MULTI-JOIN ON PARALLEL PROCESSORS

S.M. Deen, D.N.P. Kannangara, M.C. Taylor
Department of Computer Science
University of Keele
Keele

ABSTRACT

The paper describes a preliminary evaluation of some multi-join strategies and their performances on parallel hardware. The hardware used was a Sequent (under UNIX) with 11 usable processors, each with shared and private primary memory. A multi-join was broken down into a series of single joins which were then allocated to clusters, each cluster being a collection of parallel processors. The results of single joins, which were studied by both binary search and hash-merge techniques, were then further processed as necessary.

The evaluation was conducted varying a number of parameters, such as cluster size, tuple size and cardinality. The comparative results were plotted. The study highlights the importance of a number of factors that influence the performance of a multi-join operation.

1. INTRODUCTION

A join is a most expensive relational operation. A multi-join consists of a series of joins, and is much used in applications that process complex objects or objects made up of repeating groups, such as those required in AI applications, engineering drawings, graphics and molecular modelling. A graphical picture, for instance, might require hundreds of joins, and consequently the creation of such a picture takes a long time, mainly because of poor multi-join performance; some applications take several hours to form a picture. Therefore any improvement that can be made in multi-join performance will be a benefit.

Relational databases are typically designed for single joins; their optimisation techniques are not geared to handle multi-joins with many joins. The objective of this paper is to design strategies and evaluate performance of multi-joins on parallel hardware using only primary memory. The machine we used was a Sequent with 11 usable parallel processors. Our basic strategy was to distribute a multi-join as a set of single joins over a set of clusters, with one processor acting as a global controller. Each cluster consists of a number of parallel processors, ideally with one of its processors acting as a local controller. However, because of small cluster size we did not use any local controllers, but made the global controller carry out the work of the local controllers; this had a small overhead which we ignored. We varied the cluster size from 3 to 10 processors, discounting the controller.

Each parallel process has a sequential phase where one processor (the parent, i.e. the controller) carries out the initial set up activities, such as the creation of the other parallel processes (child processes), the distribution of tuples to these processes, and so on. This sequential phase is very important, and should be kept as short as possible. Once the tuples are distributed the parallel processing can begin.

A number of researchers have developed join algorithms for database machines [1-7], some of them employing hash-merge techniques as we have done, but none of
them to our knowledge handle multi-joins on parallel processors. The two main aspects of a multi-join optimisation can be seen as

Query decomposition
Phase optimisation

The query decomposition consists of dividing the overall problem into a sequence of phases (each phase having one or more single joins in parallel), and determining the optimal order for carrying them out. The decomposer examines the different join strategies and chooses the best one, taking into account overheads and other factors such as cardinalities, selectivities of join attributes, sizes of intermediate relations, the number of read/write operations and so on. In contrast phase optimisation is concerned with the optimised execution of the decomposed multi-join operation in each phase, using in our case, parallel processors. In this paper we have studied only phase optimisation. The plan of the paper is as follows:

In section 2 we discuss the hardware environment, followed by a description of our techniques in section 3 and cost models in section 4. The result of our performance study is presented in section 5, with a conclusion in section 6.

2. HARDWARE CONFIGURATION

The Sequent Symmetry S81 at Keele runs under DYNIX (a version of UNIX), with 12 processors, of which 11 were used by us for multi-join processing, and one kept in reserve for failure diagnostics. Each processor is uniquely identified by a processor identification number.

When a process runs, it can create child processes by using a system function called fork. Since a fork function is relatively expensive (55 milliseconds), it is best to create all the needed processes for parallel processing at the beginning with a single invocation of the fork function. Another function can be used to kill a child process, and again it is often cheaper to leave a process idle, and then kill all the child processes by a single invocation of the kill function. A process can have both

shared and private data in the memory; shared data reduces data communication cost and data duplication.

There are three ways of scheduling tasks in DYNIX: pre-scheduling, static scheduling and dynamic scheduling. In pre-scheduling, tasks are distributed before parallel processing begins, whereas in static scheduling, the tasks are allocated during parallel processing in a predefined sequence. In dynamic scheduling, each process schedules its task during run-time by checking a task queue; this has additional overheads, and therefore is not necessarily the best strategy. We used pre-scheduling because of its low overhead.

