Empirical (so far) Understanding of Communication Optimizations for GAS Languages

Costin Iancu
LBNL
$10,000 Questions

- Can GAS languages do better than message passing?

- Claim: maybe, if programs are optimized simultaneously both in terms of serial and parallel performance.

- If not, is there any advantage?

- Claim - flexibility in choosing the best implementation strategy.
Motivation

- Parallel programming - cycle tune parallel, tune serial
- Serial and parallel optimizations - disjoint spaces
- Previous experience with GAS languages showed performance comparable with hand tuned MPI codes.
• Traditionally parallel programming done in terms of two-sided communication.

• Previous work on parallelizing compilers and comm. optimizations reasoned mostly in the terms of two sided communication.

• Focus on domain decomposition, lowering synchronization costs or finding the best schedule.

• GAS languages are based on one-sided communication. Domain decomposition done by programmer, optimizations done by compiler.
Optimization Spaces

• Serial optimizations -> interested mostly in loop optimizations:
  - Unrolling
  - Software pipelining
  - Tiling

• Parallel optimizations:
  - Communication scheduling (comm-comm ovlp, comm/comp ovlp)
  - Message vectorization
  - Message coalescing and aggregation
  - Inspector-executor
Parameters

• Architectural:
  - Processor -> Cache
  - Network -> LogPC, contention (LogPC)

• Software interface: blocking/non blocking primitives, explicit/implicit synchronization, scatter/gather….

• Application characteristics: memory and network footprint
Modern Systems

- Large memory-processor distance: 2-10/20 cycles cache miss latency

- High bandwidth networks: 200MB/s-500M/s => cheaper to bring a byte over the network than a cache miss

- Natural question: by combining serial and parallel optimization can one trade cache misses with network bandwidth and/or overhead?
Goals

Given an UPC program and the optimization space parameters, choose the combination of parameters that minimizes the total running time.
(What am I really talking about)

LOOPS

for (i=0; i < N; i++)
    dest[g(i)] = f(src[h(i)]);

• g(i), h(i) - indirect access -> unlikely to vectorize
  Either fine grained communication or inspector-executor

• g(I) - direct access - can be vectorized
  get_bulk(local_src, src);
  for(…)
      local_dest[g[i]] = local_src[g[i]];
  put_bulk(dest, local_dest)
Fine Grained Loops

- Fine grained loops - unrolling, software pipelining and communication scheduling

```c
for(...) {
    init 1; sync 1; compute 1; write back 1;
    init 2; sync 2; compute 2; write back 2;
    .......
}
```
Fine Grained Loops

\begin{verbatim}
for(...) {
    init 1; sync1;
    compute1;
    write1;
    init 2; sync 2;
    compute 2;
    write 2;
    ....
}
(base)
\end{verbatim}

\begin{verbatim}
for (...) {
    init 1;
    init 2;
    init 3;
    ....
    sync_all;
    compute all;
    }
\end{verbatim}

\begin{verbatim}
for (...) {
    init 1;
    init2;
    sync 1;
    compute 1;
    ....
    }
\end{verbatim}

- Problem to solve - find the best schedule of operations and unrolling depth such as to minimize the total running time
Coarse Grained Loops

- Coarse grained loops - unrolling, software pipelining and communication scheduling + “blocking/tiling”

<table>
<thead>
<tr>
<th>Base</th>
<th>Register</th>
<th>Overlapping</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get_bulk(local_src, src);</code></td>
<td><code>for(...) {</code></td>
<td><code>get B1;</code></td>
</tr>
<tr>
<td><code>for(...) {</code></td>
<td><code>get B1;</code></td>
<td><code>for (...) {</code></td>
</tr>
<tr>
<td><code>local_dest[g[i]] = local_src[g[i]];</code></td>
<td><code>get B2;</code></td>
<td><code>sync Bi;</code></td>
</tr>
<tr>
<td><code>}</code></td>
<td><code>...</code></td>
<td><code>get Bj+1;</code></td>
</tr>
<tr>
<td><code>put_bulk(dest, local_dest);</code></td>
<td><code>sync B1;</code></td>
<td><code>compute Bi;</code></td>
</tr>
<tr>
<td>(base)</td>
<td><code>compute B1;</code></td>
<td><code>sync Bi+1;</code></td>
</tr>
<tr>
<td>(reg)</td>
<td><code>compute B2;</code></td>
<td><code>compute Bi+1;</code></td>
</tr>
<tr>
<td>(ovlp)</td>
<td><code>.....</code></td>
<td><code>.....</code></td>
</tr>
</tbody>
</table>
Coarse Grained Loops

- Coarse grained loops could be “tiled”. Add the tile size as a parameter to the optimization problem.

