Parallelizing Prefix Sums

William Killian
killian@udel.edu
University of Delaware

2014 May 21
Outline

The Algorithm

Parallelization
  Method
  Adapting to Parallel Execution Models

Results
  Laptop
  Chimera

CIVL Verification
What are Partial Sums?

Element Computation

Provided a one-dimensional array, \( x \), the resulting array, \( y \), is given as:

\[
\begin{align*}
y[0] &\leftarrow x[0] \\
y[1] &\leftarrow x[0] + x[1] \\
\vdots \\
y[n] &\leftarrow x[0] + x[1] + \cdots + x[n]
\end{align*}
\]

Which can be simplified to:

\[
y[i] \leftarrow \sum_{j=0}^{i} x[j]
\]
A Sequential Implementation

Require: \( n \in \mathbb{N} \)

Require: \( x \) is an array of size \( n \)

Require: \( y \) is an array of size \( n \)

Ensure: \( y \) contains the \texttt{PREFIX-SUM} elements of \( x \)

1: \( s = 0 \)

2: \textbf{for} \( 0 \leq i < n \) \textbf{do}

3: \( s \leftarrow s + x[i] \)

4: \( y[i] \leftarrow s \)

5: \textbf{end for}
Where can we parallelize?

Problem:
There is a dependence on the previous iteration’s value

Solution:
Divide $\rightarrow$ Conquer $\rightarrow$ Merge
(Still) a Sequential Implementation

Divide → Conquer → Merge

1: \( s = 0 \)
2: \( \text{for } 0 \leq i < \frac{n}{2} \text{ do} \)
3: \( s \leftarrow s + x[i] \)
4: \( y[i] \leftarrow s \)
5: \( \text{end for} \)

▷ Note the value of \( s \)

6:
7: \( \text{for } \frac{n}{2} \leq i < n \text{ do} \)
8: \( s \leftarrow s + x[i] \)
9: \( y[i] \leftarrow s \)
10: \( \text{end for} \)
(Still) a **BIGGER** Sequential Implementation

1. \( s = 0 \)
2. **for** \( 0 \leq i < \frac{n}{4} \) **do**
3. \( s \leftarrow s + x[i] \)
4. \( y[i] \leftarrow s \)
5. **end for**
6. **for** \( \frac{n}{4} \leq i < \frac{n}{2} \) **do**  
   \[ \triangleright \text{Value of } s \]  
7. \( s \leftarrow s + x[i] \)
8. \( y[i] \leftarrow s \)
9. **end for**
10. **for** \( \frac{n}{2} \leq i < \frac{3}{4}n \) **do**  
    \[ \triangleright \text{Value of } s \]  
11. \( s \leftarrow s + x[i] \)
12. \( y[i] \leftarrow s \)
13. **end for**
14. **for** \( \frac{3}{4}n \leq i < n \) **do**  
    \[ \triangleright \text{Value of } s \]  
15. \( s \leftarrow s + x[i] \)
16. \( y[i] \leftarrow s \)
17. **end for**
Parallel Implementation (1/2)

Require: All Prior Preconditions
Require: $P$ is the number of processes
Require: $S$ is a shared array of size $P$

1: function $\text{PREFIX-SUM}(x, y, s_0, s_n)$
2: $s = 0$
3: for $s_0 \leq i < s_n$ do
4: \hspace{1em} $y[i] \leftarrow s \leftarrow s + x[i]$
5: \hspace{1em} end for
6: return $s$
7: end function

8: function $\text{ADD-CONSTANT}(x, y, c, s_0, s_n)$
9: for $s_0 \leq i < s_n$ do
10: \hspace{1em} $y[i] \leftarrow c + x[i]$
11: \hspace{1em} end for
12: end function
Parallel Implementation (2/2)

13: function Shift-Right(x, n)
14:     for n > i > 0 do
15:         x[i] ← x[i − 1]
16:     end for
17:     x[0] ← 0
18: end function

19: S[0 . . . P − 1] = 0
20: for 0 ≤ p < P do
21:     spawn S[p] ← Prefix-Sum(x, y, \( \frac{p \times n}{P} \), \( \frac{(p+1) \times n}{P} \))
22: end for
23: Prefix-Sum(S, S, 0, P)
24: Shift-Right(S, P)
25: for 0 ≤ p < P do
26:     spawn Add-Constant(y, y, S[p], \( \frac{p \times n}{P} \), \( \frac{(p+1) \times n}{P} \))
27: end for
void omp_scan (double* in, double* out, size_t n, int tCount) {
    omp_set_num_threads (tCount);
    double* partial = (double*) malloc (tCount * sizeof (double));
    #pragma omp parallel default (shared)
    {
        int tID = omp_get_thread_num ();
        size_t chunk, start, stop;
        getPartition (n, tID, tCount, &start, &stop, &chunk);
        partial [tID] = scan (in, out, start, stop);
    #pragma omp barrier
    #pragma omp single
        recomputePartials (partial, tCount);
    #pragma omp barrier
    #pragma omp barrier
        addValToArray (out, partial [tID], start, stop);
    }
    free (partial);
}