There are a number of DYNIX utilities which are available for measuring the performance of a given program; the ones relevant to us are:

1. prof - This command reports the number of times each subroutine is called and the amount of time used by each subroutine. To obtain this execution profile, the program has to be compiled with a special option called -g. Then when the program is run, its profile information is accumulated in a file, which can later be interpreted by using the prof command.

2. gprof - This command is similar to prof, but it also yields profile data in the order of subroutine calls executed by the program. In addition, gprof also profiles standard library routines as well as user subroutines. The program to be profiled by gprof must be compiled using the -pg option.

3. Size - This command reports the amount of memory used by a program's machine code and data structures.

4. Time - This command reports the total execution time of a program in terms of both real and CPU time. The parallel version of time is invoked as ptime. Ptime yields the statistics on each of the processors, not just for the application as a whole.
A process can have both shared and private data. The shared data can be accessed by all the processes. Private data is accessible by only one process. A join can be carried out by different processes running in parallel. However if k parallel processes are used, the performance does not go up by a factor of k due to the following overheads:

(i) Parallel systems functions
(ii) Interprocess communications
(iii) Distribution of tuples in relation to processes
(iv) Additional read/write operations

The parallel systems functions are used to create and manipulate parallel processes, but they are expensive to use (see section 4). To reduce this overhead, we created as many processes as required at the beginning and did not terminate them until the program was completed. The interprocess communications can be reduced by using shared memory, which all processes can access; in Sequent the shared memory approach is faster than pipelining.

For a parallel join we distribute tuples to different processors; if this distribution is not even, some of the processors become idle, and this wastes resources. The idle processors cannot be terminated without incurring a further overhead of the parallel systems functions mentioned above - it is cheaper to leave them idle until the end of the processing, as we did. Since the tuples are unordered, we employ a hash function [8,9,11] - a linear hash function - based on the first three characters of the join attributes and a divisor to yield a remainder as the hash-value. This is not perhaps the best hash function one can use, but it was easy to use and we do not think that the use of a more complex function would have altered the results significantly. The main problem of hashing is that although it reduces distribution cost, it does not produce an even distribution. A possible solution is the division of the workload into several partitions (subarrays) and the dynamic scheduling of these partitions, several to a processor, for a relatively more even load, but this has scheduling overheads. We have studied this problem as shown in section 4, and finally opted for one partition for one process.

The read/write operations on subarrays are additional overheads in parallel processing. To reduce this overhead we keep only what we call reduced relations, i.e. the join attributes and tuple numbers (from the initial main arrays - see below) in these subarrays. Note that if a relation B joins with both relations A and C, it will have two join attributes: left and right attributes, both of which must be kept in the reduced relations to minimise further accesses to the main arrays. In our study we did not consider secondary storage, but instead assumed all relations to be in the primary memory, each relation in an array, referred to above as the initial main array, from which reduced relations are created. Intermediate join results (in successive joins) are also held as reduced relations, and are partitioned again for the next join by re-hashing as necessary on the new join attribute. Figure 1 shows the relationships between different arrays.

![Figure 1](image)

3. JOINING TECHNIQUES

Given that there are only 10 parallel processors (excluding one processor which acts as the controller), we considered only up to 6 relations, R1 to R6 in a multi-join
A multi-join can proceed with one or more of the following steps:

1. One-join in parallel \( R_1 \times R_2 \)
2. Two-linked joins in parallel \( (R_1 \times R_2) \times R_3 \)
3. Two-separate joins in parallel \( (R_1 \times R_2 \times R_3 \times R_4) \times R_5 \times R_6 \)
4. Three-separate joins in parallel \( (R_1 \times R_2 \times R_3 \times R_4 \times R_5) \times R_6 \)

