N90-16446

SORT COMPUTATION

John E. Dorband

NASA/Goddard Space Flight Center/635
Greenbelt, MD 20771

ABSTRACT

Sorting has long been used to organize data in preparation for further computation, but sort computation allows some types of computation to be performed during the sort. *Sort aggregation* and *sort distribution* are the two basic forms of sort computation. Sort aggregation generates an accumulative or aggregate result for each group of records and places this result in one of the records. An aggregate operation can be any operation that is both associative and commutative, i.e. any operation whose result does not depend on the order of the operands or the order in which the operations are performed. Sort distribution copies the value from a field of a specific record in a group into that field in every record of that group.

Keywords: Sorting, Aggregation, Distribution, SIMD, Massively Parallel, Data Parallel, MPP, Routing.

INTRODUCTION

Sort computation uses sorting as a control mechanism to support interspersed routing and data manipulation. Sort computation is performed on sets of records, grouped according to a key contained in each record. Groups of records contain only records that have been determined to be equal by some function. The sort computation technique which has been developed here is simple. View a sorting algorithm as having two parts — the comparison of records and the routing of records. The comparison determines if the two records are in the correct order. Routing takes this result and determines where each of the records is to go next. Thus, the sort contains a routing and a comparison routine, where the routing routine calls the comparison routine when necessary. All sort algorithms, such as merge sort, bubble sort, and bitonic sort, consist of these two parts. Sort computation can use the routing part of any sort algorithm. The routing routine only determines the order in which the records finally line up after the sort is through — not how they are modified. The comparison routine, on the other hand, is replaced with a comparison routine whose nature depends on the type of sort computation it is to perform. The comparison routine contains the code that determines how the contents of the records are changed. The comparison routine has two functions. One function is to determine if the two records being compared are in the same group (generally whether or not their keys are equal), whether a record from one group will come before or after a record from another group, and in some cases if the sort is complete. The other function is to modify the records if they both belong to the same group.

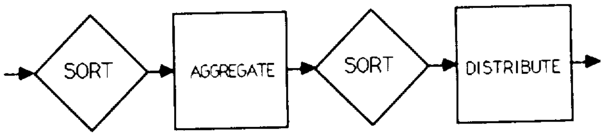

Figure 1. Conventional use of sorting to organize data in preparation for computation.

Sorting has long been used to organize data in preparation for further computation (Figure 1), but sort computation allows some types of computation to be performed during the sort (Figure 2). *Sort aggregation* and *sort distribution* are the two basic forms of sort computation. Sort aggregation generates an accumulative or aggregate result for each group of records and places this result in one of the records. Usually, it is placed in the last record or the one with the largest key value. An aggregate operation can be any operation that is both associative and commutative, i.e. any operation whose result does not depend on the order of the operands or the order in which the operations are performed. Addition, multiplication, AND, OR, and EXCLUSIVE-OR are examples of valid operations. Sort distribution copies the value from a field of a specific record in a group into that field in every record of that group. The record that contains the value to be distributed contains a flag that is set to true. Note

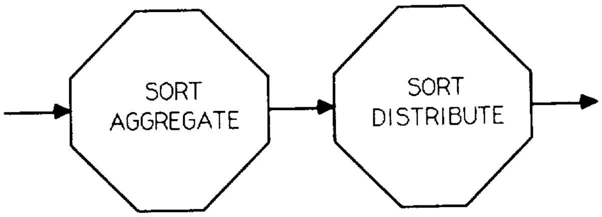

Figure 2. Sort computation allows some types of computation to be performed during the sort.

that there may be more than one record in a group in which the flag is set, as long as all such records contain the same value.

SORT AGGREGATION

Sort aggregation is described here with pseudo code and a proof is given to show that sort aggregation computes the aggregate result for each group of records within the set of records being sorted. The expression "A[5].(B,C,D)" defines an array of 5 records, where each record has 3 fields, B, C, and D. The terms, sum or summing, are used as the generic terms for finding the result of a valid aggregation operation. Thus, the command "SORT(SUM,A)" performs the sort sum over the array A defined by "A[n].(K,V)". Note that the sum operation can be replaced by any other valid aggregate operation.

