WIKIPEDIA

Polyphase merge sort

A polyphase merge sort is a variation of bottom up Merge sort that sorts a list using an initial uneven distribution of sub-lists (runs), primarily used for external sorting, and is more efficient than an ordinary merge sort when there are less than 8 external working files (such as a tape drive or a file on a hard drive). A polyphase merge sort is not a stable sort.

Contents

Ordinary merge sort

A merge sort splits the records of a dataset into sorted runs of records and then repeatedly merges sorted runs into larger sorted runs until only one run, the sorted dataset, remains.

An ordinary merge sort using four working files organizes them as a pair of input files and a pair of output files. The dataset is distributed evenly between two of the working files, either as sorted runs or in the simplest case, single records, which can be considered to be sorted runs of size 1. Once all of the dataset is transferred to the two working files, those two working files become the input files for the first merge iteration. Each merge iteration merges runs from the two input working files, alternating the merged output between the two output files, again distributing the merged runs evenly between the two output files (until the final merge iteration). Once all of the runs from the two inputs files are merged and output, then the output files become the input files and vice versa for the next merge iteration. The number of runs decreases by a factor of 2 at each iteration, such as 64, 32, 16, 8, 4, 2, 1. For the final merge iteration, the two input files only have one sorted run (1/2 of the dataset) each, and the merged result is a single sorted run (the sorted dataset) on one of the output files. This is also described at Merge sort#Use with tape drives .

If there are only three working files, then an ordinary merge sort merges sorted runs from two working files onto a single working file, then distributes the runs evenly between the two output files. The merge iteration reduces run count by a factor of 2, the redistribute iteration doesn't reduce run count (the factor is 1). Each iteration could be considered to reduce the run count by an average factor of the square root of 2 ~= 1.41. If there are 5 working files, then the pattern alternates between a 3 way merge and a 2 way merge, for an average factor of square root of 6 ~= 2.45.

In general, for an even number N working files, each iteration of an ordinary merge sort reduces run count by a factor of N/2, or for an odd number O working files, each iteration reduces the run count by an average factor of square_root((O^2-1)/4).

Polyphase merge

For N less than 8 working files, a polyphase merge sort achieves a higher effective run count reduction factor by unevenly distributing sorted runs between N-1 working files (explained in next section). Each iteration merges runs from N-1 working files onto a single output working file. When the end of one of the N-1 working files is reached, then it becomes the new output file and what was the output file becomes one of the N-1 working input files, starting a new iteration of polyphase merge sort. Each iteration merges only a fraction of the dataset (about 1/2 to 3/4), except for the last iteration which merges all of the dataset into a single sorted run. The initial distribution is set up so that only one input working file is emptied at a time, except for the final merge iteration which merges N-1 single runs (of varying size, this is explained next) from the N-1 input working files to the single output file, resulting in a single sorted run, the sorted dataset.

For each polyphase iteration, the total number of runs follows a pattern similar to a reversed Fibonacci numbers of higher order sequence. With 4 files, and a dataset consisting of 57 runs, the total run count on each iteration would be 57, 31, 17, 9, 5, 3, 1.[1][2] Note that except for the last iteration, the run count reduction factor is a bit less than 2, 57/31, 31/17, 17/9, 9/5, 5/3, 3/1, about 1.84 for a 4 file case, but each iteration except the last reduced the run count while processing about 65% of the dataset, so the run count reduction factor per dataset processed during the intermediate iterations is about 1.84 / 0.65 = 2.83. For a dataset consisting of 57 runs of 1 record each, then after the initial distribution, polyphase merge sort moves 232 records during the 6 iterations it takes to sort the dataset, for an overall reduction factor of 2.70 (this is explained in more detail later).