One-join in parallel is a single phase operation in which all the ten processors are used as a single cluster to join the two relations. The two-linked join is a two-phase operation where the output of the first join (phase 1) is joined in the next phase with the third relations, again with a single cluster of 10 processors. A five-join operation can be carried out in two separate ways, but both in three phases, as indicated by steps 3 and 4. In fact a five-join operation

\[ R_1 \times R_2 \times R_3 \times R_4 \times R_5 \times R_6 \]

of six relations can be executed in a number of different ways, such as:

**Strategy A**

Phase 1: \( S_1 = R_1 \times R_2, \quad S_2 = R_4 \times R_5 \)
Phase 2: \( S_3 = S_1 \times R_3, \quad S_4 = S_2 \times R_6 \)
Phase 3: Result = \( S_3 \times S_4 \)

**Strategy B**

Phase 1: \( S_1 = R_2 \times R_3, \quad S_2 = R_4 \times R_5 \)
Phase 2: \( S_3 = S_1 \times R_1, \quad S_4 = S_2 \times R_6 \)
Phase 3: Result = \( S_3 \times S_4 \)

**Strategy C**

Phase 1: \( S_1 = R_2 \times R_3, \quad S_2 = R_5 \times R_6 \)
Phase 2: \( S_3 = S_1 \times R_1, \quad S_4 = S_2 \times R_4 \)
Phase 3: Result = \( S_3 \times S_4 \)

**Strategy D**

Phase 1: \( S_1 = R_1 \times R_2, \quad S_2 = R_3 \times R_4, \quad S_3 = R_5 \times R_6 \)
Phase 2: \( S_4 := S_1 \times S_2 \)
Phase 3: \( S_5 := S_4 \times S_3 \)

Strategies A, B and C are the meaningful permutations of step 3. Which of the strategies is the best should be decided by the query decomposer; for our purpose, we shall assume strategy A.

Once phase 1 is completed, phase 2 can proceed as in step 2; and therefore only the phase 1 is of interest to us. The difference between this phase 1 of step 3 and step 1 is the cluster size. Since there are two separate joins to be executed in parallel, each will have a cluster of only 5 processors, in contrast to one cluster of 10 processors as used in step 1. We can then compare the performance of executing two joins in parallel using two small clusters with that of executing the same two joins in series using a larger cluster.

Strategy D implies step 4; but again only phase 1 is different, since phases 2 and 3 are the phases 1 and 2 respectively of step 2. This phase 1 of step 4 has three one-joins to be executed in parallel, using three clusters with 4, 3 and 3 processors respectively. We should then be able to compare the performance of executing three joins in parallel using three small clusters with that of executing the same three joins in series using a larger cluster.

The two relations involved in each single join are usually termed as the object relation (or the first relation) and the target relation (or the second relation). For each tuple in the object relation, the target relation is searched for matching tuples. In all our evaluations, except where otherwise indicated, we kept the cardinality of the target relation fixed at 100 tuples, while varying the cardinality of the object relation from 100 to 10,000. Relations used were made up from the following well-known ones:

\[
\begin{align*}
S(S\#, \text{ SNAME, STATUS, CITY}) \\
P(P\#, \text{ PNAME, COLOR, WEIGHT, CITY}) \\
SP(S\#, P\#, \text{ QTY}) \\
J(J\#, \text{ JNAME, CITY}) \\
SPJ(S\#, P\#, J\#, \text{ QTY})
\end{align*}
\]

Note that we also studied the impact of the number of attributes on performance.
(1) One-join in Parallel

We assume that the two relations are held originally in two separate arrays in the main memory. We then distribute each relation over 10 subarrays, by hashing on the join attribute. The subarrays hold only reduced relations. Ten processes are then created, and each pair of matching subarrays are allocated to a process (see also the Conclusion).

Up to this point the work is sequential, and is carried out by a single processor, the controller. After this, the parallel processing can begin. The ten processors work in parallel, and each joins its pair of subarrays and writes onto intermediate subarrays (as reduced relations), which are subsequently used to create the final result array, shared by all the processors. There is a counter for each processor which holds the start-position for writing in the final array; this prevents over-writing. When all the child processors have completed their tasks, the controller kills them.