double* in, *out;
create (&in, &out, N);
initialize (in, out, N);
omp_scan (in, out, N, tCount);
check (out, N);
cleanup (&in, &out);
Adapting to MPI

```c
void mpi_scan (REAL* in, REAL* out, REAL* sum, size_t n, int rank, int size) {
    REAL *local_in, *local_out, local_sum;
    int *firsts, *counts, local_n;
    getDistribution (&firsts, &counts, n, size);
    local_n = counts [rank];
    create (&local_in, &local_out, local_n);
    MPI_Scatterv (in, counts, firsts, MY_MPI_REAL, local_in, local_n, MY_MPI_REAL, 0, MPI_COMM_WORLD);
    local_sum = scan (local_in, local_out, 0, local_n);
    MPI_Exscan (MPI_IN_PLACE, &local_sum, 1, MY_MPI_REAL, MPI_SUM, MPI_COMM_WORLD);
    local_sum = (rank == 0) ? 0 : local_sum);
    addValToArray (local_out, local_sum, 0, local_n);
    MPI_Barrier (MPI_COMM_WORLD);
    MPI_Gatherv (local_out, local_n, MY_MPI_REAL, out, counts, firsts, MY_MPI_REAL, 0, MPI_COMM_WORLD);
    cleanup (&local_in, &local_out);
}

MPI_Init (&argc, &argv);
double *in, *out, *sum;
in = out = sum = NULL;
if (rank == 0) {
    create (&in, &out, N);
    initialize (in, out, N);
    sum = (double*) malloc (sizeof (double) * size);
}
mpi_scan (in, out, sum, N, rank, size);
if (rank == 0) {
    check (out, N);
    cleanup (&in, &out);
    free (sum);
}
MPI_Finalize ();
```
void hybrid_scan (REAL* in, REAL* out, REAL* sum, size_t n, int rank, int size, int tCount) {
    REAL *local_in, *local_out, *partial, local_sum;
    int *firsts, *counts, local_n;
    struct timeval start, stop, total;
    partial = (REAL*) malloc (tCount * sizeof (REAL));
    getDistribution (&firsts, &counts, n, size);
    local_n = counts [rank];
    create (&local_in, &local_out, local_n);
    MPI_Scatterv (in, counts, firsts, MY_MPI_REAL, local_in, local_n, MY_MPI_REAL, 0, MPI_COMM_WORLD);
    #pragma omp parallel num_threads (tCount)
    {
        size_t tID = omp_get_thread_num ();
        size_t chunk, start, stop;
        getPartition (local_n, tID, tCount, &start, &stop, &chunk);
        partial [tID] = scan (local_in, local_out, start, stop);
        #pragma omp barrier
        #pragma omp single
        recomputePartials (partial, tCount);
        addValToArray (local_out, partial [tID], start, stop);
        #pragma omp barrier
        #pragma omp single
        {
            local_sum = local_out [local_n - 1];
            MPI_Exscan (MPI_IN_PLACE, &local_sum, 1, MY_MPI_REAL, MPI_SUM, MPI_COMM_WORLD);
            local_sum = (rank == 0) ? 0 : local_sum;
        }
        addValToArray (local_out, local_sum, start, stop);
    }
    MPI_Barrier (MPI_COMM_WORLD);
    MPI_Gatherv (local_out, local_n, MY_MPI_REAL, out, counts, firsts, MY_MPI_REAL, 0, MPI_COMM_WORLD);
    free (partial);
    cleanup (&local_in, &local_out);
}

// MPI Initialization the same except for:
// MPI_Init_thread (&argc, &argv, MPI_THREAD_FUNNELED, &tTotal);
// hybrid_scan (in, out, sum, N, rank, size, tCount);
Execution Overview

Versions

- Sequential
- OpenMP (various sizes)
- MPI (various sizes)
- OpenMP + MPI (various configurations)

Dataset

Dataset initialization was set to all '1's

Dataset size depended on target architecture, but was always passed as a parameter
Laptop

Configuration

- Intel Core i7-4960HQ @ 2.6GHz
- 8MB L3 Cache + 128MB eDRAM
- 16GB DDR3 SDRAM
- Dataset size: 200,000,000

