serial application
Now recall the parallel application...
... and apply the serial solution
Remember the *more realistic* solution?
What if we could just...
Enter BDMPI

BigData MPI (BDMPI)

- Transparent layer between an MPI application and an MPI runtime

Node-level co-operative multi-tasking (execution model)

- MPI process will run until it blocks for a communication operation (collective, recv)
- Cost of loading data from disk is amortized over large segments of computation

Constrained memory over-subscription (memory model)

- Assumes the problem is decomposed s.t. each MPI process can fit its working set in memory
- Manages the scheduling of MPI processes per compute node to reduce pressure on OS swapping mechanism
Enter BDMPI

BigData MPI (BDMPI)

- Transparent layer between an MPI application and an MPI runtime

Node-level co-operative multi-tasking (execution model)

- MPI process will run until it blocks for a communication operation (collective, recv)
- Cost of loading data from disk is amortized over large segments of computation

Constrained memory over-subscription (memory model)

- Assumes the problem is decomposed s.t. each MPI process can fit its working set in memory
- Manages the scheduling of MPI processes per compute node to reduce pressure on OS swapping mechanism
Enter BDMPI

BigData MPI (BDMPI)

- Transparent layer between an MPI application and an MPI runtime

Node-level co-operative multi-tasking (execution model)

- MPI process will run until it blocks for a communication operation (collective, recv)
- Cost of loading data from disk is amortized over large segments of computation

Constrained memory over-subscription (memory model)

- Assumes the problem is decomposed s.t. each MPI process can fit its working set in memory
- Manages the scheduling of MPI processes per compute node to reduce pressure on OS swapping mechanism
Overview

1. SBMA
   - Motivation
   - Hypothesis and key question
   - SBMA framework

2. Results
   - Benchmarks
   - Experimental setup
   - Experiments

3. Conclusions
Pitfalls of OS swapping in BDMPI

Node 1

Time

DRAM

DISK
Pitfalls of OS swapping in BDMPI

Node 1

DRAM

DISK

rank 1 compute
Pitfalls of OS swapping in BDMPI

Node 1

- DRAM
- DISK

rank 1 compute

rank 1 comm
Pitfalls of OS swapping in BDMPI

Node 1

- rank 1 compute
- rank 1 comm
- rank 2 compute
Let’s back up...
Let’s back up...
... and reduce disk contention
Important perspective

Hypothesis

- Exploiting the BDMP! memory and execution models will lead to reduced disk contention compared with deferring to the OS VMM

Key question

- How aggressively should a process’ virtual address space be exchanged between physical memory and disk to maintain to prevent memory over-subscription?
Hypothesis

- Exploiting the BDMPI memory and execution models will lead to reduced disk contention compared with deferring to the OS VMM

Key question

- How aggressively should a process’ virtual address space be exchanged between physical memory and disk to maintain to prevent memory over-subscription?
SBMA Overview

What it is...

- Storage-Backed Memory Allocation (SBMA)

- Built as part of the BDMPI library

- User space virtual memory manager

How it works...

- Uses C interposition to fulfill applications’ memory allocation requests

- Relies on memory protection and signal handling to track status of allocated pages
An illustrative example

```c
int * arr;
arr = malloc(n);
...
for (i=0; i<n; ++i)
    if (!arr[i])
        arr[i] = 1;
...
free(arr);
```
Memory access patterns

[Diagram showing memory allocations and communication patterns]
Benchmarks

**Synthetic**
- Sequence of reads and writes
- Used to quantify the overhead introduced by the SBMA library

**PageRank**
- Memory footprint fixed
- Multiplying a sparse matrix by a vector

**ParMetis**
- Memory footprint changes throughout execution
- Recursively contracting a graph

**SPLATT**
- Memory footprint fixed, but has different phases requiring different amounts of memory
- Multiplying a sparse tensor and dense matrices
Experimental setup

**System**
- Four machine cluster with an aggregate 16GB DRAM and 1.2TB swap

**Datasets**
- Synthetic - dynamically generated random data (4GB in memory)
- PageRank - 6.6B edges, ordered randomly (35GB in memory)
- ParMetis - 760M edges (13GB in memory)
- SPLATT - 2.9M × 2.1M × 25.5M with 143.6M non-zeros (26GB in memory)
## Synthetic benchmark