There are two separate ways in which the join operation can be carried out: binary search which requires sorting of at least one relation, and sequential search which does not require any sorting. In binary search, each process first sorts (by Quicksort) its own partition of the target relation, and then begins matching by binary search. It scans its partition of the object relation from the beginning, and for each tuple it finds the matching tuples in the target relation by binary search. In sequential search none of the partitions are sorted; for each tuple of the object partition, a full scan is made of the target partition for matching tuples. It is shown in section 4 (One-Join in Parallel) that under certain conditions the sequential search technique will outperform the binary search technique for a sufficiently small number of tuples (largely because of the sorting overhead), as was the case in our experimentation (see figure 2). Therefore we did not use binary search in the other steps (i.e. steps 2, 3 and 4, listed earlier) of our study.

(2) Two-Linked Join

Given three relations R1, R2 and R3, they can be joined in three different ways, namely (R1*R2)*R3, (R2*R3)*R1 or (R1*R3)*R2, as indicated earlier. We assume the order to be (R1*R2)*R3, which can be evaluated in two phases.

In the first phase, we evaluate S1 := R1*R2 as a single join in parallel, and in the second phase, we evaluate S1*R3, again as a single join in parallel (see also the Conclusion). S1 is a reduced relation, each S1-tuple containing a pair of tuple numbers and a join attribute for the next join. In the sequential part of the second phase, the controller re-hashes S1 on the new join attribute, and re-partitions and re-distributes it. The controller also hashes, partitions and distributes R3 for the second phase. S1 and R3 are then joined as a single join in parallel. A single cluster of ten processors is used in each phase.

(3) Two-separate Join

In the first phase, in which we are interested, we join S1 := R1*R2 and S2 := R4*R5, in parallel, each using only 5 processors. The controller hashes the four relations of these two single joins, produces 5 partitions from each relation, creates 10 parallel processes, allo-
cates 5 processes to each join and distributes partitions - a pair (object and matching target) of partitions to each process. The controller kills the child processes only when both the joins are completed.

(4). Three-separate Join

The phase 1 of this step, in which we are interested, has three single joins to be executed in parallel, with processors distributed as 4, 3 and 3 for the three clusters, one for each single join. The controller hashes the six relations into partitions, with 4 partitions for the relations of the first join, 3 partitions each for the relations of the second and third joins. A pair of relevant partitions is then allocated to each processor. The parallel processing then begins. The controller kills the child processes only when all the three joins are completed.

4. COST MODELS

We present here a number of algorithms used to estimate the execution costs of various join strategies. Their derivation is quite straightforward. The following parameters are used in the cost models:

Cardinalities of relations N1, N2, N3, ....

Time required to execute hash function h

Time required to compare two attributes values c

Time to write a tuple from one array to another w

Number of processors involved k

Number of distinct values in attribute A V(A)

Ratios of size of (Join attribute + tuple number) to tuple size p1, p2, p3, ....

One-Join in Parallel

If the join attribute for joining relations R1 and R2 is A, we assume that the number of tuples in the result relation is equal to N1*N2/V(A) [10]. This formula works well if the join attribute in one relation is the primary key and in the other a foreign key.

Algorithm with Binary Search

Time to hash tuples of source relation = h*N1
Time to hash tuples of target relation = h*N2
Time to write tuples in object relation to subarrays = w*N1*p1
Time to write tuples in target relation to subarrays = w*N2*p2
Time to sort (in parallel) the subarrays of the target relation by Quicksort = (c + 3*w/2)*N2/k log(N2/k)
Time to pick matching tuples in subarrays = (p1+p2)*c*(N1/k)*log(N2/k)
Time to write tuples to separate result subarrays = (p1+p2)*(N1*N2*2*w)/(k*V(A))
Time to write tuples to result array = (N1*N2*2w)/(k*V(A))

(The presence of factor 2 in the above two expressions makes the length of a joined tuple double the original)

Total Time

Algorithm with Sequential Search

Time to hash tuples of source relation = h*N1
Time to hash tuples of target relation = h*N2
Time to write tuples in object relation to subarrays = w*N1*p1
Time to write tuples in target relation to subarrays = w*N2*p2
Time to pick matching tuples in subarrays = (p1+p2)*c*(N1/k)*(N2/k)
Time to write tuples to separate result subarrays = (p1+p2)*(N1*N2*2*w)/(k*V(A))
Time to write tuples to result array = (N1*N2*2w)/(k*V(A))

Total Time

97
The condition that the binary search (including its sorting time) performs better is:
\[
C \times N_2 \times 2w > (1/3) \times (c+1) \times N_2 \times 2w \times \log(N_2/k)
\]
where we have assumed a Quicksort. The binary search should perform better if the cardinality of each partition of a processor is sufficiently large.