SUM (Figure 3) is the comparison routine that will, when used in conjunction with a sort routine, sum all the values in field V of the records for which the K fields are equal. SUM returns a value of true if the records A1 and A2 are in the correct order, and false if they are not. SUM puts the sum of all the V fields of records of the same group in the last (or largest) record in the group.

```
boolean function SUM(A1,A2)
        given A1.(K,V)
        given A2.(K,V)
        if A1.K = A2.K then
                A1.V = 0
                A2.V = A1.V + A2.V
                return(true)
                end if
        if A1.K < A2.K then
                return(true)
                end if
        if A1.K > A2.K then
                return(false)
                end if
end function
```

Figure 3. SUM routine.

The proof that aggregation works as described goes as follows. Even though the keys of the records being compared may be equal, SUM can affect their ordering by returning the response to the routing routine that the records are in the correct order (true) or not (false). This in effect gives order within a group. SUM always designates the record that contains the result of the sum as the larger of the two records, the larger contains a value of zero. This means that the sum of the value fields of the group's records will be contained in the record that was designated larger than all others. Assume, however, that not all values of records in a specific group were summed into the same record. This means that at least two records contain only part of the result for that group. Each one of these records would have been designated greater than all records of that group. Yet, the records that contained partial results must not have been compared to any others or the partial results would have been summed into it. Thus, each record would have been designated the largest in the group. Because only one record is the largest of a group, there can only be one record that contains the result for any group.

A comparison routine such as SUM can be written for any operation that is both associative and commutative, as described previously.

SORT DISTRIBUTION

Sort distribution is slightly more complex than sort aggregation and is constrained somewhat compared to sort aggregation. The constraint stems from the fact the result of a sort distribution must be migrated to all the members of a group of records while the result of a sort aggregation only needs to migrate to one record of a group of records. This constraint will be clarified further after the proof.

```
boolean function COPY(A1,A2)
        given A1.(K,F,V)
        given A2.(K,F,V)
        if A1.K = A2.K then
                if A2.F then
                        A1.V = A2.V
                        A1.V = true
                        return(true)
                        end if
                if A1.F then
                        A2.V = A1.V
                        A2.V = true
                        return(true)
                        end if
        else
                return(true)
                end if
        if A1.K < A2.K then
                return(true)
                end if
        if A1.K > A2.K then
                return(false)
                end if
end function
```

Figure 3. COPY routine.

The idea in sort distribution is to copy the value of a record in a group of records, which has been flagged as having a valid value for that group, to all records that do not already have that value. The command to perform this is "SORT(COPY,A)", where SORT is a routing routine, COPY is a comparison routine, and A is an array of records. This array of *n* records is of the form "A[n].(K,F,V)", where K is the key, F is the valid value flag, and V is the value field. COPY used in conjunction with SORT distributes the flagged value in each group to all members of the group (see Figure 4). Like SUM, COPY returns a value of true if the records A1 and A2 are in the correct order, and false if they are not. COPY puts the same value in all records of the same group, or no value at all if no record of the group had its valid value flag set prior to performing the distribution.

The proof that distribution can be accomplished during a sort is similar to that of aggregation. Note that when two records are determined to be in the same group, and one of the records contains a valid value, it is copied to the other record and its valid value flag is set. This, in effect, causes the record with a valid value to be considered both larger and smaller than a record that *does not have a valid value*. Thus, at the completion of the sort computation, at least the largest and smallest record of each group that had a record with a valid value will contain a valid value. Assume that a record without a valid value remained after the sort was completed. If it was either the largest or the smallest record of the group, then no other record in the group had a valid value. If it was not the smallest or the largest value of the group, either there was no record in the group with a valid value, or it was not compared to a record in the group with a valid value. If there is a record without a valid value and one with a valid value in the same group, such a pair exists logically next to each other and has never been compared. If such a pair exists, there is no way of knowing which one is larger, since they have never been compared. Thus, the sort must not have been completed. Therefore, a record can only be left without a valid value if there are no records in its group with a valid value when the sort is complete.

