Implementation of Parallel Bitonic Sort using MPI and Intel TBB

Presented for CSE702 Fall 2021
Instructor: Dr. Russ Miller
Presenter: Zainul Abideen Sayed
Why care about parallel sorting algorithms?
Why parallel sorting?

• **Sorting** is one of the most common operations in computation.
• The advancement in parallel hardware.
• Increasing nodes in a cluster and cores in a processor.
• Efficient utilization of resources.
• Therefore, good parallel sorting algorithms are needed.
Popular parallel sorting algorithms

- **Bitonic sort**: Bitonic sorting algorithm is based on bitonic sorting network. The key operation is based on the sorting network which converts a given sequence into a bitonic sequence and finally bitonic merge can produce a monotonically increasing or decreasing sequency.
- **Sample sort**
- **Merge sort**
- **Quick sort**
- **Radix sort**
Bitonic Sort Principle

- Bitonic sequence \(<1, 2, 4, 7, 6, 0> <8, 9, 2, 1, 0, 4> <0, 4, 8, 9, 2, 1> <3, 4, 7, 8, 6, 5, 2, 1>\)
- Let \(s = <a_0, a_1, ..., a_{n-1}>\)
- \(s_1 = \{ \min( a_0, a_{n/2} ), \min( a_1, a_{n/2+1} ), ..., \min( a_{n/2-1}, a_{n-1} ) \} \)
- \(s_2 = \{ \max( a_0, a_{n/2} ), \max( a_1, a_{n/2+1} ), ..., \max( a_{n/2-1}, a_{n-1} ) \} \)
- In sequence \(s_1\), there is an element \(b_i = \min\{ a_i, a_{n/2+i} \}\) such that all the elements before \(b_i\) are from the increasing part of the original sequence and all the elements after \(b_i\) are from the decreasing part.
- Opposite case for \(b_i \` = \max\{ a_i, a_{n/2+i} \}\)

*Introduction to Parallel Computing 2nd Edition, Ananth Grama*
Example

Log$_2$N

3 4 7 8 | 6 5 2 1
3 4 2 1 | 6 5 7 8
2 1 | 3 4 | 6 5 | 7 8
1 | 2 | 3 | 4 | 5 | 6 | 7 | 8

All elements in second sequence is greater than the first

Log$_2$N
Example: Sorting Network (Sort + Merge)
Example: 16 lines

*Introduction to Parallel Computing 2nd Edition, Ananth Grama
Example: 16 lines

Hypercube

*Introduction to Parallel Computing 2nd Edition, Ananth Grama*
Algorithm

COMPARE SECTION

procedure BITONIC_SORT(label, d)
begin
  for i := 0 to d - 1 do
    for j := i downto 0 do
      if (i + 1)st bit of label != j th bit of label then
        comp_exchange_max(j);
      else
        comp_exchange_min(j);
    end BITONIC_SORT
end

Complexity

- \( (1 + \log n) \frac{(\log n)}{2} \)
- \( T_p = \theta(n \log^2 n) \)

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>comp_exchange_min(0);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>comp_exchange_max(1);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>comp_exchange_max(0);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>comp_exchange_max(2);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>comp_exchange_min(1);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>comp_exchange_min(0);</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
More on complexity…

• N/P Block of data per processor
• Fast sequential sort
  - Merge sort $\theta((n/p) \log(n/p))$
• Bitonic Merge
  - $\theta(\log 2 p)$

\[
T_p = \Theta\left(\frac{n}{p} \log \frac{n}{p}\right) + \Theta\left(\frac{n}{p} \log^2 p\right) + \Theta\left(\frac{n}{p} \log^2 p\right).
\]
Results for 1 Billion keys

**Key Observation:** Inter-node parallelism give better performance than Intra-node parallelism.

**Relative speed-up:**
\[ T_1 = 6.53; \quad T_{16} = 0.8 \]

\[ S_p = 8 \]
\[ E_p = 0.5 \]

- Was able to sort 100 Billion keys on 512 ranks in 44.7524 sec
- Input Data size : 383G
Speed-up and Efficiency
Results for 1 Billion keys

Observation:

• The code stops scaling after 128 Nodes.

• The execution time increases in some cases.

• However, it will be interesting to see the performance at 256 nodes.
Amdahl's Law

The Law focuses on strong scaling where input remains constant, and we increase the processors expecting the runtime to reduce in proportion to the number of processors added maintaining reasonable efficiency.

\[ S_p = \frac{1}{\beta + \frac{(1-\beta)}{p}} \]

where \( \beta \) is the serial part of the code which cannot be parallelized.

- In the above experiment: \((S_p = 16, p = 140)\) Therefore, \( \beta \) comes to = 5.5%.
- So according to Amdahl's Law 5.5% of code will never be parallelized.
- For input size of 1 Billion keys we are able to strongly scale up to 32 nodes. After that the efficiency decreases dramatically.
Gustafson's law
Gustafson's law

As we increase the processor, we are able to solve bigger and bigger problems thus, achieving weak scaling:

\[ S_p = p - \alpha (p - 1) \]

\[ \alpha = \frac{T_{seq}}{T_{seq} + T_{par}} \]

*Substituting the values from above experiment: \( T_{seq} = 1.4, T_{par} = 9.6, \alpha = 0.127 \)*

\[ S_p = 14.05 \]

Therefore, the algorithm is weakly scalable at 16 nodes as N:P are in ratio.
Running without TBB (Inter MPI Tracer)
Running with TBB (Inter MPI Tracer)
# Slurm.sh

```bash
#!/bin/bash

#SBATCH --mem=64000
#SBATCH --exclusive
#SBATCH --constraint=IB
#SBATCH --job-name="702"
#SBATCH --partition=general-compute
#SBATCH --qos=general-compute
#SBATCH --account=zsayed
#SBATCH --output=/panasas/logs/bitonic_%a_%j.stdout
#SBATCH --error=/panasas/logs/bitonic_%a_%j.stderr
#SBATCH --time=00:10:00
#SBATCH --nodes=16
#SBATCH --ntasks-per-node=1

module load intel-oneapi-2021.3
module load intel-oneapi-mpi/2021.3.0
export I_MPI_PMI_LIBRARY=/usr/lib64/libpmi.so

module load intel-tbb/2019.3
source /util/academic/intel/19.3/compilers_and_libraries/linux/tbb/bin/tbbvars

srun --mpi=pmi2 /user/zsayed/projects/bitonic-sort/bin/bitonic 1000000000

#mpirun -trace /user/zsayed/projects/bitonic-sort/bin/bitonic 1000000000
```

### Bitonic sort

```bash
./build.sh

mpirun -np 8 ./bin/bitonic 1024
mpirun -np 8 <tau_exec> ./bin/bitonic 1024
mpirun -trace ./bin/bitonic 1024

sbatch slurm.sh
```

- install tau pzt to instrument and profile

```bash
export TAU_TRACE=1; export TAU_PROFILE=1; export TAU_COMM_MATRIX=1;
export TRACEDIR=./tau_trace; export PROFILEDIR=./tau_trace;

tau_tremerge.pl;
tau2slog2 tau.trc tau.edf -o tau.slog2;
jumshot tau.slog2
```

---

Repo: https://gitlab.com/zain_s/bitonic-sort
References

• Introduction to Parallel Computing Solutions Manual on the Web
  Grama, Gupta, Karypis & Kumar

Thank you!