Parallelizing Prefix Sums

William Killian
killian@udel.edu

University of Delaware

2014 May 21

Outline

What are Partial Sums?

Element Computation

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

$$y[0] \leftarrow x[0]$$
$$y[1] \leftarrow x[0] + x[1]$$
$$y[2] \leftarrow x[0] + x[1] + x[2]$$
$$\vdots$$
$$y[n] \leftarrow x[0] + x[1] + \cdots + x[n]$$

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 Prefix-Sum elements of x

$s = 0$
for $0 \leq i < n$ **do**
    $s \leftarrow s + x[i]$
    $y[i] \leftarrow s$
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 $\rightarrow$ **Conquer** $\rightarrow$ Merge

$s = 0$
for $0 \leq i < \frac{n}{2}$ **do**
    $s \leftarrow s + x[i]$
    $y[i] \leftarrow s$
end for
▷ Note the value of s
for $\frac{n}{2} \leq i < n$ **do**
    $s \leftarrow s + x[i]$
    $y[i] \leftarrow s$
end for

(Still) a **BIGGER** Sequential Implementation

$s = 0$

for $0 \le i < \frac{n}{4}$ **do**

  $s \leftarrow s + x[i]$

  $y[i] \leftarrow s$

end for

for $\frac{n}{4} \le i < \frac{n}{2}$ **do** ▷ Value of s ?

  $s \leftarrow s + x[i]$

  $y[i] \leftarrow s$

end for

for $\frac{n}{2} \le i < \frac{3}{4}n$ **do** ▷ Value of s ?

  $s \leftarrow s + x[i]$

  $y[i] \leftarrow s$

end for

for $\frac{3}{4}n \le i < n$ **do** ▷ Value of s ?

  $s \leftarrow s + x[i]$

  $y[i] \leftarrow s$

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

function PREFIX-SUM(x, y, s_0, s_n)
    $s = 0$
    for $s_0 \leq i < s_n$ **do**
        $y[i] \leftarrow s \leftarrow s + x[i]$
    end for
    return s
end function

function ADD-CONSTANT(x, y, c, s_0, s_n)
    for $s_0 \leq i < s_n$ **do**
        $y[i] \leftarrow c + x[i]$
    end for
end function

Parallel Implementation (2/2)

```
function Shift-Right(x, n)
    for n > i > 0 do
        x[i] ← x[i − 1]
    end for
    x[0] ← 0
end function

S[0 ... P − 1] = 0
for 0 ≤ p < P do
    spawn S[p] ← Prefix-Sum(x, y, p*n/P, (p+1)*n/P)
end for
Prefix-Sum(S, S, 0, P)
Shift-Right(S, P)
for 0 ≤ p < P do
    spawn Add-Constant(y, y, S[p], p*n/P, (p+1)*n/P)
end for
```

Adapting to OpenMP

```
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
    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

```
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 ();
```

Adapting to Hybrid (OpenMP+MPI)

```
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:
```

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

(a) OpenMP

Cores	Time	Speedup
1	2.261138	1.51897
2	1.194443	1.98244
4	0.827265	1.41849
8	0.801261	1.46452

(b) MPI

Cores	Time	Speedup
1	5.742962	0.20433
2	3.071816	0.38201
4	1.111330	1.05591
8	0.833837	1.40731

Laptop

Figure 3 : Hybrid Execution Time (seconds)

	OpenMP Threads							
MPI	1	2	3	4	5	6	7	8
1	6.3362	2.4157	1.6818	3.7314	1.4097	1.3169	1.6628	3.1944
2	2.5486	1.4186	1.9464	1.1498	1.1851	1.2890	2.5754	2.9715
3	1.8168	2.7403	1.1098	1.1346	1.1696			
4	1.5722	3.0802	1.1857	1.5968				
5	1.4207	2.8765	1.1591					
6	1.2737	3.0981						
7	1.2900	2.1132						
8	1.1887	1.1204						

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

Cores	Time	Speedup
1	21.2341	1.0000
2	12.4890	1.7002
4	6.6700	3.1835
8	3.5037	6.0605
12	4.1883	5.0699
16	3.5600	5.9647
24	2.5418	8.3541
32	2.2778	9.3221
40	2.0729	10.2434
48	2.3834	8.9091

Chimera

Notes

- 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

Insights

- Not enough compute to offset setup/teardown
- MPI overhead still slightly above OpenMP (3x)

Figure 5 : MPI Execution

Cores	Time	Speedup
1	34.126681	1.00000
8	20.623163	1.65477
16	13.840286	2.46575
24	10.871229	3.13917
32	8.848653	3.85671
96	3.816591	8.94167
192	1.282415	26.61126
384	0.692448	49.28411
768	0.348209	98.00632
1536	0.145927	233.86132

CIVL Verification

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