After the first polyphase iteration, what was the output file now contains the results of merging N-1 original runs, but the remaining N-2 input working files still contain the remaining original runs, so the second merge iteration produces runs of size (N-1) + (N-2) = (2N - 3) original runs. The third iteration produces runs of size (4N - 7) original runs. With 4 files, the first iteration creates runs of size 3 original runs, the second iteration 5 original runs, the third iteration 9 original runs and so on, following the Fibonacci like pattern, 1, 3, 5, 9, 17, 31, 57, ... , so the increase in run size follows the same pattern as the decrease in run count in reverse. In the example case of 4 files and 57 runs of 1 record each, the last iteration merges 3 runs of size 31, 17, 9, resulting in a single sorted run of size 31+17+9 = 57 records, the sorted dataset. An example of the run counts and run sizes for 4 files, 31 records can be found in table 4.3 of.[3]

Perfect 3 file polyphase merge sort

It is easiest to look at the polyphase merge starting from its ending conditions and working backwards. At the start of each iteration, there will be two input files and one output file. At the end of the iteration, one input file will have been completely consumed and will become the output file for the next iteration. The current output file will become an input file for the next iteration. The remaining files (just one in the 3 file case) have only been partially consumed and their remaining runs will be input for the next iteration.

File 1 just emptied and became the new output file. One run is left on each input tape, and merging those runs together will make the sorted file.

```
File 1 (out):                                         <1 run> *       (the sorted file)
File 2 (in ): ... | <1 run> *              -->     ... <1 run> | *          (consumed)
File 3 (in ):     | <1 run> *                         <1 run> | *          (consumed)

...  possible runs that have already been read
|    marks the read pointer of the file
*    marks end of file
```

Stepping back to the previous iteration, we were reading from 1 and 2. One run is merged from 1 and 2 before file 1 goes empty. Notice that file 2 is not completely consumed—it has one run left to match the final merge (above).

```
File 1 (in ): ... | <1 run> *                      ... <1 run> | *
File 2 (in ):     | <2 run> *           -->            <1 run> | <1 run> *
File 3 (out):                                          <1 run> *
```

Stepping back another iteration, 2 runs are merged from 1 and 3 before file 3 goes empty.

```
File 1 (in ):     | <3 run>                        ... <2 run> | <1 run> *
File 2 (out):                              -->         <2 run> *
File 3 (in ): ... | <2 run> *                          <2 run> | *
```

Stepping back another iteration, 3 runs are merged from 2 and 3 before file 2 goes empty.

```
File 1 (out):                                          <3 run> *
File 2 (in ): ... | <3 run> *              -->     ... <3 run> | *
File 3 (in ):     | <5 run> *                          <3 run> | <2 run> *
```

Stepping back another iteration, 5 runs are merged from 1 and 2 before file 1 goes empty.

```
File 1 (in ): ... | <5 run> *                      ... <5 run> | *
File 2 (in ):     | <8 run> *              -->         <5 run> | <3 run> *
File 3 (out):                                          <5 run> *
```

Distribution for polyphase merge sort

Looking at the perfect 3 file case, the number of runs for merged working backwards: 1, 1, 2, 3, 5, ... reveals a Fibonacci sequence. The sequence for more than 3 files is a bit more complicated; for 4 files, starting at the final state and working backwards, the run count pattern is {1,0,0,0}, {0,1,1,1}, {1,0,2,2}, {3,2,0,4}, {7,6,4,0}, {0,13,11,7}, {13,0,24,20}, ...

For everything to work out optimally, the last merge phase should have exactly one run on each input file. If any input file has more than one run, then another phase would be required. Consequently, the polyphase merge sort needs to be clever about the initial distribution of the input data's runs to the initial output files. For example, an input file with 13 runs would write 5 runs to file 1 and 8 runs to file 2.

In practice, the input file will not have the exact number of runs needed for a perfect distribution. One way to deal with this is by padding the actual distribution with imaginary "dummy runs" to simulate an ideal run distribution.[4] A dummy run behaves like a run with no records in it. Merging one or more dummy runs with one or more real runs just merges the real runs, and merging one or more dummy runs with no real runs results in a single dummy run. Another approach is to emulate dummy runs as needed during the merge operations[5].

"Optimal" distribution algorithms require knowing the number of runs in advance. Otherwise, in the more common case where the number of runs is not known in advance, "near optimal" distribution algorithms are used. Some distribution algorithms include rearranging runs.[6] If the number of runs is known in advance, only a partial distribution is needed before starting the merge phases. For example, consider the 3 file case, starting with n runs on File_1. Define fib(i) as the "ith" Fibonacci number, where fib(i) = fib(i-1) + fib(i-2). If n = fib(i), then move fib(i-2) runs to File_2, leaving fib(i-1) runs remaining on File_1, a perfect run distribution. If fib(i) < n < fib(i+1), move n-fib(i) runs to File_2 and fib(i+1)-n runs to File_3. The first merge iteration merges n-fib(i) runs from File_1 and File_2, appending the n-fib(i) merged runs to the fib(i+1)-n runs already moved to File_3. File_1 ends up with fib(i-2) runs remaining, File_2 is emptied, and File_3 ends up with fib(i-1) runs, again a perfect run distribution. For 4 or more files, the math is more complicated, but the concept is the same.