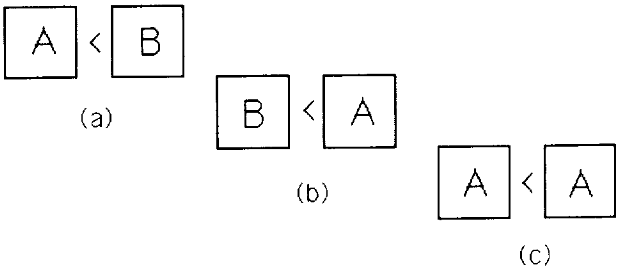

Figure 5. (a) Record A is smaller than B.
(b) Record A is larger than B.
(c) Record A is both larger and smaller than

The statement in the proof written in italics is the key to whether a sort algorithm can be used to perform sort distribution. The distribution record must be seen as being both larger and smaller than the replaced record. Figure 5(a) shows the order of records A and B if A is smaller than B, 5(b) shows the order if A is larger than B, but in 5(c) A appears to be both larger and smaller than B by replacing B with A. If the solution to the fact that two records are out of order is simply that they need to be swapped, then A may be made to appear to be both larger and smaller than B by simply replacing B with A. This is the case with *merge*, *bubble*, and *bitonic sorts*, for example. But this is not the case however with *insertion sorts* that use a log *n* time insertion. In an insertion sort, one of the records being compared has already found its position in the list. Therefore, it is not the case that if the records being compared are out of order, they are simply swapped. Such a sort may be extensively modified to support sort distribution, but it might be more effective to just use a sort that needs no modification.

GENERALIZATION OF SORT COMPUTATION

Sort computation requires that records of data be grouped according to some criterion. Order merely forces this grouping to occur. Thus, any function that causes the desired grouping may be used to perform the comparison part of the sort. The function used for comparison must evaluate to one of three results, *less than*, *greater than*, or *equal to*, depending on the two records that are being compared. The *data values* need not literally be less than, greater than, or equal to, as long as the end result is an unambiguous ordering that causes the desired grouping of records that are designated as equal.

Records can be grouped, for instance, as a set of non-overlapping ranges. In this case, the conditions of the comparison function would be lower than the minimum of the range, higher than the maximum of the range, or within the range. Range ordering uses two types of records — records whose keys are ranges and records whose keys are single values. Note that in the case of in-range ordering, once a record is determined to be in-range, not only must the appropriate action be performed on its aggregation or distribution fields, but the key field of the in-range record must be modified so that it becomes a range key rather than a single-value key record.

AGGREGATE DISTRIBUTION

Aggregate distribution differs from aggregation in that all member records of a group obtain the results of the aggregation instead of just one member. It uses a sort algorithm that is made up of merge steps, because the flag field must be set between each merge step. It is not known if aggregate distribution will work for sorts that are not made of merge steps. The following describes how a merge aggregate distribution is performed.

Start with two sorted lists of records A_i and B_j, where i=1...n and j=1...m. Each record contains a 2-bit flag. The flags of records in list A are set to 1 and the flags of records in list B are set to 2. During the merge, if two records are determined to be in the same group and one record's flag is 1 and the other's is 2, then the aggregate function is performed, both records are given the result, and their flags are set to 3. If one record's flag is 3 and the other's is not, then the aggregate result contained in the record with the flag value of 3 is copied to the other record. Otherwise, if both records' flags are the same, nothing is done to either record. When the merge is done, all records within a group have the same aggregate results.

OPTIMIZATION OF SORT COMPUTATION

Sorting is generally a very time consuming function, particularly on a single processor machine. However, on a multiple processor machine such as the Massively Parallel Processor[1] (MPP), a sort of 65536 records of 32 bits each takes about 29 milliseconds, and a sort of 512K records of 32 bits each takes about 1 second. This is very fast but still time-consuming if it is meant to be used very often, as may be the case with sort computation. The time needed to perform the *necessary* sort computation can be minimized in several ways — in either hardware or software. Hardware can be improved by decreasing the interprocessor communication time or using a more far-reaching or elaborate processor interconnection scheme such as a complete hypercube. Hardware improvements to sorting or sort computation will not be discussed here. However, methods to improve sort computation performance through the use of prudent software design and programming techniques have been developed by the author and are discussed next.

