BigSort

Summary Version
1.0

Purpose of Benchmark
The BigSort benchmark sorts a large number of 64-bit integers (from 0 to T) in parallel. In particular, the total size of the data set can exceed the aggregated memory size of all nodes. The goal is to exercise and study a computer system's memory hierarchy performance when it comes to big data management. The emphasis here is IO, all-to-all communication, and integer operations.

Characteristics of Benchmark

The BigSort benchmark has two major phases, as described below:

1. Hash all elements to N bins, where N = # of nodes. Note the benchmark uses 1 MPI rank per node. By the end of this phase, each bin contains all elements falling within a sub range. For example, the first bin contains numbers ranging from 0 to T/N. Each bin is then stored as a "bin file". In this phase, each rank performs three steps:
 a. Read a part (proportional to the DRAM size) of the input file (each rank reads a different part)
 b. Hash this part locally according to the bins defined above.
 c. Exchange elements among MPI ranks (AllToAll communication) to send them to the MPI rank that owns the corresponding bin.
 d. Each rank writes the received elements to its "bin file".

2. Sort each "bin file" individually in each node. (Note: we assume the numbers are uniformly distributed so there is little concern for load balancing). No MPI communications. Each MPI rank treats its bin file as a collection of chunks, each chunk is of page_size (page_size is recommended to be no smaller than the page size of the file system).
 a. Sort chunks in parallel (with OpenMP)
 b. Merge every M chunks, repeat until there is only 1 chunk left. This chunk is the sorted bin file
 Note: M = DRAM_size/page_size, so the number of merging stages is $\log_M$(T/N/page_size)

3. Concatenate all sorted bin files together into a single file.

The benchmark exercises OpenMP threads, MPI-IO, Posix IO, and all-to-all communications with both small and large message sizes. The majority of the time is spent in IO.

Mechanics of Preparing Input

There are two programs (GenSeq and ShuffleSeq) that are involved in preparing the input. Use GenSeq to generate a file containing numbers from 0 to T in ascending order. Then use ShuffleSeq to shuffle the numbers.

To compile either of them:
* make CLUSTER=<system>. For a list of systems available, see commons/Makefile. Add your own system if nothing matches.

Below are more details on how to run the benchmarks

GenSeq: Generate an ordered sequence of data from 0 to <num>, stored in <file_name>.
To run:

```
qsub -n <nodes> --mode <mode(e.g. c1 on Mira)> <binary> <num> <DRAM_ALLOC> <file_name>
```

<DRAM_ALLOC> is the amount of DRAM to be used for each rank.
<nodes> is the number of nodes to collectively generating the sequence

ShuffleSeq:
Shuffle the sequence of numbers stored in a disk file. It will overwrite the input file.
This is done by repeatedly bringing in scattered "blocks" of data from random locations
in the data file, shuffle them in the DRAM, and then write back to the disk.
Each round is named as a "stage".
To run:

```
qsub -n <nodes> --mode <mode(e.g. c1 on Mira)> <binary>  <num> <DRAM_ALLOC> <file_name>
<blks_per_rank> <stages>
```

<num> is the total number of elements in the file
<file_name> is the name of the file (used for both input and output)
<DRAM_ALLOC> is the amount of DRAM to be used for each rank.
<nodes> is the number of nodes to collectively generating the sequence
<blks_per_rank> is the number of random blocks of data to shuffle in a stage in each rank
<stages> is the number of stages

Mechanics of Building Benchmark

To compile, do make CLUSTER=<system> Debug=no OPENMP=yes
For a list of systems available, see commons/Makefile. Add your own system if nothing matches.

Mechanics of Running Benchmark

```
qsub -n <nodes> --mode c1 ./bigsort <num> <DRAM_ALLOC> <input_file> <output_file> <temp_dir>
<page_size>
```

<num> is the total number of elements in the file
<DRAM_ALLOC> is the amount of DRAM to be used for each rank.
<input_file> is the file containing the input data
<output_file> is the resulting sorted file
<nodes> is the number of nodes to collectively generating the sequence
<temp_dir> is the directory to store temporary, intermediate files
<page_size> The buffer size for file IO transfers

Example runs:

Create folders to store input, output, and temporary files

```
mkdir -p /gpfs/temp /test/temp
```

Generating an ascending sequence of numbers:

```
qsub  --mode c1  -n 16  ./genseq 4294967296 1073741824 /gpfs/temp/data/4294967296_unsorted.dat
```

Shuffle the sequence of numbers:

```
qsub  --mode c1  -n 16  ./shuffleseq 4294967296 1073741824 /gpfs/temp/data/4294967296_unsorted.dat
16 64
```

Build the binary for bigsort:

```
make CLUSTER=Vesta Debug=no OPENMP=yes
```

Start sorting:

```
qsub  --mode c1  -n 16 --env OMP_NUM_THREADS=32 ./bigsort 4294967296 1073741824
/gpfs/temp/data/4294967296_unsorted.dat /gpfs/temp/test/sorted.data /gpfs/temp/test/temp 8388608
```

Test problem:
<num>: 137438953472 (the number of 64-bit integers, totally 1TB data)
<nodes>: 512
<DRAM_ALLOC>: 1073741824 (1GB)
<page_size>: 8388608 (8MB)

Coral class problem:
There are two problem dataset sizes needed. The total size of the 64-bit integers (<num>*8) should be:
1) 12 PB
2) 3X the total system memory
Other parameters can vary according to the system's preferences.

With regard to I/O management, the input dataset will be read from the storage system, and the final sorted output should be on the storage system. The user is free to modify the code as appropriate.

Verification of Results

If the stdout file contains the line "Ordered=1", then the sorting is verified.