**Two-Linked Joins in Parallel**

In two-linked join \((R_1 \times R_2) \times R_3\), we assume that the number of tuples in the final result relation is \(N_1 \times N_2 \times N_3 / (V(A) \times V(B))\) where \(A\) and \(B\) are the join attributes of two joins.

**The first join**

- Time to hash \(R_1\) tuples: \(h \times N_1\)
- Time to hash \(R_2\) tuples: \(h \times N_2\)
- Time to write \(R_1\) tuples in subarrays: \(w \times N_1 \times p\)
- Time to write \(R_2\) tuples in subarrays: \(w \times N_2 \times p\)
- Time to pick matching tuples in subarrays: \((p_1 + p_2) \times (N_1 \times N_2 \times 2w) / (k \times V(A))\)
- Time to write tuples to separate result subarrays: \((p_1 + p_2) \times (N_1 \times N_2 \times 2w) / (k \times V(A))\)
- (The presence of factor 2 above makes the length of a joined tuple double the original)

**The Second join**

- Time to hash \(R_3\) tuples: \(h \times N_3\)
- Time to hash tuples from the 1st join: \(h \times (N_1 \times N_2) / V(A)\)
- Time to write \(R_3\) tuples in subarrays: \(w \times N_3 \times p\)
- Time to write tuples from 1st join in subarrays: \(w \times (N_1 \times N_2) / V(A) \times (p_1 + p_2)\)
- Time to pick matching tuples in subarrays: \((p_1 + p_2 + p_3) \times (N_1 \times N_2 \times N_3 \times 2w) / (k \times V(A) \times V(B))\)
- Time to write tuples to separate result subarrays: \((p_1 + p_2 + p_3) \times (N_1 \times N_2 \times N_3 \times 2w) / (k \times V(A) \times V(B))\)
- Time to write tuples to result array

**Two-separate joins in parallel**

We assume that the first join is carried out by \(k_1\) processes and the second join by \(k_2\) processes.

**Time for the \(R_1 \times R_2\) join**

- Time to hash \(R_1\) tuples: \(h \times N_1 + h \times N_2 + w \times N_1 \times p_1 + w \times N_2 \times p_2 + (p_1 + p_2) \times c \times (N_1 \times N_2) / (k \times V(A))\)
- Time to write tuples to result array: \((N_1 \times N_2 \times 2w) / (k \times V(A))\)

**Time for the \(R_3 \times R_4\) join**

- Time to hash \(R_3\) tuples: \(h \times N_3 + h \times N_4 + w \times N_3 \times p_3 + w \times N_4 \times p_4 + (p_3 + p_4) \times c \times (N_3 \times N_4) / (k \times V(C))\)
- Time to write tuples to result array: \((N_3 \times N_4 \times 2w) / (k \times V(C))\)

**Three separate joins in parallel**

We assume that the three joins are carried out by \(k_1\), \(k_2\) and \(k_3\) processes respectively.

**Time for the \(R_1 \times R_2\) join**

- Time to hash \(R_1\) tuples: \(h \times N_1 + h \times N_2 + w \times N_1 \times p_1 + w \times N_2 \times p_2 + (p_1 + p_2) \times c \times (N_1 \times N_2) / (k \times V(A))\)
- Time to write tuples to result array: \((N_1 \times N_2 \times 2w) / (k \times V(A))\)