Comparison versus ordinary merge sort

After the initial distribution, an ordinary merge sort using 4 files will sort 16 single record runs in 4 iterations of the entire dataset, moving a total of 64 records in order to sort the dataset after the initial distribution. A polyphase merge sort using 4 files will sort 17 single record runs in 4 iterations, but since each iteration but the last iteration only moves a fraction of the dataset, it only moves a total of 48 records in order to sort the dataset after the initial distribution. In this case, ordinary merge sort factor is 2.0, while polyphase overall factor is ~2.73.

To explain how the reduction factor is related to sort performance, the reduction factor equations are:

```
reduction_factor = exp(number_of_runs*log(number_of_runs)/run_move_count)
run_move_count = number_of_runs * log(number_of_runs)/log(reduction_factor)
run_move_count = number_of_runs * log_reduction_factor(number_of_runs)
```

Using the run move count equation for the above examples:

- ordinary merge sort → $16 \times \log_2(16) = 64$,
- polyphase merge sort → $17 \times \log_{2.73}(17) = 48$.

Here is a table of effective reduction factors for polyphase and ordinary merge sort listed by number of files, based on actual sorts of a few million records. This table roughly corresponds to the reduction factor per dataset moved tables shown in fig 3 and fig 4 of polyphase merge sort.pdf (http://www.computer.org/csdl/proceedings/afips/1960/5057/00/50570143.pdf)

```
# files
|     average fraction of data per iteration
|     |     polyphase reduction factor on ideal sized data
|     |     |     ordinary reduction factor on ideal sized data
|     |     |     |
3     .73   1.94  1.41  (sqrt  2)
4     .63   2.68  2.00
5     .58   3.20  2.45  (sqrt  6)
6     .56   3.56  3.00
7     .55   3.80  3.46  (sqrt 12)
8     .54   3.95  4.00
9     .53   4.07  4.47  (sqrt 20)
10    .53   4.15  5.00
11    .53   4.22  5.48  (sqrt 30)
12    .53   4.28  6.00
32    .53   4.87 16.00
```

In general, polyphase merge sort is better than ordinary merge sort when there are less than 8 files, while ordinary merge sort starts to become better at around 8 or more files.[7][8]

References

1. Donald Knuth, The Art of Computer Programming, Volume 3, Addison Wesley, 1973, Algorithm 5.4.2D.
2. http://oopweb.com/Algorithms/Documents/Sman/Volume/ExternalSorting.html
3. "Archived copy" (https://web.archive.org/web/20160128191110/http://pluto.ksi.edu/~cyh/cis501/ch4.htm). Archived from the original (http://pluto.ksi.edu/~cyh/cis501/ch4.htm) on 2016-01-28. Retrieved 2016-01-22.
4. Knuth
5. https://www.fq.math.ca/Scanned/8-1/lynch.pdf
6. http://i.stanford.edu/pub/cstr/reports/cs/tr/76/543/CS-TR-76-543.pdf
7. http://bluehawk.monmouth.edu/rclayton/web-pages/s06-503/esort.html
8. http://www.mif.vu.lt/~algis/dsax/DsSort.pdf

Further reading

- Bradley, James (1982), *File and Data Base Techniques* (https://archive.org/details/filedatabasetech0000brad), Holt, Rinehart and Winston, ISBN 0-03-058673-9
- Reynolds, Samuel W. (August 1961), "A generalized polyphase merge algorithm", *Communications of the ACM*, New York, NY: ACM, **4** (8): 347–349, doi:10.1145/366678.366689 (https://doi.org/10.1145%2F366678.366689)
- Sedgewick, Robert (1983), *Algorithms* (https://archive.org/details/algorithms00sedg/page/163), Addison-Wesley, pp. 163–165 (https://archive.org/details/algorithms00sedg/page/163), ISBN 0-201-06672-6

External links