A bitonic sort has been implemented by the author on the MPP. This sort and most other sorts, require log *n* merge steps, each merging two sorted lists into one sorted list. Time can be saved during sort computation by performing a merge only if certain conditions are met: the records in each list must be in an order and have values consistent with the result of a sort computation performed on each list. An example of this is the use of sort computation for table look-up. The table can be sorted once before it is used. Therefore, it need only be *merged* with sorted data records when its values are to be distributed to the data records, rather than having to be *sorted* into the data records.

If it is necessary to extract table records out from among the data records to complete a table look-up, the table records could be sorted using a major key that distinguishes them from data records. This, however, defeats the use of the merge to combine the two record types because the sort takes so much longer than the merge. The records can be unmerged in no more time than it takes to merge them by leaving a "trail of corn", so to speak. During the merge, a set of log *n* bits in each processor is used to record whether or not the pair of records in that processor are exchanged during each of the log *n* comparison steps of the merge. This set of bits is then used during the unmerge operation to route the records back to their original locations.

Another means of reducing the time spent in sort computation is to, at times, perform only partial or local sorting of the data. This has shown to be useful during image registration[2] when records are being generated whose values need to be accumulated. Each original pixel in the image is divided into much smaller subpixels. These subpixels carry a fraction of the original pixel's value and a calculated new position. The subpixel values are then summed into their new pixel's value using sort summing. Since the subpixels are likely to be summed with nearby subpixels, many small local sort sums are performed to accumulate as much as possible locally before sort summing across the entire image. This saves space in the processor memory, as well as saving time.

Partial sorting can also be used when a table look-up needs to be performed and the size of the table is much smaller than the number of data records to which the table information is to be distributed. In this case, multiple copies of the table are distributed across the processors, allowing the use of smaller sort distribution operations confined to local areas of the array of processors.

Sort computation can be made faster simply by using a faster sort algorithm. This is interesting because where the records were before the sort and where the records end up after the sort is irrelevant. This allows sorts to be used that leave the records in unusual orders, such as snake row major or shuffle row major, if they are faster.

In the case of sort distribution, if it is known that all records either contain a value or will obtain a valid value during the sort operation, a check can be performed after every comparison step to see if the sort distribution has been completed. Thus, the sort distribution may be terminated before the sort is actually complete.

To extend this concept one step further, it may not be necessary for any arbitrary record to obtain the value it is looking for in any given invocation of the sort computation. Therefore, many local sort operations may be performed to get some local sort computation done quickly between successive complete sort operations. This brings up an issue for further study: can the keys used in the sort operations be generated for records that are created between sort computation operations, so as to minimize the number of complete sort computation operations that need to be performed.

AN EXAMPLE OF SORT COMPUTATION

Multiplication of a sparse matrix times a vector is now presented as an example of sort computation. This is presented as an iterative refinement of the vector V ($V_{i+1} = M*V_i$). The form of the record used is "T.(R,C,M,V)", where T has four fields: the row R, the column C, the matrix coefficient at row R and column C, and the vector coefficient at position C of the vector. To perform a matrix multiply, first multiply M times C in each record, giving new record values "T.(R,C,M,V=M*V)". A sort sum operation is performed using R as the key and summing over M*V. This leaves one record for each R which contains the value of the new vector at position R. At this point the matrix multiply is complete, but if further iterations must be perform the new vector coefficients must be distributed so the value of V corresponds to the value of C, not of R. This is done by making another set of records "T_1.(R_1=C,C_1=R,M_1=M,V_1)" which contains a record for every record in "T". V_1 has been given no value yet. Then form a set of records that is the union of T and T_1. A sort distribution is performed using R as the key and distributing the values of V from T to T_1. All record of T are deleted and a new set of records for T are created of the form "T.(R=C_1,C=R_1,M=M_1,V=V_1)" from the records of T_1. Another matrix multiply may now be performed since the values of V correspond to the columns of M.

VIRTUAL LOCALITY

Virtual memory and *virtual processors* have become common concepts. The concept of virtual memory allows the programmer to imagine that there is as much memory as needed, alleviating the need to account for physical memory constraints in designing a program. It also allows him to imagine that he has complete control of all physical memory. This concept is used in most large computers, minicomputers, and the newest 32-bit microcomputers. The concept of virtual processors allows the programmer to view a problem as though it was executing on as many processors as needed, yet it may be using fewer processors[3].