**Time for the \(R_3 \times R_4\) join**

- Time to hash \(R_3\) tuples: \(h \times N_3 + h \times N_4 + w \times N_3 \times p_3 + w \times N_4 \times p_4 + (p_3 + p_4) \times c \times (N_3 \times N_4) / (k \times V(C))\)
- Time to write tuples to result array: \((N_3 \times N_4 \times 2w) / (k \times V(C))\)
Time for the R5*R6 join

\[ h \times N5 + h \times N6 + w \times N5 \times p5 + w \times N6 \times p6 \]

\[ + \frac{(p5+p6) \times e \times ((N5 \times k3) \times (N6 \times k3))}{(N5 \times N6 \times w) / (k3 \times V(E))} \]

\[ + \frac{(p5+p6) \times (N5 \times N6 \times 2w) / (k3 \times V(E))}{(N5 \times N6 \times 2w) / (k3 \times V(E))} \]

5. PERFORMANCE EVALUATION

We have developed and implemented algorithms for the execution phases discussed in the last section. In each case we varied the cardinality of the object relation, as shown in the following figures. The cardinality of the target relation is kept fixed at 100 tuples, except where stated otherwise.

Figure 2 shows the time in ms (milliseconds) for a single join operation: (i) non-parallel execution (ii) parallel execution with sequential search and (iii) parallel execution with binary search. As indicated earlier a cluster of ten processors was used in the parallel executions.

In our non-parallel algorithm, we carried out joins by sequential search in the primary memory, but using only a single processor. Clearly performance is better for parallel search, although only by a factor of about 2 for 10K tuples. The divergence of the curves implies greater efficiency for larger cardinality. The fact that the that the sequential search performs better than the binary search implies that the partition sizes are too small for the binary search to be profitable, as indicated earlier. As a result, we did not use binary search any further.

To estimate the effect of the fork function, which is used to create parallel processes, the percentage of time used by this function against cardinality of the first relation is shown in figure 3, for a single join operation. Its effect becomes less significant as the cardinality increases, with about 4% for 10K tuples, but around 25% for 1K tuples.

We have also studied the impact of the reduced relations used in parallel joins. Full relations not only consume more memory, but they also increase read/write operations, as shown in figure 4 for 7 and 14 attributes in the final result relations (equi-join). In the curve marked X, two relations of degree 3 (target) and 4 (object) were joined, whereas in the curve marked 2X, the degrees were 6 and 8 respectively. However, in their reduced forms, each relation had only two attributes for both X and 2X (the lower two curves). It should also be mentioned that there was no significant extra cost in the production of the reduced relations; instead of copying all the attributes into the partitions, we copied less attributes. The advantages of using reduced relations is quite obvious; it cuts down the processing time to half. When the number of attributes increases to 14, the performance with reduced relations worsens only slightly; this is to be expected, since the full relations are accessed less frequently - only at the beginning and at the end of the processing. However, one might expect the performance to worsen considerably when full relations with 14 attributes, compared to full relations with 7 attributes, are used in place of reduced relations. Our results show only slightly worse performance, but not as bad as we expected. The likely explanation for this is that the number of read/write accesses depends only partially on the degree of a relation. However, we intend to investigate this a little further.
The effect of using a larger number of partitions, with dynamic scheduling, was also studied for the single join of figure 3, with an algorithm which has more pairs of subarrays (partitions) than the number of processors. In this algorithm, as soon as a process finishes joining one pair of subarrays, it starts joining the next available pair. The execution time in milliseconds for a given join operation was recorded for ratio \( R \) where \( R \) is the number of subarrays divided by the number of processors. The table below shows execution times for different ratios for different sets of data.

<table>
<thead>
<tr>
<th>Ratio R</th>
<th>1:1</th>
<th>1:2</th>
<th>1:3</th>
<th>1:4</th>
</tr>
</thead>
<tbody>
<tr>
<td>no. of tuples</td>
<td>100</td>
<td>1000</td>
<td>10000</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>0.55</td>
<td>1.47</td>
<td>10.67</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>0.61</td>
<td>1.51</td>
<td>10.44</td>
<td></td>
</tr>
<tr>
<td>10000</td>
<td>0.63</td>
<td>1.52</td>
<td>10.63</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.66</td>
<td>1.50</td>
<td>10.39</td>
<td></td>
</tr>
</tbody>
</table>

Clearly, the allocation of more than one subarray per processor does not lead to any savings in execution time for up to 1K or so tuples. The scheduling overhead seems to outweigh the gain from the evenness of a distribution. Only at 10K tuples there is some, though insignificant, gain in performance. Therefore our choice of 1:1 ratio which we used for our evaluation was not a bad one.