Default execution time: 1.173470s

Figure 1: OpenMP and MPI Execution

<table>
<thead>
<tr>
<th>Cores</th>
<th>Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.261138</td>
<td>1.51897</td>
</tr>
<tr>
<td>2</td>
<td>1.194443</td>
<td>1.98244</td>
</tr>
<tr>
<td>4</td>
<td>0.827265</td>
<td>1.41849</td>
</tr>
<tr>
<td>8</td>
<td>0.801261</td>
<td>1.46452</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cores</th>
<th>Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5.742962</td>
<td>0.20433</td>
</tr>
<tr>
<td>2</td>
<td>3.071816</td>
<td>0.38201</td>
</tr>
<tr>
<td>4</td>
<td>1.111330</td>
<td>1.05591</td>
</tr>
<tr>
<td>8</td>
<td>0.833837</td>
<td>1.40731</td>
</tr>
</tbody>
</table>
Figure 3: Hybrid Execution Time (seconds)

<table>
<thead>
<tr>
<th>MPI</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6.3362</td>
<td>2.4157</td>
<td>1.6818</td>
<td>3.7314</td>
<td>1.4097</td>
<td>1.3169</td>
<td>1.6628</td>
<td>3.1944</td>
</tr>
<tr>
<td>2</td>
<td>2.5486</td>
<td>1.4186</td>
<td>1.9464</td>
<td>1.1498</td>
<td>1.1851</td>
<td>1.2890</td>
<td>2.5754</td>
<td>2.9715</td>
</tr>
<tr>
<td>3</td>
<td>1.8168</td>
<td>2.7403</td>
<td>1.1098</td>
<td>1.1346</td>
<td>1.1696</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1.5722</td>
<td>3.0802</td>
<td>1.1857</td>
<td>1.5968</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1.4207</td>
<td>2.8765</td>
<td>1.1591</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1.2737</td>
<td>3.0981</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1.2900</td>
<td>2.1132</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1.1887</td>
<td>1.1204</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Chimera

Configuration

- 4x AMD Opteron 6164HE 12-core @ 1.7GHz
- 2x 6MB L3 Cache
- 64GB ECC DDR2 SDRAM
- Dataset size: 2,000,000,000

Figure 4 : OpenMP

<table>
<thead>
<tr>
<th>Cores</th>
<th>Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>21.2341</td>
<td>1.0000</td>
</tr>
<tr>
<td>2</td>
<td>12.4890</td>
<td>1.7002</td>
</tr>
<tr>
<td>4</td>
<td>6.6700</td>
<td>3.1835</td>
</tr>
<tr>
<td>8</td>
<td>3.5037</td>
<td>6.0605</td>
</tr>
<tr>
<td>12</td>
<td>4.1883</td>
<td>5.0699</td>
</tr>
<tr>
<td>16</td>
<td>3.5600</td>
<td>5.9647</td>
</tr>
<tr>
<td>24</td>
<td>2.5418</td>
<td>8.3541</td>
</tr>
<tr>
<td>32</td>
<td>2.2778</td>
<td>9.3221</td>
</tr>
<tr>
<td>40</td>
<td>2.0729</td>
<td>10.2434</td>
</tr>
<tr>
<td>48</td>
<td>2.3834</td>
<td>8.9091</td>
</tr>
</tbody>
</table>
Hybrid Implementation performed similarly

launch configuration was set to `-n N -c T`

with $N$ being number of MPI procs

and $T$ being number of OpenMP threads

Not enough compute to offset setup/teardown

MPI overhead still slightly above OpenMP (3x)

**Figure 5: MPI Execution**

<table>
<thead>
<tr>
<th>Cores</th>
<th>Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>34.126681</td>
<td>1.000000</td>
</tr>
<tr>
<td>8</td>
<td>20.623163</td>
<td>1.65477</td>
</tr>
<tr>
<td>16</td>
<td>13.840286</td>
<td>2.46575</td>
</tr>
<tr>
<td>24</td>
<td>10.871229</td>
<td>3.13917</td>
</tr>
<tr>
<td>32</td>
<td>8.848653</td>
<td>3.85671</td>
</tr>
<tr>
<td>96</td>
<td>3.816591</td>
<td>8.94167</td>
</tr>
<tr>
<td>192</td>
<td>1.282415</td>
<td>26.61126</td>
</tr>
<tr>
<td>384</td>
<td>0.692448</td>
<td>49.28411</td>
</tr>
<tr>
<td>768</td>
<td>0.348209</td>
<td>98.00632</td>
</tr>
<tr>
<td>1536</td>
<td>0.145927</td>
<td>233.86132</td>
</tr>
</tbody>
</table>
CIVL Verification

- Sequential - Verified
- OpenMP - Verified (using $\text{barrier}$)
- MPI - added Exscan, Gatherv, and Scatterv
- Hybrid - Verified using above two
- Problem: ran into new bugs with CIVL for Hybrid and MPI