The key point here is that virtual memory addresses are not physical addresses, but appear to be, and virtual processors are not real processors, but also appear to be. The programmer must still deal with addresses and fixed locations of data and the knowledge that he is using one or several processors. The programmer still has to deal with a hardware view of his computational environment, that of memory and processing units, in spite of the fact that it is a virtual hardware view. The author has developed the concept of *virtual locality* to move the programmer further away from hardware architecture concerns and closer to the perception of a computationally pure environment. This is especially important, as well as particularly feasible, when it comes to massively parallel architectures, such as the MPP.

Virtual locality views data in computational units of records. Computations are carried out on the data of these records according to the groups to which the records belong (their locality). These records contain fields, as in any traditional view of data records. Groups can contain any number of records. Records are grouped according to any number of schemes, based on field values. Because all computation is dependent only on the values within the records and the interrelationship of those values, the computation is independent of the record's location in the computation environment before, during, and after the computation and, it is also independent of the number of processors used to accomplish the computation. This differs from the view of computation in other massively parallel architectures that use more traditional routing schemes, or, for that matter, any computer architecture that depends on pointers or fixed addresses to direct data to and from specific locations in the environment. Virtual locality facilitates position-independent computation. It only matters that the appropriate data comes together sometime during the computational step. Generalized routing schemes and traditional memory addressing schemes require that data is placed where it can later be found. Therefore, it has to be allocated space and can only be moved after all places that refer to it have been changed. This makes dynamic allocation, reallocation, de-allocation and garbage collection difficult, if not impossible in some circumstances. Position dependent computation is used in the implementation of virtual locality, but is not seen by the programmer. With virtual locality, records of data may be created and deleted at will without allocating them to specific locations in the environment. Virtual locality is possible through the use of the sort computation concept[2]. Sort computation defines the types of operations supported under virtual locality and describes how they are implemented.

APPLICATION OF SORT COMPUTATION

Currently, image rotation, image registration, and computer graphic generation by ray tracing have been implemented by the author on the MPP using sort computation techniques. Three-dimensional rendering of elevation maps has also been implemented on the MPP using these techniques by a NASA summer student, Jennifer Trainer, under the direction of the author. However other applications exist that require the processing of irregular arrangements of data. For example, the implementation of pure LISP, which was designed and implemented by Tim Busse of Science Applications Research and the author, requires this capability.

The pure LISP is implemented by distributing the pointer pairs that make up the LISP data structure across the processors of the MPP. Sort computation is used to bring the pointer pairs together according to the functions that must be performed on them, such as the creation of a new pointer. The basic functions of pure LISP were implemented (i.e., CAR, CDR, CONS, EQ, ATOM, COND, APPLY, EVAL, EVLIST, and LAMBDA). The MPP ray tracing approach[4] is based on an algorithm that finds the intersections of light rays and objects in a 3-dimensional space. It is done by recursively subdividing space. Records are created that keep track of whether a specific ray or object intersects a subdivision of space. If a subdivision of space is not intersected by both a ray and an object, all records associated with it are deleted. Sort computation is used to determine where this condition is true. These two applications have been implemented on the MPP using MPP Parallel FORTH.

CONCLUSION

Future plans in the area of application of sort computation include the study of its use on data bases and for implementation of a compiler inside the MPP array. Virtual locality is worthy of further study also because it allows the simultaneous development of parallel algorithms and hardware architectures, requiring only a minimal amount of effort to port and test previously developed algorithms on new architectures. Sort computation is a feasible means of facilitating virtual locality. As with other virtual concepts, care must be taken, while knowledge about it's effective use and implementation in both software and hardware develops.

REFERENCES

(1) *The Massively Parallel Processor*, J.L. Potter, ed., ISBN: 0-262-16100, The MIT Press, Cambridge, MA, 1985.

(2) Dorband, John E., *Sort Computation and Conservative Image Registration*, Ph.D. thesis, Pennsylvania State Univ., December 1985.

(3) Hillis, W. Daniel, *The Connection Machine*, ISBN: 0-262-08157-1, The MIT Press, Cambridge, MA, 1985, p. 135.

(4) Dorband, John E., *3-D Graphic Generation on the MPP*, Proceedings of the 2nd International Conference on Supercomputing, Vol. II, pg 305-309, 1987.