<table>
<thead>
<tr>
<th></th>
<th>Read ((x == y))</th>
<th>Write ((x = y))</th>
<th>Read/Write ((x += y))</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>OS SBMA</td>
<td>OS SBMA</td>
<td>OS SBMA</td>
</tr>
<tr>
<td>AI</td>
<td>1195 1194</td>
<td>514 373</td>
<td>472 352</td>
</tr>
<tr>
<td>LI</td>
<td>1195 927</td>
<td>514 325</td>
<td>472 310</td>
</tr>
<tr>
<td>AR</td>
<td>28 28</td>
<td>514 373</td>
<td>28 28</td>
</tr>
<tr>
<td>LR</td>
<td>30 30</td>
<td>514 325</td>
<td>30 30</td>
</tr>
</tbody>
</table>

**Throughput (system pages/sec)**

- **A** Aggressive
- **L** Lazy
- **I** In-memory
- **R** On disk
## Synthetic benchmark

<table>
<thead>
<tr>
<th></th>
<th>Read ((x == y))</th>
<th>Write ((x = y))</th>
<th>Read/Write ((x += y))</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>OS SBMA</td>
<td>OS SBMA</td>
<td>OS SBMA</td>
</tr>
<tr>
<td>AI</td>
<td>1195 1194</td>
<td>514 373</td>
<td>472 352</td>
</tr>
<tr>
<td>LI</td>
<td>1195 927</td>
<td>514 325</td>
<td>472 310</td>
</tr>
<tr>
<td>AR</td>
<td>28 28</td>
<td>514 373</td>
<td>28 28</td>
</tr>
<tr>
<td>LR</td>
<td>30 30</td>
<td>514 325</td>
<td>30 30</td>
</tr>
</tbody>
</table>

Throughput (system pages/sec)

- **A** Aggressive
- **L** Lazy
- **I** In-memory
- **R** On disk
## Synthetic benchmark

<table>
<thead>
<tr>
<th></th>
<th>Read (x == y)</th>
<th>Write (x = y)</th>
<th>Read/Write (x += y)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>OS (SBMA)</td>
<td>OS (SBMA)</td>
<td>OS (SBMA)</td>
</tr>
<tr>
<td>AI</td>
<td>1195</td>
<td>514</td>
<td>472</td>
</tr>
<tr>
<td>LI</td>
<td>1195</td>
<td>514</td>
<td>472</td>
</tr>
<tr>
<td>AR</td>
<td>28</td>
<td>514</td>
<td>28</td>
</tr>
<tr>
<td>LR</td>
<td>30</td>
<td>514</td>
<td>30</td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>

**Throughput (system pages/sec)**

- **A** Aggressive
- **L** Lazy
- **I** In-memory
- **R** On disk
## Real world benchmarks

<table>
<thead>
<tr>
<th>Benchmark Application</th>
<th>Serial</th>
<th>SPLATT</th>
<th>SBMA-ARAW</th>
<th>SBMA-ARLW</th>
<th>SBMA-LRLW</th>
</tr>
</thead>
<tbody>
<tr>
<td>PageRank</td>
<td>19.86</td>
<td>14.84</td>
<td>13.91</td>
<td>12.13</td>
<td>10.28</td>
</tr>
<tr>
<td>ParMetis</td>
<td>40.85</td>
<td>35.34</td>
<td>22.07</td>
<td>11.92</td>
<td>8.41</td>
</tr>
<tr>
<td>SPLATT</td>
<td>38.32</td>
<td>35.34</td>
<td>22.07</td>
<td>11.92</td>
<td>7.17</td>
</tr>
</tbody>
</table>

**Runtime (m)**

![Bar chart showing runtime comparison for PageRank, ParMetis, and SPLATT benchmark applications using different methods.]
Conclusions

What we’ve learned

- Possible to implement a user space virtual memory manager with less a $2 \times$ slowdown in memory throughput
- Exploiting BDMPI’s execution and memory models improves performance over OS VMM with speedups from $2 \times$ to $12 \times$
Conclusions

What we’ve learned

- Possible to implement a user space virtual memory manager with less a $2\times$ slowdown in memory throughput
- Exploiting BDMPI’s execution and memory models improves performance over OS VMM with speedups from $2\times$ to $12\times$

Moving forward

- Add support for MPI+X
- Allow more than one process to run simultaneously on each compute node so long as memory constraint is not violated
Questions?

jiverson@cs.umn.edu

http://glaros.dtc.umn.edu/gkhome/bdmpi/download