- Problem to solve - find the best schedule of operations, unrolling depth and “tile” size such as to minimize the total running time

- Questions:
  - Is the tile constant?
  - Is the tile size a function of cache size and/or network parameters?
How to Evaluate?

- Synthetic benchmarks - fine grained messages and large messages

- Distribution of the access stream varies: uniform, clustered and hotspot => UPC datatypes

- Variable computation per message size - \( kN, N, K^2N, N^2 \).

- Variable memory access pattern - strided and linear.
Evaluation Methodology

- Alpha/Quadrics cluster
- X86/Myrinet cluster
- All programs compiled with highest optimization level and aggressive inlining.
- 10 runs, report average
Fine Grained Communication
Fine Grained Communication

```c
for(...) {
    init 1; sync 1;
    compute1;
    write1;
    init 2; sync 2; ....
    compute 2;
    write 2;
    ....
}
```

```c
for (...) {
    init 1;
    init 2;
    sync 1;
    compute 1;
    sync_all;
    compute all;
    ....
}
```

- Interested in the benefits of communication communication overlap
X86/Myrinet (os > g)

- comm/comm overlap is beneficial

- loop unrolling helps, best factor $32 < U < 64$
Write Pipelining (clustered distribution)

Read pipelining (clustered)

X86/Myrinet (os > g)
Myrinet: communication/communication overlap works, use non-blocking primitives for fine grained messages. There’s a limit on the number of outstanding messages (32 < L <64).
Read Pipelining (uniform distribution)

Write Pipelining (uniform distribution)

Alpha/Quadrics (g > os)
Read Pipelining (clustered sequence)

Write Pipelining (clustered sequence)

Alpha/Quadrics
On Quadrics, for fine grained messages where there the amount of computation available for overlap is small - use blocking primitives.
Coarse Grained Communication
Benchmark

- Fixed amount of computation
- Vary the message sizes.
- Vary the loop unrolling depth.

```c
get_bulk(local_src, src);
for(...) {
    local_dest[g[i]] = local_src[g[i]];
}
put_bulk(dest, local_dest);
```

(base)

```c
for(...) {
    get B1;
    get B2;
    ...
    sync B1;
    compute B1;
    sync B2;
    compute B2;
    ......
}
```

(reg)

```c
get B1;
...
for (...) {
    sync Bi;
    get Bj+1;
    compute Bi;
    sync Bi+1;
    compute Bi+1;
    ......
}
```

(ovlp)
Alpha/Quadrics
Software pipelining with staggered gets is slower.
Alpha/Quadrics

- Both optimizations help.

- Again knee around tile x unroll = cache_size

- The optimal value for the blocking case - is it a function of contention or some other factor (packet size, TLB size)
Hotspot - linear computation - 1.0

Alpha/Quadrics

Staggered better than back-to-back - result of contention.
Conclusion

• Unified optimization model - serial+parallel likely to improve performance over separate optimization stages

• Fine grained messages:
  os > g -> comm/comm overlap helps
  g > os -> comm/comm overlap might not be worth

• Coarse grained messages:
  - Blocking improves the total running time by offering better opportunities for comm/comp overlap and reducing pressure
  - “Software pipelining” + loop unrolling usually better than unrolling alone
Future Work

• Worth further investigation - trade bandwidth for cache performance (region based allocators, inspector executor, scatter/gather)

• Message aggregation/coalescing ?
Other Questions

- Fact: Cache miss time same order of magnitude as G.
  Question - can somehow trade cache misses for bandwidth? (scatter/gather, inspector/executor)

- Fact: program analysis often over conservative.
  Question: given some computation communication overlap how much bandwidth can I waste without noticing in the total running time. (prefetch and region based allocators)
Effect of contention (cluster length)

Message Size (2x Bytes) vs. Time (sec)