Optimizing Collective Communication on Multicores

Rajesh Nishtala\textsuperscript{1} Katherine Yelick\textsuperscript{1}

\textsuperscript{1}University of California, Berkeley

(2009)
Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors

John M. Mellor-Crummey, Michael L.Scott
(1991)
PGAS Languages

- Focus on Partitioned Global Address Space languages
Partitioned Addressspace

one address space

$T_1 \quad T_2 \quad \ldots \quad T_n$
One Sided Communication

$T_1$, $T_2$, ..., $T_n$

read

write
PGAS Languages

- UPC, Unified Parallel C
- CAF, Co-array Fortran
- Titanium, a Java dialect
The gap between processors and memory systems is still enormous
Today: processors don’t get faster, but we see more and more processors on a single chip
<table>
<thead>
<tr>
<th>Processor</th>
<th>GHz</th>
<th>Cores (Threads)</th>
<th>Sockets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Clovertown</td>
<td>2.66</td>
<td>8 (8)</td>
<td>2</td>
</tr>
<tr>
<td>AMD Barcelona</td>
<td>2.3</td>
<td>32 (32)</td>
<td>8</td>
</tr>
<tr>
<td>Sun Niagara 2</td>
<td>1.4</td>
<td>32 (256)</td>
<td>4</td>
</tr>
</tbody>
</table>

**Table**: Experimental Platforms
Sun Niagara 2

The number of processors on a chip grows at an exponential pace
Intel Single-Chip Cloud Computer (48 Cores)

http://techresearch.intel.com/ProjectDetails.aspx?Id=1
Communication in its most general form is the movement of data within cores, between cores or within memory systems.
Collective Communication

- Communication-intensive problems often involve global communication
Broadcast
Gather
These operations are thought of as collective communication operations.
Example: Sum of Vector Elements

1  2  3  4  5  6  7  8  9  10
Create workers

$W_1 \quad W_2 \quad W_3 \quad W_4 \quad W_5$
Example: Sum of Vector Elements

- Every worker sums up its part of the vector

1 → 3
2 → 7
3 → 11
4 → 15
5 → 19
6
7
8
9
10

\[ \frac{3 + 7 + 11 + 15 + 19}{57} \]
Example: Sum of Vector Elements

- The main thread gathers the partial results and sums them up

```
1 2 3 4 5 6 7 8 9 10
3 7 11 15 19
55
```
Example: Sum of Vector Elements

Pseudocode (main thread):

```java
double [] vector = read_vector();
Thread [] workers = spwan_workers();

start_workers(workers);
double result = calculate_result(workers);
```
Example: Sum of Vector Elements

Pseudocode (main thread):

```plaintext
double [] vector = read_vector();
Thread [] workers = spawn_workers();

start_workers(workers);
wait_until_everything_finished(workers);
double result = calculate_result(workers);
```
Barrier

- Synchronization method for a group of threads
- A thread can only continue its execution after every thread has called the barrier
Collective Communication Operation

“... group of threads works together to perform a global communication operation ...”
Reduce

- Divide a problem into smaller subproblems
- Every thread contributes its part to the solution
- Example: Calculate the smallest entry of a vector
Flat vs. Tree

- For communication among threads, different topologies can be used
Flat Topology

- Example: we have a reduce operation
- in the end the main thread $W_{main}$ has to wait for every worker thread $W_1, \ldots, W_7$
Tree Topology

- Example: we have a reduce operation
- in the end the main thread $W_{main}$ has to wait for every worker thread $W_1, \ldots, W_7$

$W_{main} \ W_1 \ W_2 \ W_3 \ W_4 \ W_5 \ W_6 \ W_7$
Analysis

Figure: Barrier Performance

Nishtala, R., Yelick, K. Optimizing Collective Communication on Multicores
Barrier Implementation

```c
#define N 4

pthread_t threads[N];
volatile int ready[N];
volatile int go[N];

void barrier(int id) {
    if (id == 0) {
        // wait for each thread
        for (int i = 1; i < N; i++)
            while (ready[i] == 0);

        // reset the ready flags
        for (int i = 0; i < N; i++)
            ready[i] = 0;

        // signal each thread
        for (int i = 0; i < N; i++)
            go[i] = 1;
    }
    else {
        ready[id] = 1;
        // wait until thread is signalled
        while (go[id] == 0);
        go[id] = 0;
    }
}
```
Experiment: Barrier Implementation
Strict synchronization: Data movement can only start after all threads have entered the collective and must be completed before the first thread exits the collective.
Strict Synchronization
Loosening Synchronization Requirements

- **Loose synchronization**: Data movement can begin as soon as any thread has entered the collective and continue until the last thread leaves the collective.
Loose Synchronization
Figure 5: Optimal Algorithm Selection on Niagara2

(32 cores, 256 threads)
Figure 6: Optimal Algorithm Selection on Clovertown

(8 cores, 8 threads)
Figure 4: Optimal Algorithm Selection on Barcelona

(32 cores, 32 threads)
Summary

- Best strategy depends on the hardware and on the problem
- Using a library that can automatically adapt to a given situation can bring a great performance improvement, since hand tuning takes far too long
Words on the Paper

- Very high level
- Description of the problem without concrete solution
- No implementation
- Plots aren’t always clear and precise