Figure 5 shows the performance of two-separate joins of phase 1 of step 3 against two one-join algorithms of step 1. Recall that in a two-separate join algorithm (\( S_1 := R_1 \ast R_2 \), and \( S_2 := R_4 \ast R_5 \)) we used two clusters, each with five processors, whereas in a two one-join algorithm we used a single cluster of 10 processors for both the joins, which were executed in sequence, first \( R_1 \ast R_2 \) and then \( R_4 \ast R_5 \). The two one-join algorithm is marginally better, as it has, we assume, less scheduling and control overheads.

The performance of three-separate joins with three clusters of 4, 3 and 3 processors (phase 1 step 4), shown in figure 6, is worse than that of three one-joins with a single cluster of 10 processors, again we assume due to scheduling and control overheads. It would appear that three or four or five processors per cluster are too few to offset overheads.

We also compared the performance of two-linked join algorithms \((R_1 \ast R_2) \ast R_3) \ast R_4 \) against three one-join algorithms \( S_1 := R_1 \ast R_2 \), \( S_2 := R_3 \ast R_4 \), \( S_3 := S_1 \ast S_2 \), both using ten processors. Their performances were more or
less identical, although we have not shown the results here.

6. CONCLUSION

From the results presented, it is quite clear that the best strategy is to form a single cluster of parallel processors and to carry out the multi-join as a series of single joins one after another. Ideally we should have used a better hashing technique and sort/merge (rather than sequential search) in each processor. However, we do not think these would have changed our results in any significant fashion.

The reader might have noticed that we have created clusters somewhat artificially, and with an additional overhead, particularly since the inter-cluster communication cost is the same as intra-cluster communication cost. If the inter-cluster communication cost higher than intra-cluster communication cost, then the result would be different, perhaps favouring distribution into clusters. It is not suggested that one should artificially increase the inter-cluster communication cost; the point is that if the machine architecture is such that it is naturally divided into clusters with higher inter-cluster communication cost, and if a cluster can read directly - currently with other clusters - from secondary memory, then one must consider the use of clusters.

The results presented here are preliminary, and there are number of obvious limitations in what we did. We could have reduced the sequential processing phase further by letting the slave processes to hash and create partitions, however, it would not have saved the hashing time, since each process will have to hash the whole relation to find its share of tuples. Only the partitioning time could have been saved, but then this would have made the partitions exclusive, and unavailable to other processes for subsequent joins, if any. So, this is not a good option, unless used for only single joins. A more serious criticism is the two linked join, where time could have been saved by not creating the intermediate result relation, but joining it with R3 by pipelining. We also assume the memory size to be infinite, as we did not consider any secondary storage.

The work also highlights a number of parameters that are important in multi-join by parallel processors. These are:

- sequential processing overheads
- scheduling and control overheads
- advantages of reduced relations
- inter-cluster communication cost (i.e. do not use clusters if the inter-cluster communication cost is the same as intra-cluster communication cost)
- cost of sorting in memory
- 1:1 versus 1:Many ratio in processors to partitions
- cardinality

We hope to carry out a further study with a larger number of tuples in both object and target relations, and also varying the number of tuples in both. The other issues we need to consider are the use of a larger number of processes, memory overflow, different hashing techniques, dynamic scheduling (it could be interesting when a large number of large relations and a large number of processes are involved). Pipelining could also be considered, but in Sequent it is slower than a shared
memory approach; and therefore its use will be limited to special cases. We also intend to vary the inter-cluster/intra-cluster communication time artificially to assess its effect.

Acknowledgements

We wish to thank Brenda Banks and Alan Trehune for reading through the manuscript and for making valuable comments. Thanks also to Atula Herath for his assistance with the Sequent configuration.

REFERENCES


